

DANH MỤC CÁC TỪ VIẾT TẮT

Từ viết tắt	Ý nghĩa
IDE	Integrated Development Environment
Java API	Java Application Programming Interface
Java EE	Java platform, Enterprise Edition
Java ME	Java platform, Micro Edition
Java SE	Java platform, Standard Edition
JDK	Java Development Kit
JVM	Java Virtual Machine
OOP	Object Oriented Programming
ORD	Objecct Relationship Diagram
P.I.E	Polymorphism - Inheritance – Encapsulation
POP	Procedure Oriented Programming

DANH MỤC BẢNG BIỂU

Bảng 1.1. So sánh phương thức khởi tạo và phương thức thông thường.. ..	31
Bảng 1.2. 8 kiểu dữ liệu nguyên thủy trong Java.....	45
Bảng 1.3. Các toán tử số học trong Java.....	51
Bảng 1.4. Các toán tử quan hệ trong Java.....	51
Bảng 1.5. Các toán tử logic trong Java.	52
Bảng 1.6. Toán tử thao tác bít trong Java.	53
Bảng 1.7. Thứ tự ưu tiên của các toán tử.	56
Bảng 2.1. Các mức truy nhập đối tượng.	102
Bảng 2.2. Các mức truy nhập thành phần đối tượng.	104
Bảng 3.1. Bảng tổng hợp các quyền truy cập giữa các lớp.....	150
Bảng 4.1. Danh sách một số ngoại lệ trong Java.	205
Bảng 5.1. Bảng các phương thức của lớp String.	233
Bảng 5.2. Các phương thức của Math.....	242
Bảng 5.3. Bảng các Interface collection.	246
Bảng 5.4. Các phương thức của interface collection.	247
Bảng 5.5. Các lớp collection.	251
Bảng 5.6. Bảng các phương thức mô tả thuật toán trong collection.....	266
Bảng 5.7. Một số phương thức của Iterator.	268
Bảng 5.8. Một số phương thức của ListIterator.	269
Bảng 6.1. Bảng các phương thức trong vòng lặp của Applet.....	287
Bảng 6.2. Các thành phần cơ bản của gói Swing.....	296
Bảng 6.3. Các mẫu thiết kế cụ thể trong từng nhóm.....	328

DANH MỤC HÌNH VẼ

Hình 1.1. Ví dụ mối quan hệ đặc biệt hóa, tổng quát hóa giữa 2 lớp User và Admin.....	14
Hình 1.2. Người khởi xướng Java - Jame Gosling và biểu tượng Java.....	16
Hình 1.3. Quá trình biên dịch 1 chương trình Java.....	18
Hình 1.4. Kiến trúc JVM.....	19
Hình 1.5. Các Tags trong Javadoc.	42
Hình 1.6. Các kiểu dữ liệu nguyên thủy.	43
Hình 2.1. Đóng gói thuộc tính và hành vi của đối tượng.....	83
Hình 2.2. Đối tượng A gửi thông điệp tới đối tượng B.	85
Hình 2.3. Cấu trúc tổ chức chương trình Quản lý điểm.....	96
Hình 2.4. Các lớp trong chương trình Quản lý điểm.	97
Hình 2.5. Gói thư viện Java.	98
Hình 2.6. Minh họa truy cập các thành phần public.	102
Hình 2.7. Minh họa truy cập các thành phần protected.	103
Hình 2.8. Minh họa truy cập các thành phần mặc định.	103
Hình 2.9. Minh họa truy cập các thành phần private.....	104
Hình 2.10. Ví dụ thiết kế gói.....	105
Hình 2.11. Thiết kế lớp CanBo.	107
Hình 2.12. Thiết kế tổ chức chương trình quản lý điểm.	110
Hình 2.13. Minh họa tham chiếu đối tượng.	114
Hình 2.14. Kết quả chạy chương trình.	119
Hình 2.15. Biến count được dung chung cho mọi thể hiện của Lop.	120

Hình 2.16. Phương thức tĩnh có thể gọi trực tiếp từ tên lớp.....	121
Hình 3.1. Mô hình quan hệ has-a: Apple có 1 thuộc tính là 1 đối tượng tham chiếu tới lớp Fruit.....	127
Hình 3.2. Quan hệ <i>is-a</i> : Táo là 1 loại quả. Apple cũng là Fruit.....	129
Hình 3.3. Biểu diễn 1 lớp.....	133
Hình 3.4. Cây thừa kế Người - Cán bộ - Sinh viên.....	136
Hình 3.5. Cây thừa kế dữ liệu kiểu số trong Java.....	137
Hình 3.6. Cây thừa kế của gói AWT của Java.....	138
Hình 3.7. Biểu diễn lớp cha.	139
Hình 3.8. Biểu diễn lớp con.	142
Hình 3.9. Trình tự gọi constructor khi có thừa kế.	147
Hình 3.10. In danh sách cán bộ và danh sách sinh viên.	149
Hình 3.11. Kết quả chạy chương trình với ẩn danh.	152
Hình 3.12. Cây thừa kế hình học.	158
Hình 3.13. Hai đối tượng hiểu một thông điệp theo 2 cách riêng.	163
Hình 3.14. Liên kết lời gọi hàm trong lập trình có cấu trúc	166
Hình 3.15. Kết quả chạy Circle.....	171
Hình 3.16. Cài thêm <i>Rectangle</i>	172
Hình 4.1. Cây phân thừa kế các lớp điều khiển dòng nhập.	186
Hình 4.2. Cây thừa kế các lớp điều khiển dòng xuất.	186
Hình 4.3. Cây thừa kế các lớp đọc dữ liệu luồng kí tự.	187
Hình 4.4. Cây thừa kế các lớp xuất dữ liệu luồng kí tự.	188
Hình 4.5 Hệ thống phân cấp các lớp ngoại lệ trong Java.	202

Hình 5.1. Một số gói và lớp của java và javax.....	224
Hình 5.2. Sử dụng hàm Scanner và nextLine() để nhập chuỗi.....	234
Hình 5.3. Kết quả so sánh xâu.....	236
Hình 5.4. Sơ đồ khuôn mẫu collection.....	245
Hình 5.5. Các kí tự đại diện.....	280
Hình 5.6. Thừa kế trên nhiều lớp.....	281
Hình 6.1. Chạy applet với <i>AppletViewer</i>	288
Hình 6.2. Vòng đời của applet.	289
Hình 6.3. Kết quả chạy InitStartStop.	290
Hình 6.4. Các thành phần khác nhau tạo nên một giao diện người dùng.	294
Hình 6.5. Sơ đồ phân cấp các lớp trong gói AWT.....	295
Hình 6.6. Các siêu lớp phổ biến của các thành phần Swing.....	295
Hình 6.7. Kết quả chương trình minh họa JFrame.....	298
Hình 6.8: Ví dụ tạo JLabel.	301
Hình 6.9: Ví dụ cách tạo JTextField.	304
Hình 6.10. Sơ đồ thể hiện sự kế thừa của các button trong jComponent.	305
Hình 6.11. Kết quả chương trình minh họa JButton.....	306
Hình 6.12. Kết quả chạy thử nghiệm minh họa JComboBox.....	308
Hình 6.13. Kết quả thu được với bố cục FlowLayout.	311
Hình 6.14. Kết quả thu được với bố cục BorderLayout.....	312
Hình 6.15: Kết quả minh họa bố cục GridLayout.....	314
Hình 6.16. Những lớp sự kiện của gói java.awt.events.	315
Hình 6.17. Các giao diện lắng nghe sự kiện của gói java.util.event.....	316

Hình 6.18. Kết quả kiểm thử chương trình MouseTrakerFrame.	319
Hình 6.19. Kết quả thực thi chương trình KeyDemoFrame.	322
Hình 6.20. Mô hình cấu trúc mẫu Abstract Factory.	329
Hình 6.21. Các lớp và phương thức xây dựng một trình soạn thảo văn bản.	331
Hình 6.22. Mô hình cấu trúc mẫu <i>factory method</i>	332
Hình 6.23. Mô hình cấu trúc mẫu Adapter 1.	334
Hình 6.24. Mô hình cấu trúc mẫu Adapter 2.	334
Hình 6.25. Mô hình cấu trúc mẫu Façade.	337
Hình 6.26. Mối quan hệ giữa các danh sách.	338
Hình 6.27. Mô hình cấu trúc mẫu Iterator.	339
Hình 6.28. Mô hình cấu trúc mẫu Observer.....	341

DANH MỤC THUẬT NGỮ

Thuật ngữ	Ý nghĩa
abstract	Trùu tượng, từ khóa
Base class	Lớp cơ sở, cũng là lớp cha
class	Lớp, từ khóa
Compile - time	Trong thời gian biên dịch (chương trình)
Constructor	Phương thức khởi tạo
Definition/ Declaration	Khai báo / Định nghĩa (hay dùng khi viết biến, hàm, phương thức)
Derived class	Lớp dẫn xuất, cũng là lớp con
Downcast - downcasting	Chuyển kiểu xuống
Dynamic	Động, trái ngược với Static là tĩnh
Early binding	Liên kết sớm
Exception	Ngoại lệ
extends	Từ khóa mang nghĩa thừa kế từ lớp cha
final	Hằng số, là từ khóa
Function	Hàm
Function call binding	Liên kết lời gọi hàm
implements	Cài đặt, từ khóa, để khai báo một lớp cài đặt giao diện
Inherit - Inheritance	Thừa kế
Instance	Thể hiện - là một biến kiểu dữ liệu tham chiếu đã được cấp phát
instanceof	Toán tử, xác định đối tượng thuộc kiểu lớp đối tượng nào
interface	Giao diện, từ khóa, là khai báo một kiểu lớp

member	Chỉ các thành phần bên trong một lớp như là thuộc tính hay phương thức
method	Phương thức
Method call bidding	Liên kết lời gọi phương thức
ORD	Object Relationship Diagram - Sơ đồ quan hệ đối tượng
Overload - overloading	Nạp chồng
Override - Overriding	Vượt quyền/ ghi đè
Parametric - Parameter	Tham số
Polymorphism	Đa hình
private	Riêng, là quyền truy cập
protected	Được bảo vệ, là quyền truy cập
public	Công khai, là quyền truy cập
Run - time	Trong thời gian chạy (chương trình)
Signature	Chữ ký
static	Tĩnh, là từ khóa
Sub class	Lớp con
super	Toán tử, tham chiếu lên thành phần ở lớp cha
Super class	Lớp cha
Upcast - upcasting	Chuyển kiểu lên

LỜI NÓI ĐẦU

Ngôn ngữ lập trình là công cụ cần thiết để tạo ra các chương trình máy tính nhằm giải quyết nhiều bài toán khác nhau. Trải qua quá trình phát triển, đã có nhiều ngôn ngữ lập trình được xây dựng, từ dạng mã máy, hợp ngữ cho đến các ngôn ngữ lập trình bậc cao được sử dụng phổ biến hiện nay. Ngôn ngữ lập trình Java là một ví dụ điển hình của quá trình tiến hóa của ngôn ngữ lập trình, là một trong số các ngôn ngữ lập trình bậc cao, hướng đối tượng được sử dụng phổ biến trong cộng đồng lập trình nói riêng và các công ty phát triển ứng dụng nói chung.

Giáo trình *Lập trình hướng đối tượng với JAVA* được biên soạn theo đề cương chi tiết học phần Lập trình hướng đối tượng với Java đã được ban hành trong chương trình đào tạo ngành CNTT và ATTT tại Học viện ANND. Giáo trình cung cấp cho học viên những kiến thức cơ bản nhất về nguyên lý lập trình bằng ngôn ngữ Java, qua đó giúp học viên có được một nền tảng vững chắc về Java và có thể dùng ngôn ngữ này để phát triển các ứng dụng phần mềm phục vụ học tập và công tác sau này. Bộ cục của giáo trình gồm 6 chương:

- Chương 1. Giới thiệu lập trình hướng đối tượng và ngôn ngữ Java: Các hướng tiếp cận trong lập trình; khái niệm và các tính chất cơ bản của lập trình hướng đối tượng; lịch sử, kiến trúc và đặc trưng ngôn ngữ lập trình Java, các ứng dụng và công cụ lập trình Java, các thành phần và thao tác lập trình Java cơ bản.
- Chương 2. Lớp và đối tượng: Trừu tượng hóa đối tượng và đóng gói dữ liệu vào lớp; khai báo và sử dụng lớp, đối tượng.
- Chương 3. Thừa kế và đa hình: Giới thiệu về thừa kế, biểu diễn quan hệ thừa kế, cài đặt thừa kế trong Java; các loại đa hình, đa hình và liên kết phương thức, cài đặt đa hình trong Java, mở rộng đa thừa kế bằng giao diện.
- Chương 4. Các luồng vào ra và xử lý ngoại lệ: Khái niệm và phân loại luồng vào ra, thao tác luồng vào ra; khái niệm và phân loại ngoại lệ, các kiểu xử lý ngoại lệ.

- Chương 5. Lớp cơ sở và lập trình tổng quát: Giới thiệu một số lớp cơ sở; giới thiệu lập trình tổng quát, lớp tổng quát, phương thức tổng quát, các ứng dụng của lập trình tổng quát và hạn chế của nó
- Chương 6: Giới thiệu lập trình giao diện và mẫu thiết kế: Giới thiệu Applet và lập trình giao diện GUI; các mẫu thiết kế phổ biến.

Giáo trình **Lập trình hướng đối tượng với Java** có sự tham gia biên soạn của 04 tác giả, l [REDACTED]
[REDACTED]
[REDACTED]

[REDACTED] Mặc dù các tác giả đã hết sức cố gắng, song do biên soạn lần đầu nên không tránh khỏi những thiếu sót. Kính mong đồng nghiệp và bạn đọc đóng góp ý kiến để giáo trình được hoàn thiện hơn.

Nhóm tác giả

Chương 1. GIỚI THIỆU LẬP TRÌNH HƯỚNG ĐÓI TƯỢNG VÀ NGÔN NGỮ JAVA

I. GIỚI THIỆU LẬP TRÌNH HƯỚNG ĐÓI TƯỢNG

1. Các cách tiếp cận trong lập trình

a) *Lập trình tuyến tính*

Lập trình tuyến tính là phương pháp lập trình truyền thống, trong đó chương trình được thực hiện theo thứ tự từ đầu đến cuối, lệnh này kế tiếp lệnh kia cho đến khi kết thúc chương trình. Trong lập trình tuyến tính chỉ có duy nhất một luồng công việc và các phần việc được thực hiện tuân tự trong luồng đó.

Ưu điểm của lập trình tuyến tính là chương trình đơn giản, dễ hiểu. Tuy nhiên, nhược điểm của lập trình tuyến tính là không thể áp dụng cho các chương trình phức tạp. Do đó, lập trình tuyến tính ngày nay chỉ được sử dụng trong phạm vi các mô-đun nhỏ nhất của các phương pháp lập trình khác.

b) *Lập trình hướng cấu trúc*

Lập trình hướng cấu trúc hay còn gọi là lập trình hướng thủ tục (Procedure Oriented Programming - POP) là một kỹ thuật lập trình truyền thống, trong đó sử dụng cách tiếp cận top-down trong thiết kế chương trình. Điều này có nghĩa là một chương trình lớn được chia thành các hàm (chương trình con), mỗi chương trình con có thể được gọi tới một hoặc nhiều lần theo thứ tự bất kỳ.

Ưu điểm của lập trình hướng cấu trúc là tư duy giải thuật rõ ràng, chương trình đơn giản, dễ hiểu. Tuy nhiên, nhược điểm của nó là không phù hợp với các bài toán lớn với kiểu dữ liệu phức tạp, trừu tượng.

Một số ngôn ngữ lập trình hướng cấu trúc như Pascal, C...

c) *Lập trình hướng đối tượng*

Lập trình hướng đối tượng (Object Oriented Programming – OOP) là một trong những kỹ thuật lập trình quan trọng hiện nay, đã trở thành một trong những

khuôn mẫu phát triển phần mềm. Phần lớn các ngôn ngữ lập trình (Java, C#, Python, Ruby, Swift, Object-C) và framework đều hỗ trợ lập trình hướng đối tượng.

Lập trình hướng đối tượng cho phép biểu diễn trong ngôn ngữ lập trình các đối tượng tự nhiên với các hành động và đặc tính của chúng. Ví dụ **sinh viên** là một đối tượng trong đời sống tự nhiên, với các đặc tính như **họ tên, ngày tháng năm sinh, quê quán**. Đối tượng này khi được biểu diễn bởi lập trình hướng đối tượng sẽ là một lớp **class sinhvien** (biến **hoten**, biến **namsinh**, biến **quequan**). Nói cách khác, lập trình hướng đối tượng cho phép chúng ta phân tích các bài toán thành các thực thể gọi là các đối tượng và xây dựng các dữ liệu, các hàm xung quanh các đối tượng này. Dữ liệu được liên kết với các hàm thành các vùng riêng và chỉ có các hàm đó tác động lên, các hàm bên ngoài không được truy cập vào. Các đối tượng có thể tác động và trao đổi thông tin với nhau thông qua các thông điệp.

Lập trình hướng đối tượng ra đời sau nên nó khắc phục được tất cả các điểm yếu của các phương pháp lập trình trước đó. Cụ thể nó có các ưu điểm sau:

- Dễ dàng quản lý mã chương trình (code) khi có sự thay đổi chương trình.
- Dễ mở rộng dự án.
- Tiết kiệm được tài nguyên đáng kể cho hệ thống.
- Có tính bảo mật cao.
- Có tính tái sử dụng cao.

2. Các khái niệm và tính chất cơ bản của lập trình hướng đối tượng

a) Các khái niệm cơ bản

- Đối tượng (Object):

Trong lập trình hướng đối tượng, đối tượng được hiểu như là 1 thực thể: người, vật hoặc 1 bảng dữ liệu... Mỗi đối tượng đều có 2 thành phần: thuộc tính và phương thức. Thuộc tính là những thông tin, đặc điểm của đối tượng. Ví dụ:

một người sẽ có họ tên, ngày sinh, màu da, kiểu tóc... Phương thức là những thao tác, hành động mà đối tượng đó có thể thực hiện. Ví dụ: một người sẽ có thể thực hiện hành động nói, đi, ăn, uống...

- Lớp (Class):

Các đối tượng có các đặc tính tương tự nhau được nhóm lại thành một lớp đối tượng. Bên trong lớp cũng có 2 thành phần chính đó là thuộc tính và phương thức. Ngoài ra, lớp còn được dùng để định nghĩa kiểu dữ liệu mới.

Sự khác nhau giữa đối tượng và lớp thể hiện ở chỗ lớp là một khuôn mẫu còn đối tượng là một thể hiện cụ thể dựa trên khuôn mẫu đó. Ví dụ ta có lớp ***sinh viên***, với các thông tin như *họ tên, ngày tháng năm sinh, quê quán...*. Còn đối tượng là một sinh viên cụ thể nào đó, ví dụ họ tên ***Nguyễn Văn A***, sinh ngày ***13 tháng 4 năm 1985***, quê quán ***Hà nội***.

Một trường hợp cụ thể của lớp là Instance. Tất cả các instance của một lớp có các thuộc tính tương tự như mô tả trong lớp. Ví dụ: ta có thể định nghĩa một lớp "Sinhvien" và tạo 3 instances của lớp Sinhvien là "Dung", "Hong" và "Thu".

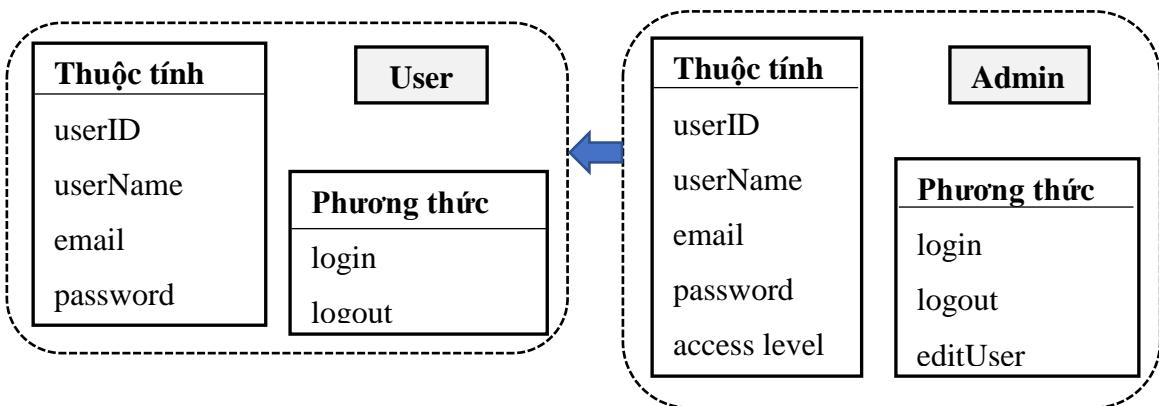
b) Các tính chất cơ bản của lập trình hướng đối tượng

- Trừu tượng hóa (Abstraction)

Trùu tượng hóa là việc mô tả một đối tượng với những thuộc tính, phương thức cần thiết cho việc giải quyết bài toán đang lập trình mà không nhất thiết phải mô tả đối tượng với tất cả các thuộc tính, phương thức của đối tượng đó. Một đối tượng có rất nhiều thuộc tính, phương thức, nhưng tùy từng bài toán cụ thể ta sẽ phải thực hiện trừu tượng hóa để chọn ra những thuộc tính, hành động của đối tượng phù hợp với chương trình. Ví dụ với bài toán quản lý sinh viên cần thuộc tính "*mã sinh viên*", "*họ tên*", "*ngày sinh*", "*quê quán*", "*học lớp*", mà không cần các thuộc tính khác như "*sở thích*", "*cân nặng*", "*chiều cao*",..; bài toán quản lý phạm nhân cần những thuộc tính như "*tính chất tội phạm*", "*mức án*",.., "*đặc điểm thân nhân*".

- Ké thừa (inheritance)

Trong lập trình hướng đối tượng, các lớp được trừu tượng hóa và tổ chức thành một sơ đồ phân cấp lớp (ví dụ **Error! Reference source not found.**), khi đó tính kế thừa để chỉ mối quan hệ đặc biệt hóa – tổng quát hóa giữa các lớp. Đây là mối quan hệ giữa hai lớp đối tượng mà đối tượng này là trường hợp đặc biệt của đối tượng kia và đối tượng kia là trường hợp tổng quát của đối tượng này. Trong ví dụ ở **Error! Reference source not found.** thì lớp *Admin* là trường hợp đặc biệt của lớp *User* và lớp *User* là trường hợp tổng quát của lớp *Admin*.



Hình 1.1. Ví dụ mối quan hệ đặc biệt hóa, tổng quát hóa giữa 2 lớp User và Admin.

Tính kế thừa trong lập trình hướng đối tượng cho phép một lớp có thể kế thừa các thuộc tính và phương thức từ các lớp khác đã được định nghĩa. Điều này cho phép các đối tượng có thể tái sử dụng hay mở rộng các thuộc tính sẵn có mà không phải tiến hành định nghĩa lại, từ đó giúp tối ưu được thời gian, công sức phát triển ứng dụng, đồng thời cũng dễ dàng nâng cấp và chỉnh sửa mã nguồn.

Lớp được kế thừa còn được gọi là lớp cha (superclass) hay lớp cơ sở (base class) và lớp kế thừa được gọi là lớp con (subclass) hay lớp dẫn xuất (derived class). Nhiều lớp có thể dẫn xuất từ một lớp cha và một lớp có thể là dẫn xuất của nhiều lớp cha. Ví dụ **Error! Reference source not found.**, lớp cha là lớp *User* và lớp con là lớp *Admin*. Lớp *User* có các thuộc tính (*userID*; *userName*; *email*; *password*) và phương thức (*login*, *logout*). Lớp *Admin* kế thừa các thuộc tính và

phương thức này của lớp *User* và mở rộng thêm với thuộc tính access level và phương thức editUser.

- Đóng gói (*encapsulation*)

Đóng gói mô tả ý tưởng về việc gói dữ liệu (biến, trạng thái) và các phương thức hoạt động trên dữ liệu đó trong một đơn vị (lớp). Khi được đóng gói, dữ liệu không được truy cập trực tiếp mà được truy cập thông qua các phương thức hiện diện bên trong lớp. Mỗi lớp được xây dựng để thực hiện một nhóm chức năng đặc trưng riêng của lớp, trong trường hợp một đối tượng thuộc lớp cần thực hiện một chức năng không nằm trong khả năng vì chức năng đó thuộc về một đối tượng thuộc lớp khác, thì nó sẽ yêu cầu đối tượng đó đảm nhận thực hiện công việc.

Một điểm quan trọng trong cách giao tiếp giữa các đối tượng là một đối tượng sẽ không được truy xuất trực tiếp vào thành phần dữ liệu của đối tượng khác cũng như không đưa thành phần dữ liệu của mình cho đối tượng khác một cách trực tiếp. Tất cả mọi thao tác truy xuất vào thành phần dữ liệu từ đối tượng này qua đối tượng khác phải được thực hiện bởi các phương thức (method) của chính đối tượng chứa dữ liệu. Đây cũng chính là một tính chất quan trọng trong lập trình hướng đối tượng gọi là tính đóng gói (*encapsulation*) dữ liệu. Tính đóng gói cho phép giấu thông tin của đối tượng bằng cách kết hợp thông tin và các phương thức liên quan đến thông tin trong đối tượng. Ví dụ trong đối tượng kế toán có một chức năng tính lương, chức năng này có những phép tính mà các đối tượng khác không hề biết cách tính nó như thế nào. Đối tượng kế toán sẽ có những chức năng dành cho các hàm kế thừa nó có thể truy xuất vào thông qua các mức truy cập để giới hạn chúng. Đây cũng chính là bảo mật thông tin cho đối tượng. Một ví dụ minh họa khác đó là xe ô tô có các chức năng (phương thức phô diễn bên ngoài) như Ngừng, Chạy tới, Chạy lùi. Đây là những gì cần thiết cho tài xế tương tác với xe ô tô. Xe ô tô có thể có một đối tượng Động cơ nhưng tài xế không cần phải quan tâm đến đối tượng này. Tất cả những gì cần quan tâm là những chức năng để có thể vận hành xe. Do đó, khi thay một động cơ khác, tài xế vẫn sử dụng các

chức năng cũ để vận hành xe, các chức năng (phương thức) đưa ra bên ngoài (Interface) không bị thay đổi.

- *Đa hình (polymorphism)*

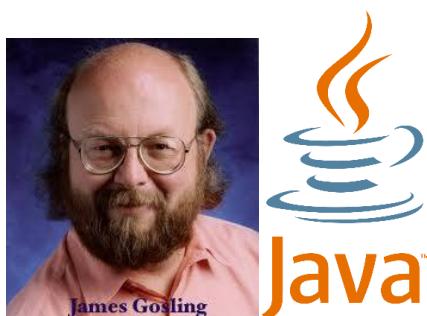
Đa hình là tại từng thời điểm đối tượng sẽ có các hình thái khác nhau trong những hoàn cảnh khác nhau. Ví dụ: Khi ở trường học bạn là sinh viên thì có nhiệm vụ nghe giảng, lên thư viện đọc sách... nhưng khi ở nhà bạn đóng vai trò là thành viên trong gia đình thì có nhiệm vụ làm việc nhà...khi bạn vào siêu thị thì bạn đóng vai trò là khách hàng mua hàng... Vì vậy, đa hình của đối tượng được hiểu là trong từng trường hợp, hoàn cảnh khác nhau thì đối tượng có khả năng thực hiện các công việc khác nhau.

II. NGÔN NGỮ LẬP TRÌNH JAVA

1. Lịch sử, kiến trúc và đặc trưng của ngôn ngữ Java

a) *Lịch sử Java*

Java được đề xuất bởi James Gosling và các đồng nghiệp ở Sun Microsystems năm 1991. Từ ý tưởng muốn lập trình để điều khiển mà không phụ thuộc vào loại CPU cho các thiết bị điện tử như tivi, máy giặt, lò nướng... họ đã xây dựng một ngôn ngữ chạy nhanh, gọn, hiệu quả, độc lập thiết bị gọi là ngôn ngữ “Oak”, sau này được đổi tên thành Java vào năm 1995. Java là tên một hòn đảo của Indonesia, là nơi đầu tiên sản xuất ra cà phê. Đó là lý do tại sao biểu tượng của Java là cốc cà phê bốc khói.



Hình 1.2. Người khởi xướng Java - Jame Gosling và biểu tượng Java.

Java được phát triển đầu tiên tại Sun Microsystems (hiện là công ty con của Oracle Corporation) và được công bố năm 1995 với phiên bản JDK Alpha và Beta. Cũng trong năm 1995, tạp chí Time bình chọn Java là một trong 10 sản phẩm tốt nhất năm 1995. Các năm sau đó, đã có nhiều phiên bản Java được tạo ra như JDK 1.0 (23/1/1996), JDK 1.1 (19/2/1997), J2SE 1.2 (8/12/1998), J2SE 1.3 (8/5/2000), J2SE 1.4 (6/5/2002), J2SE 5.0 (30/9/2004), Java SE 6 (11/12/2006), Java SE 7 (28/7/2011); Java SE 8 (18/3/2014); Java SE 9 (21/9/2017); và phiên bản Java mới hiện nay là Java SE 10 (20/3/2018).

b) Kiến trúc, môi trường và đặc trưng Java

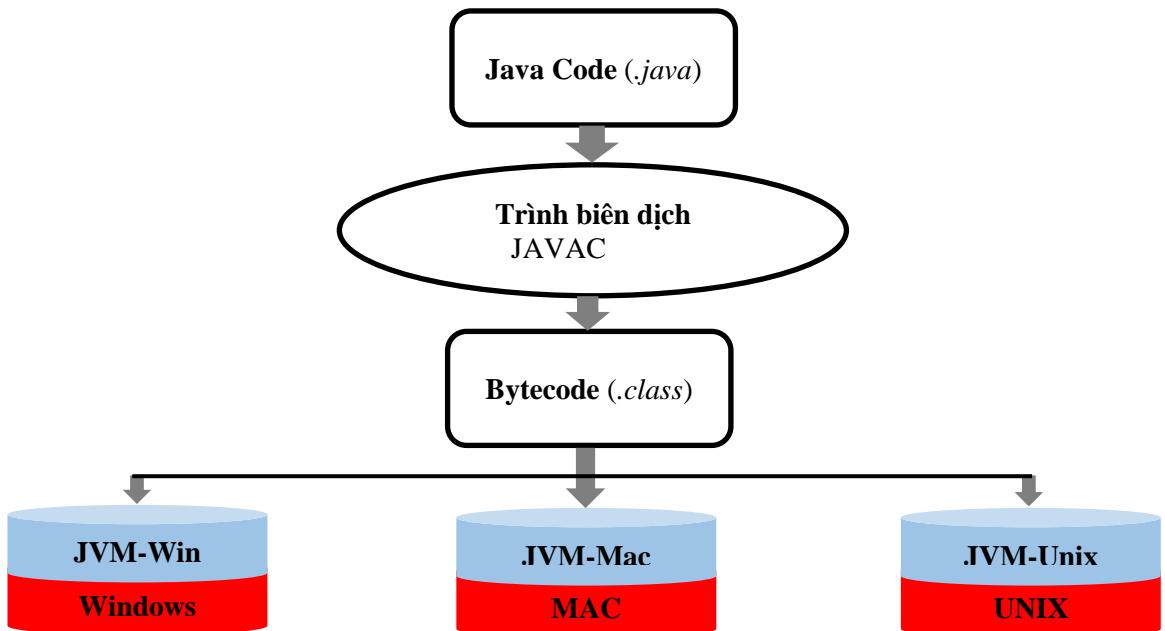
- Kiến trúc, môi trường Java:

Bất cứ môi trường phần cứng hoặc phần mềm nào mà trong đó có một chương trình chạy trên đó thì được hiểu như là một Platform. Java Platform gồm 3 thành phần chính:

- + Java Virtual Machine (JVM): Máy ảo Java.
- + Java Application Programming Interface (Java API): Giao diện lập trình ứng dụng Java.
- + Java Development Kit (JDK): Gồm trình biên dịch, thông dịch, trợ giúp, soạn thảo... và các thư viện chuẩn.

Máy ảo Java JVM: Để đảm bảo tính đa nền, có nghĩa là phần mềm viết ra có thể thực thi được trên các hệ điều hành khác nhau, Java sử dụng JVM để cung cấp môi trường thực thi cho chương trình Java. JVM thực chất là một trình thông dịch của Java. Một chương trình sau khi được viết bằng ngôn ngữ Java (có phần mở rộng là .java) phải được biên dịch thành tập tin thực thi được trên máy ảo Java (có phần mở rộng là .class). Tập tin thực thi này chứa các chỉ thị dưới dạng mã Bytecode, là ngôn ngữ máy của JVM. Khi thực hiện một chương trình, máy ảo Java lần lượt thông dịch các chỉ thị dưới dạng Bytecode thành các chỉ thị dạng nhị phân của máy tính thực và thực thi chúng trên máy tính thực. Các hệ điều hành

khác nhau sẽ có các máy ảo khác nhau để thực thi một ứng dụng Java trên hệ điều hành cụ thể.



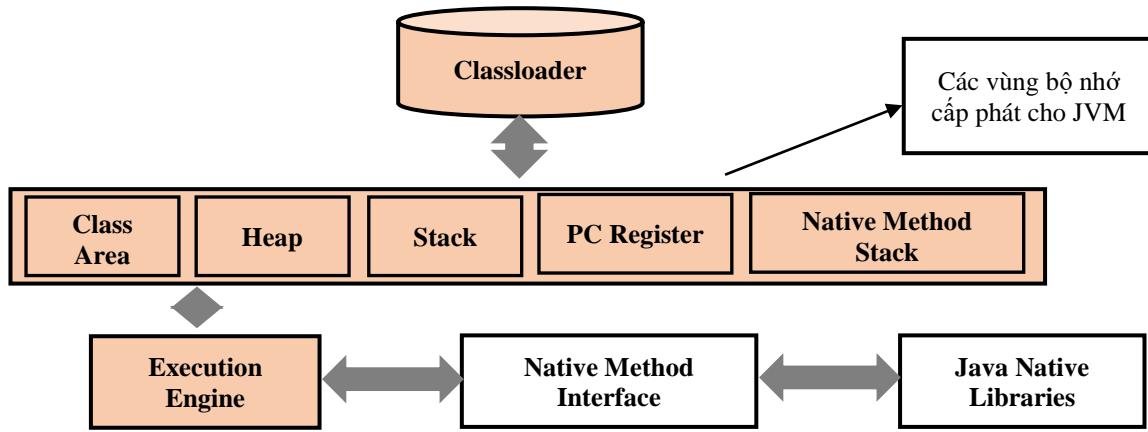
Hình 1.3. Quá trình biên dịch 1 chương trình Java.

Error! Reference source not found. minh họa quá trình nêu trên: một tệp chương trình java *.java* được viết theo mã Unicode, trình biên dịch JAVAC sẽ chuyển mã Unicode sang mã Bytecode của máy ảo Java (tệp tin *.class*). JVM sẽ dịch mã Bytecode thành các mã nhị phân của máy vật lý tương ứng với các hệ điều hành khác nhau.

JVM có 3 thành phần chính:

- + Class Loader: Tìm kiếm và tải các file *.class* vào vùng nhớ của java dưới dạng Bytecode.
- + Data Area: Vùng nhớ hệ thống cấp phát cho JVM.
- + Execution Engine: chuyển các lệnh của JVM trong file *.class* thành các lệnh của máy, hệ điều hành tương ứng và thực thi chúng.

Kiến trúc bên trong JVM được mô tả ở **Error! Reference source not found.**, với các thành phần chức năng cụ thể như sau:



Hình 1.4. Kiến trúc JVM.

- + Classloader là một hệ thống phụ của JVM được sử dụng để tải các file *class*. Sau khi Classloader làm xong nhiệm vụ của mình các file sẽ được máy ảo JVM cung cấp bộ nhớ tương ứng với chúng.
- + Class Area: Vùng nhớ cấp phát cho class (phương thức), được phân chia thành các vùng nhớ heap, stack, PC register, native method stack.
- + Heap: Là nơi các đối tượng khởi tạo bằng toàn từ "new" sẽ được lưu ở đây tại thời điểm chương trình chạy.
- + Stack: Các phương thức và tham chiếu tới đối tượng cục bộ được lưu trữ trong Stack. Mỗi Thread quản lý một stack. Khi phương thức được gọi, nó được đưa vào đỉnh của Stack. Stack lưu trữ trạng thái của phương thức bao gồm: dòng mã lệnh thực thi, tham chiếu tới đối tượng cục bộ. Khi phương thức chạy xong, vùng nhớ (dòng mã lệnh thực thi, tham chiếu tới đối tượng cục bộ) được đẩy ra khỏi stack và tự động giải phóng.
- + PC register: Khi JVM thực thi mã lệnh, một thanh ghi cục bộ có tên "Program Counter" được sử dụng. Thanh ghi này trả tới lệnh đang thực hiện. Khi cần thiết, có thể thay đổi nội dung thanh ghi để đổi hướng thực thi của chương trình. Trong trường hợp thông thường từng lệnh một nối tiếp nhau sẽ được thực thi.

+ Native method stack: Nơi chứa tất cả các method native trong chương trình.

+ Execution Engine: Chứa một bộ xử lý ảo, trình phiên dịch để đọc dòng Bytecode sau đó thực hiện các hướng dẫn, và trình biên dịch JIT (Just-In-Time) dùng để biên dịch đồng thời các phần của mã byte có chức năng tương tự nhau, từ đó giảm thời gian cần thiết cho việc biên dịch.

Giao diện lập trình ứng dụng Java - Java API: gồm nhiều lớp thư viện Java. Các lớp này làm đơn giản quá trình phát triển các ứng dụng Java cho người lập trình. Danh sách đầy đủ các lớp thư viện trong Java API có thể tra cứu tại trang web của Oracle: <https://docs.oracle.com/javase/7/docs/api/>.

Môi trường lập trình JDK gồm:

+ Java SE (Java platform, Standard Edition): Là một nền tảng cơ bản cho phép phát triển giao diện điều khiển, các ứng dụng mạng và các ứng dụng dạng Win Form..

+ Java EE (Java Platform, Enterprise Edition): Được xây dựng trên nền tảng Java SE, giúp phát triển các ứng dụng web, các ứng dụng ở cấp doanh nghiệp, ...

+ Java ME (Java Platform, Micro Edition): Là một nền tảng cho phép phát triển các ứng dụng nhúng vào các thiết bị điện tử như mobile,...

- Đặc trưng Java:

+ Ngôn ngữ lập trình hướng đối tượng; độc lập với mọi nền tảng phần cứng, mọi hệ điều hành; hỗ trợ phát triển ứng dụng chạy trên mạng.

+ Ngôn ngữ lập trình mạnh: tất cả dữ liệu đều phải được khai báo một cách tường minh; việc kiểm tra mã lệnh (code) được thực hiện tại thời điểm biên dịch và thông dịch; hạn chế các lỗi của chương trình.

+ Ngôn ngữ lập trình có tính bảo mật cao: xây dựng môi trường bảo mật cho việc thực thi chương trình; có nhiều mức khác nhau cho việc kiểm soát bảo mật.

- + Ngôn ngữ lập trình đa tuyến: cho phép thực hiện nhiều nhiệm vụ đồng thời trong một ứng dụng.

2. Các ứng dụng Java và công cụ lập trình Java

a) *Ứng dụng Java*

Có 4 kiểu ứng dụng chính của Java:

- Standalone Application: Còn được biết đến là ứng dụng Desktop (Desktop Application) hoặc ứng dụng trên nền tảng hệ điều hành Windows (Window based Application). Để tạo ra ứng dụng kiểu này ta thường sử dụng AWT, Swing hoặc JavaFX framework.
- Web Application: Là ứng dụng chạy trên server và tạo được các trang động. Hiện nay, servlet, jsp, struts, jsf, spring... là những công nghệ được sử dụng để tạo Web Application trong java.
- Enterprise Application: Là ứng dụng dành cho doanh nghiệp, ví dụ như các ứng dụng cho ngân hàng (Banking Application) với lợi thế là tính bảo mật cao, cân bằng tải (load balancing) và phân cụm (clustering). Trong java, EJB được sử dụng để tạo các Enterprise Application.
- Mobile Application: Là ứng dụng được tạo ra cho các thiết bị di động. Hiện nay Android và Java ME được sử dụng để tạo các ứng dụng này.

b) *Công cụ lập trình Java*

Là môi trường phát triển tích hợp IDE (Integrated Development Environment) hay là một chương trình để viết các mã (code). Chương trình này hỗ trợ nhiều tính năng tự động hóa cho người phát triển, như gợi ý khi lập trình, tự hoàn thiện mã,... Một số IDE thông dụng hiện nay cho Java gồm:

- NetBean: Được đánh giá là môi trường phát triển tích hợp mã nguồn mở miễn phí mạnh nhất cho Java. Nó được các nhà phát triển phần mềm chuyên nghiệp sử dụng để xây dựng các ứng dụng doanh nghiệp, web, di động và desktop.

NetBeans là đa nền tảng IDE và được sử dụng để hỗ trợ trên Linux, Windows, Mac cũng như trên Oracle Solaris. Ngoài ra, NetBeans là một môi trường được phát triển nhằm mục đích hỗ trợ phân tích, thiết kế, mã hóa, lập hồ sơ, thử nghiệm, biên soạn, gỡ lỗi, chạy và triển khai các ứng dụng.

Trang web tải NetBean: https://netbeans.org/index_vi.html.

- IntelliJ IDEA: Được đánh giá là một IDE có đầy đủ tất cả các tính năng cho các nhà phát triển Java EE. IntelliJ IDEA có hai phiên bản miễn phí và bản “Ultimate”. Phiên bản miễn phí tích hợp nhiều tính năng để xây dựng ứng dụng Android cũng như các ứng dụng JVM. Phiên bản Ultimate có các tính năng bổ sung như hỗ trợ cho các ngôn ngữ phát triển web (JavaScript, Coffeescript,...), hỗ trợ các framework (Node.js, Angular và React), hỗ trợ Java EE (JSF, JAX-RS, CDI, JPA,...), kiểm soát phiên bản với Team Foundation Server, Perforce, Clearcase và Visual SourceSafe.

- Trang web tải IntelliJ IDEA:

<https://www.jetbrains.com/idea/download/>.

- Eclipse IDE: Có hệ sinh thái riêng với một cộng đồng lớn của các nhà phát triển, với nhiều tài liệu hay và nhiều plugin để phát triển Java một cách tốt nhất. Các lập trình viên sử dụng Eclipse để phát triển ứng dụng Java cho di động, máy tính để bàn, web, doanh nghiệp cũng như các ứng dụng hệ thống nhúng.

Trang web tải Eclipse:

<https://www.eclipse.org/downloads/packages/eclipse-ide-javadevelopers/keplersr1>

Ngoài các IDE nêu trên, còn nhiều công cụ hỗ trợ lập trình Java khác như: Java Studio Enterprise, Sun Java Studio Creator, Borland Jbuilder, Jdeveloper, Jcreator,...

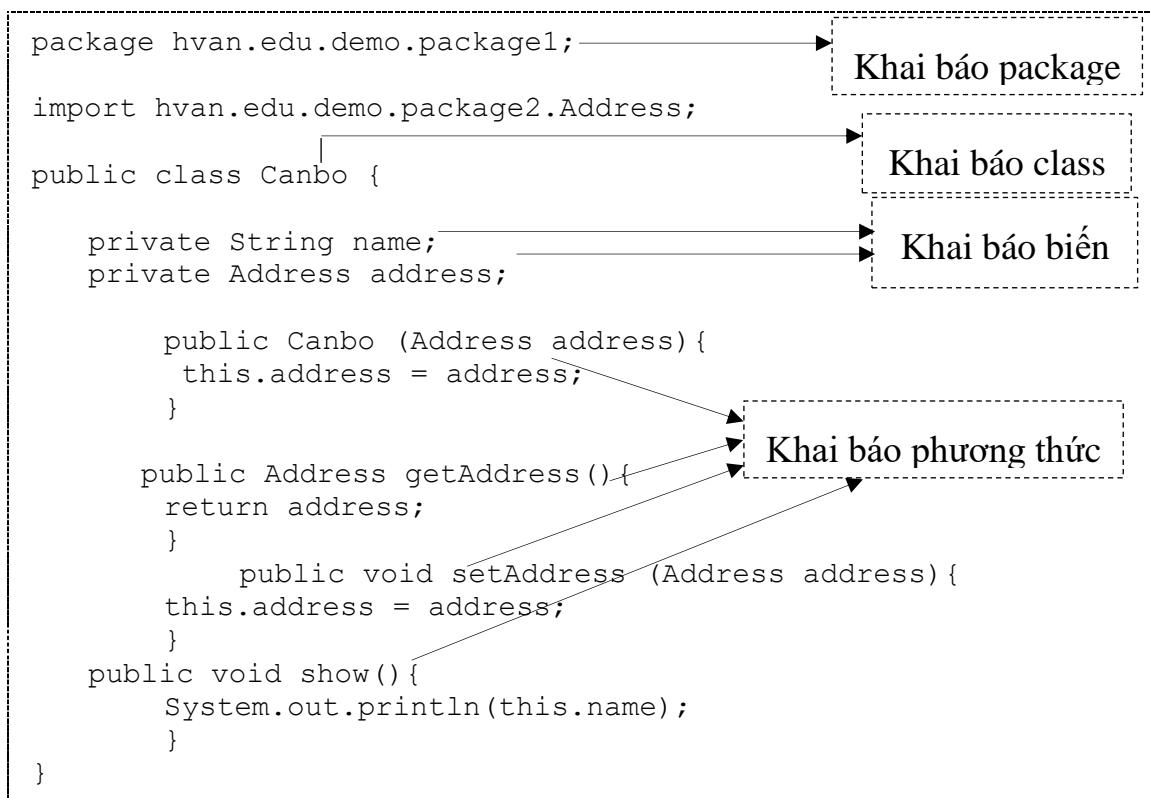
3. Cấu trúc một chương trình Java

Một chương trình Java về cơ bản bao gồm các thành phần sau:

- Khai báo gói (package).
- Khai báo lớp (class).
- Khai báo phương thức (method).
- Khai báo biến.
- Các câu lệnh.

Để hiểu rõ hơn các thành phần cơ bản trên, ta phân tích một ví dụ cụ thể sau:

Ví dụ 1.1



- Khai báo *package*:

Định nghĩa không gian tên, là nơi lưu trữ các lớp. *Package* được sử dụng để tổ chức các lớp dựa trên chức năng. Nếu không thực hiện khai báo *package*, các tên lớp sẽ được đặt vào gói mặc định là không có tên. Khai báo *package* phải được đặt ở dòng đầu tiên trong chương trình.

Trong ví dụ trên, khai báo package là: package hvan.edu.demo.package1 với tên package là hvan.edu.demo.package1.

- Khai báo lớp:

Trong ví dụ trên phần khai báo lớp là public class Canbo, với tên lớp là Canbo. Sau phần khai báo lớp là cặp dấu {} với các biến, phương thức sẽ được khai báo trong cặp dấu này. Trong khai báo lớp có sử dụng modifier - là các từ dùng trước các khai báo lớp, phương thức, biến, để quy định khả năng truy cập các thành viên của lớp đó ở các lớp khác. Khi nói đến khả năng truy cập (access) ta cần phân biệt 2 trường hợp:

+ Trường hợp 1: khi một phương thức của một lớp có thể truy cập biến hoặc phương thức của lớp khác.

Ví dụ 1.2

```
public class Person {  
    public void hello() {  
        System.out.println("hello");  
    }  
}  
  
public class Student {  
    public void demo() {  
        Person person = new Person();  
        person.hello();  
    }  
}
```

Trong Ví dụ 1.2, phương thức hello() của lớp Person là public nên tại lớp Student ta có thể truy cập nó từ lớp Person (person.hello()).

+ Trường hợp 2: khi một lớp con có thể thừa kế các biến/phương thức của lớp cha.

Ví dụ 1.3:

```
public class Person{  
    public void hello(){  
        System.out.println("hello");  
    }  
}
```

```

public class Student extends Person{
    public void demo(){
        this.hello();
        Person person = new Person();
        person.hello();
    }
}

```

Trong Ví dụ 1.3: Ví dụ 1.3, phương thức `hello` của lớp `Person` là `public` nên các lớp con của nó có thể kế thừa (`this.hello()`) và phương thức `hello()` của lớp `person` là `public` nên ta có thể truy cập nó từ lớp `Person` (`person.hello()`).

Với lớp ta có 2 loại modifier là `public` và `default` nhưng với biến và phương thức thì ta có 4 modifier là `public`, `protected`, `default`, `private`.

+ Khai báo `default` với lớp: Một lớp với `default access` (không sử dụng các từ khóa `public`) sẽ chỉ được nhìn thấy và truy cập bởi các lớp khác trong cùng 1 package.

Ví dụ 1.4: lớp `Address` thuộc package `hvan.edu.demo.package2` được khai báo là `default access`:

```

package hvan.edu.demo.package2;
class Address {
}

```

Nếu bên trong lớp `Canbo` thuộc package `hvan.com.demo.package1` ta tạo thẻ hiện của lớp `Address` như sau:

```

package hvan.com.demo.package1;
import hvan.com.demo.package2.Address;

class Canbo {
    private Address address;
}

```

Khi đó chương trình sẽ báo lỗi:

```

3  Address is not public in hvan.com.demo.package2; cannot be accessed from outside package
4  ---
5  (Alt-Enter shows hints)
6
7  import hvan.com.demo.package2.Address;
8
9  class Canbo {
10    private Address address;
11
12 }
13

```

+ Khai báo public với lớp: Một lớp được khai báo với từ khóa public sẽ cho phép tất cả các lớp từ tất cả các package truy cập. Lưu ý nếu truy cập public lớp ở một lớp thuộc package khác ta vẫn phải khai báo import package cho lớp đó. Như ở ví dụ trên ta có khai báo: `import hvan.edu.demo.package2.Address;`

- Khai báo biến và phương thức: Trong ví dụ trên ta có các khai báo biến:

```

private String name;
private Address address;

```

và khai báo phương thức:

```

public Canbo (Address address) {
    this.address = address;
}

public Address getAddress () {
    return address;
}

public void setAddress (Address address) {
    this.address = address;
}

public void show() {
    System.out.println(this.name);
}

```

+ Khai báo public với biến và phương thức: Khi một phương thức hoặc biến được khai báo là public, có nghĩa là tất cả các lớp khác, kể cả các lớp không thuộc cùng package đều có thể truy cập.

Ví dụ 1.5:

```

package hvan.com.demo.package1;
public class A{
    public String str = "abc";
}

```

```

        public void hello() {
            System.out.println("hello");
        }
    }
package hvan.com.demo.package2;
import hvan.com.demo.package1.A;

public class B{
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.str);
        a.hello();
    }
}

```

Trong Ví dụ 1.5:, biến str là public nên ở package hvan.com.demo.package2 ta có thể truy cập biến này từ lớp khác package (lớp A thuộc package hvan.com.demo.package1):

```

A a = new A();
System.out.println(a.str);

```

và phương thức hello là public nên có thể truy cập từ lớp khác package:
a.hello();

+ Khai báo private với biến và phương thức: Khi một phương thức hoặc biến được khai báo là private nó sẽ không thể truy cập từ lớp khác, kể cả các lớp cùng file nguồn hay các lớp con. Tuy nhiên lớp bên trong một lớp khác (Inner class) có thể truy cập được thành phần private của lớp chứa nó.

Ví dụ 1.6:

```

package hvan.com.demo.package1;

public class A {
    private String str = "abc";
    class B {
        public void demo() {
            A a = new A();
            System.out.println(a.str);
        }
    }
    class C {
        public void demo() {
            A a = new A();
            System.out.println(a.str); // Lỗi biến dịch
        }
    }
}

```

```
    }  
}
```

+ Khai báo default với biến và phương thức: Khi một phương thức hoặc biến được khai báo là default thì chỉ có các lớp thuộc cùng package với nó mới có thể truy cập.

Ví dụ 1.7:

```
package hvan.com.demo.package1;  
  
public class A {  
    // default access  
    String str = "abc";  
}  
  
package hvan.com.demo.package2; // không cùng package với class A  
import hvan.com.demo.package1.A;  
  
public class B {  
    public void demo() {  
        A a = new A();  
        System.out.println(a.str); // Lỗi biên dịch  
    }  
}
```

+ Khai báo protected với biến và phương thức: protected modifier gần giống với default modifier, nó hạn chế khả năng truy cập trong cùng 1 package, tuy nhiên với protected modifier thì nó cho phép truy cập từ các lớp con kế cả khi lớp con không nằm cùng package với lớp cha (truy cập theo trường hợp thừa kế).

Ví dụ 1.8:

```
package hvan.com.demo.package1;  
public class A {  
    protected String str = "abc";  
}  
  
package hvan.com.demo.package2; // không cùng package với class A  
import hvan.com.demo.package1.A;  
  
public class B extends A{  
    public void demo() {  
        System.out.println(this.str); // compile success  
        A a = new A();  
        System.out.println(a.str); // compile error  
    }  
}
```

```
}
```

+ Khai báo phương thức kiểu void: cho phép tạo các phương thức mà không trả về giá trị nào.

+ Khai báo phương thức có tên là “main”: public static void main(String[] args) với đối số (args) là mảng chuỗi ký tự (String[]). Đây được xem là điểm bắt đầu của một ứng dụng Java. Hầu hết ứng dụng Java đều phải có phương thức này.

+ Phương thức khởi tạo (constructor): Là một phương thức đặc biệt, nó được dùng để khởi tạo và trả về đối tượng của lớp mà nó được định nghĩa. Phương thức khởi tạo sẽ có tên trùng với tên của lớp mà nó được định nghĩa và chúng không được định nghĩa một kiểu giá trị trả về. Mỗi lớp có ít nhất một phương thức khởi tạo. Nếu ta không tạo một phương thức khởi tạo rõ ràng cho một lớp thì bộ biên dịch Java sẽ tạo một phương thức khởi tạo mặc định cho lớp đó.

Ví dụ 1.9:

```
public class LopHoc{  
    public Lophoc () {  
    }  
  
    public Lophoc (String ten){  
        // Contructor nay co mot tham so la ten.  
    }  
}
```

Có hai loại phương thức khởi tạo trong Java là phương thức khởi tạo mặc định không có tham số và phương thức khởi tạo được tham số hóa.

* Cú pháp khai báo phương thức khởi tạo mặc định: ten_lop() {}

Ví dụ 1.10: Phương thức khởi tạo mặc định

```
class Giaovien {  
    Giaovien (){  
        System.out.println("Giaovien duoc tao");  
    }  
  
    public static void main(String args[]){  
        Giaovien gv=new Giaovien ();  
    }  
}
```

```
}
```

Trong ví dụ trên, ta tạo phương thức khởi tạo không có tham số trong lớp Giaovien. Nó sẽ được triệu hồi tại thời điểm tạo đối tượng.

Mục đích sử dụng phương thức khởi tạo mặc định là để cung cấp các giá trị mặc định cho đối tượng như 0, null,... tùy thuộc vào kiểu dữ liệu.

Ví dụ 1.11: Trường hợp không tạo phương thức khởi tạo cho lớp thì trình biên dịch sẽ tự tạo phương thức khởi tạo mặc định cho lớp đó:

```
class Student1{
    int id;
    String name;

    void display(){
        System.out.println(id+" "+name);
    }

    public static void main(String args[]){
        Student3 s1=new Student3();
        Student3 s2=new Student3();
        s1.display();
        s2.display();
    }
}
```

* Phương thức khởi tạo được tham số hóa dùng để cung cấp các giá trị khác nhau cho các đối tượng riêng biệt.

Ví dụ 1.12:

```
class Student2{
    int id;
    String name;

    Student2(int i,String n){
        id = i;
        name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student4 s1 = new Student4(111,"Hong");
        Student4 s2 = new Student4(222,"Thu");
        s1.display();
        s2.display();
    }
}
```

Trong Ví dụ 1.12:, ta tạo phương thức khởi tạo của lớp Student2 có tham số.

Bảng 1.1. So sánh phương thức khởi tạo và phương thức thông thường.mô tả những điểm khác nhau giữa phương thức khởi tạo và phương thức thông thường.

Phương thức khởi tạo	Phương thức thông thường
Được sử dụng để khởi tạo trạng thái của một đối tượng.	Được sử dụng để biểu diễn hành vi của một đối tượng.
Không có kiểu trả về	Phải có kiểu trả về.
Được triệu hồi một cách ngầm định.	Phải được triệu hồi một cách tường minh.
Trình biên dịch cung cấp một constructor mặc định nếu ta không tạo bất kỳ constructor nào trong lớp.	Không được cung cấp bởi trình biên dịch trong bất cứ trường hợp nào.
Tên constructor phải giống tên lớp.	Tên phương thức có thể hoặc không giống như tên lớp.

Bảng 1.1. So sánh phương thức khởi tạo và phương thức thông thường.

- Các câu lệnh

+ Câu lệnh trong Java là các dòng chỉ thị yêu cầu thực hiện một việc gì đó, mỗi câu lệnh được kết thúc bởi một dấu chấm phẩy (';'). Trong ví dụ trên có các câu lệnh như:

```
import hvan.edu.demo.package2.Address;  
  
private String name;  
  
this.address = address;  
  
return address;  
  
this.address = address;  
  
System.out.println(this.name);
```

Câu lệnh *import* trong Java giúp ta có thể sử dụng tên của các class khác trong class của mình mà không cần phải khai báo thêm tên package chứa class đó.

Trong ví dụ trên, ta có khai báo:

```
import hvan.edu.demo.package2.Address;
```

Lúc này lớp Address và lớp Canbo nằm trong hai package khác nhau, khi lớp Canbo sử dụng lớp Address, để không phải khai báo tên package ta đã sử dụng câu lệnh import trên.

Ta cũng có thể không sử dụng câu lệnh import, tuy nhiên trong trường hợp này phải khai báo cả tên package. Ví dụ với khai báo có câu lệnh import như trên, ta có khai báo biến trong lớp Canbo là:

```
private Address address;
```

Tuy nhiên, nếu không có câu lệnh khai báo import trên thì câu khai báo biến sẽ là:

```
private hvan.edu.demo.package2.Address address
```

- Lưu ý:

- + Trong trường hợp sử dụng 2 lớp có cùng tên nằm trong 2 package khác nhau thì ta không sử dụng câu lệnh import với chúng được. Lúc này phải khai báo tên package cho mỗi class.

Ví dụ 1.13:

```
public class Example {  
    private com.huongdancoban.Student student;  
    private com.huongdancoban.java.Student student1;  
}
```

Trong ví dụ trên, có 2 lớp cùng tên Student được sử dụng, một lớp thuộc package com.huongdancoban, còn package còn lại là com.huongdancobanJava. Do đó bắt buộc phải khai báo tên package cho cả hai lớp mà không thể khai báo chúng như sau:

```
import com.huongdancoban.Student;
import com.huongdancoban.java.Student;
public class Example {
    private Student student;
    private Student student1;
}
```

Tuy nhiên, ta cũng có thể sử dụng câu lệnh import cho một lớp, lớp còn lại bắt buộc phải khai báo thêm tên package.

Ví dụ 1.14:

```
import com.huongdancoban.Student;
public class Example {
    private Student student;
    private com.huongdancoban.java.Student student1;
}
```

Trong ví dụ này, lớp `Student` trong package `com.huongdancoban` đã sử dụng câu lệnh import nên ta không cần phải khai báo thêm tên package nữa. Còn lớp `Student` trong package `com.huongdancoban.java` bắt buộc phải khai báo thêm tên package.

+ Nếu ta sử dụng nhiều lớp nằm trong cùng một package thì ta có thể sử dụng một câu lệnh import cho tất cả như sau:

```
import com.huongdancoban.*;
public class Example {
    private Student student;
    private School school;
}
```

Ở ví dụ này, lớp `Student` và lớp `School` nằm trong cùng một package có tên là `com.huongdancoban`. Do đó, để sử dụng hai lớp này chúng ta có thể khai báo tất cả các lớp nằm trong package này bằng câu lệnh `import com.huongdancoban.*;`

+ Nếu ta muốn sử dụng các biến hay phương thức static của lớp khác trong lớp của mình, ta có thể sử dụng câu lệnh `import static`. Ví dụ:

```
package com.huongdancoban;
public class Student {
    public static final String NAME = "Huyen";
    public static void goToSchool() {
    }
```

```
}
```

Ở đây ta định nghĩa cho lớp Student một biến static NAME và một phương thức static là goToSchool(). Để sử dụng những biến và phương thức này mà không cần phải khai báo thêm tên package và tên lớp, ta có thể khai báo như sau trong lớp Example:

```
import static com.huongdancoban.Student.NAME;
import static com.huongdancoban.Student.goToSchool;
public class Example {
    public static void main(String[] args) {
        System.out.println(NAME);
        goToSchool();
    }
}
```

Chú ý không thể hoán đổi vị trí của hai từ import và static trong câu lệnh import static.

+ Khối lệnh là một nhóm các câu lệnh được đặt trong cặp dấu ngoặc nhọn {}.

Ví dụ 1.15:

```
class Block {
    public static void main(String[] args) {
        boolean pass = true;
        if(pass == true) {
            System.out.println("Condition is true");
            System.out.println("You are passed");
        } else {
            System.out.println("Condition is false");
            System.out.println("You are not passed");
        }
    }
}
```

Trong Java có 2 loại khối lệnh là static block và instance block. Static block là một khối gồm nhiều dòng code, thuộc về một lớp cụ thể nào đó chứ không thuộc về instance của một lớp. Static block được định nghĩa với từ khóa static.

Ví dụ 1.16:

```
package HuongDanCoBan;
public class StaticBlock {
    static {
        System.out.print("Đây là static block");
    }
}
```

```
}
```

Instance block cũng là một khối gồm nhiều dòng code, nhưng nó không thuộc về lớp mà thuộc về instance của lớp đó. Nó được thực thi mỗi khi instance của lớp được khởi tạo. Ví dụ:

```
package HuongDanCoBan;

public class InstanceBlock {
    System.out.print("Đây là instance block");
}
```

4. Các phần tử cơ sở

a) Định danh

Trong Java, một định danh có thể là tên một lớp, tên một phương thức, tên một biến. Trong quá trình viết chương trình, nếu các định danh không hợp lệ thì lỗi sẽ xuất hiện ngay tại lúc biên dịch. Thông thường các IDE (Eclipse, Netbeans...) hiện nay đều hỗ trợ báo lỗi ngay khi có định danh không hợp lệ.

Các nguyên tắc, qui định bắt buộc về đặt tên, định danh như sau:

- Chỉ bao gồm các ký tự là chữ số hoặc chữ cái ([A-Z],[a-z],[0-9]), ký tự '\$' và ký tự '_'
- Định danh không được bắt đầu bằng chữ số.
- Định danh có phân biệt hoa thường. Ví dụ int tongso và int Tongso là hai định danh khác nhau.
- Chiều dài của định danh không bị giới hạn tuy nhiên chỉ nên dùng các định danh có chiều dài 4 – 15 ký tự.
- Không được sử dụng các từ khóa trong Java để làm định danh (ví dụ: if, else, true, false...)

Ví dụ các từ khóa hợp lệ: int tongso; int \$tongso; int _tongso; int Tongso; int tongso1; int _\$tongso; int tong_so;

Ví dụ các từ khóa không hợp lệ: int tongso@; int 1age; int if; int tongso#;

- Đặt tên Package: Package thường được đặt tên giống như đặt tên thư mục trên ổ đĩa tức là sẽ được bắt đầu bằng tên có phạm vi lớn cho đến phạm vi nhỏ dần (giống như thư mục cha có chứa các thư mục con). Các ký tự trong định danh package là chữ, số in thường. Thông thường ta sẽ đặt tên package với các phạm vi và thứ tự: Tên tổ chức, tên miền → Tên đơn vị → Tên dự án → Tên module → Tên chức năng module.

Ví dụ với tên miền **hvan.vn**; tên project là **Demo**; project có hai module là **demo1** và **demo2**. Trong module **demo1** có 1 lớp là **Demo.java**. Khi đó ta sẽ khai báo package trong lớp Demo.java là package hvan.vn.demo.demo1

- Đặt tên lớp: Tên lớp nên là danh từ, tránh dùng tên trùng với các kiểu dữ liệu đã được định nghĩa sẵn (ví dụ như Number, String, Float...). Tên lớp nên đặt theo quy tắc lắc đà (camelCase), tức là chữ cái đầu tiên của mỗi từ viết hoa, các chữ cái tiếp theo viết thường (ví dụ: DemoJava.java, DemoHelloWorld.java...).

- Đặt tên phương thức: Tên các phương thức thường là động từ, và được đặt theo kiểu camelCase nhưng chữ cái đầu tiên viết thường. Ví dụ: setAge, isTurnOn, getAge...

- Đặt tên biến: Tên được đặt theo kiểu camelCase nhưng chữ cái đầu tiên viết thường. Nên dùng các từ dễ nhớ, tránh dùng các định danh chỉ gồm 1 ký tự. Các định danh gồm 1 ký tự thường chỉ dùng làm biến tạm, ví dụ i,j,k dùng làm biến tạm cho kiểu số; c,d,e thường dùng làm biến tạm có kiểu ký tự. Ví dụ: dateOfBirth, age...

- Đặt tên hằng số: Các hằng số được đi kèm với các từ khóa static, final khi khai báo và thường là danh từ. Tên hằng số nên dùng tất cả các chữ cái viết hoa và phân cách nhau bằng dấu gạch dưới. Ví dụ: MIN_HEIGHT, MAX_WIDTH

Lưu ý, có một cách đặt tên khác không dùng kiểu camelCase mà dùng kiểu snake_case, tức là tất cả các chữ cái viết thường và phân cách nhau bởi dấu gạch

dưới, ví dụ date_of_birth. Cách viết này dễ đọc hơn nhưng dài hơn và thường chỉ dùng hiển thị kết quả trả về, đặt tên trong java script...

b) Các từ khóa

Ngôn ngữ lập trình Java có khoảng 50 từ khóa, chia thành 9 nhóm, với các từ khóa thường được sử dụng ở từng nhóm như sau:

- Tổ chức các lớp:

+ package: Xác định một gói sẽ chứa một số lớp ở trong file mã 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.

- Định nghĩa các lớp:

+ interface: Được sử dụng để định nghĩa interface

+ class: Được sử dụng để định nghĩa lớp

+ extends: Được sử dụng để định nghĩa lớp con kế thừa các thuộc tính và phương thức từ lớp cha.

+ implements: Xây dựng một lớp mới cài đặt những phương thức từ interface xác định trước.

- Các từ khóa cho các biến và các lớp:

+ abstract: Khai báo lớp, phương thức, interface trừu tượng không có thể hiện (instance) cụ thể.

+ public: Khai báo lớp, biến dữ liệu, phương thức công khai có thể truy cập ở mọi nơi trong hệ thống.

+ private: Khai báo biến dữ liệu, phương thức riêng trong từng lớp và chỉ cho phép truy cập trong lớp đó.

+ protected: Khai báo biến dữ liệu, phương thức chỉ được truy cập ở lớp cha và các lớp con của lớp đó.

+ static: Định nghĩa biến, phương thức của một lớp có thể được truy cập trực tiếp từ lớp mà không thông qua khởi tạo đối tượng của lớp.

+ synchronized: Chỉ ra việc 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, thường được sử dụng trong lập trình đa luồng (multithreading).

+ volatile: Báo cho chương trình dịch biết là biến khai báo volatile có thể thay đổi tùy ý trong các luồng (thread).

+ final: Chỉ ra các biến, phương thức không được thay đổi sau khi đã được định nghĩa. Các phương thức final không thể được kế thừa và ghi đè.

+ native: Giúp lập trình viên có thể sử dụng mã chương trình được viết bằng các ngôn ngữ khác.

- 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 với các giá trị biểu diễn theo dạng dấu phẩy động 64 bit (8 byte).

+ Boolean: Khai báo biến kiểu logic với 2 trị: true, false.

+ void: Chỉ định một phương thức không trả về giá trị.

- Những từ khóa cho các giá trị và các biến:

+ false: Kiểu logic với giá trị sai.

- + true: Kiểu logic với giá trị đúng.
- + this: Biến chỉ tới đối tượng hiện thời.
- + super: Biến chỉ tới đối tượng ở lớp cha.
- + null: Được sử dụng như một giá trị đặc biệt để biểu thị trạng thái chưa khởi tạo, điều kiện chấm dứt, đối tượng không tồn tại, hay giá trị không xác định.
- Xử lý ngoại lệ:
 - + throw: Tạo một đối tượng ngoại lệ để chỉ định một trường hợp ngoại lệ xảy ra.
 - + throws: Chỉ định cho qua ngoại lệ khi ngoại lệ xảy ra.
 - + try: Thủ thực hiện cho đến khi gặp một ngoại lệ.
 - + catch: Được sử dụng để bắt ngoại lệ, được sử dụng cùng với try để xử lý các ngoại lệ xảy ra trong chương trình.
 - + finally: Thực hiện một khối lệnh đến cùng bất chấp các ngoại lệ có thể xảy ra. Được sử dụng trong try-catch.
- Tạo lập và kiểm tra các đối tượng:
 - + new: Khởi tạo đối tượng.
 - + instanceof: Kiểm tra xem một đối tượng nào đó có phải là một thể hiện của một lớp được định nghĩa trước hay không.
- Dòng điều khiển:
 - + switch: Sử dụng trong câu lệnh điều khiển switch case.
 - + case: Trường hợp được tuyển chọn theo switch (chỉ được dùng khi đi kèm switch).
 - + default: Mặc định được thực thi khi không có case nào trả về giá trị true (dùng trong switch case).
 - + break: Thoát ra khỏi vòng lặp hoặc lệnh switch-case.

- + if: Lệnh chọn theo điều kiện logic.
- + else: Rẽ nhánh theo điều kiện ngược lại của if.
- + continue: Dừng chu trình lặp hiện tại và bắt đầu chu trình tiếp theo.
- + return: Kết thúc phương thức và trả về giá trị cho phương thức.
- + do: Dùng trong vòng lặp do while.
- + while: Được sử dụng trong lệnh điều khiển while.
- + for: Sử dụng cho vòng lặp for với bước lặp được xác định trước.
- Những từ khóa chưa được sử dụng: byvalue future outer; const generic rest; goto inner var; cast operator.

c) *Chú thích*

Trong chương trình Java, chú thích là dòng, đoạn ghi chú lại mục đích, ý nghĩa của đoạn mã chương trình được viết ra để giúp cho việc đọc lại nó dễ hiểu hơn. Một dự án tốt là dự án có chú thích đầy đủ cho những hàm được viết ra. Khi thực thi chương trình, các dòng, đoạn chú thích sẽ bị bỏ qua, chương trình chỉ thực hiện những câu lệnh mà không thực hiện những gì viết trong chú thích. Có 3 kiểu chú thích trong chương trình Java:

- Chú thích trên một dòng:

Cú pháp:

```
//<nội dung cần chú thích>
```

Ví dụ 1.17:

```
public static void main(String[] args) {
    //khởi tạo và truyền giá trị cho biến course
    String course = "Java";

    //khởi tạo và truyền giá trị cho biến lecture
    String lecture = " Java cơ bản";
    test t = new test();

    //Hàm Xuatthongtin dùng để in thông tin về khóa học
```

```

        t.Xuatthongtin(course, lecture);
    }

public void Xuatthongtin(String course, String lecture){
    System.out.println("Course: "+course);
    System.out.println("Lecture: "+lecture);
}

```

Trong Ví dụ 1.17: chú thích trên một dòng có thể được đặt sau dòng mã chương trình hoặc đặt trước nó.

- Chú thích trên nhiều dòng

Cú pháp:

```

/*
<nội dung ghi chú dòng 1>
<nội dung ghi chú dòng 2>
.....
*/

```

Ví dụ 1.18:

```

public static void main(String[] args) {
    /*
    Khởi tạo 2 biến course và lecture
    Truyền giá trị vào cho 2 biến đó để xuất thông tin
    */
    String course = "Java";
    String lecture = "LapTrinh";

    /*
    Sau khi khởi tạo giá trị, khởi tạo hàm test để có thể gọi
    hàm Xuatthongtin
    Sau đó chúng ta truyền 2 biến vừa khởi tạo là hoàn thành
    việc in thông tin
    */

    test t = new test();
    t.Xuatthongtin(course, lecture);
}

public void Xuatthongtin(String course, String lecture) {
    System.out.println("Course: "+course);
    System.out.println("Lecture: "+lecture);
}

```

- Chú thích sử dụng Javadoc: Sử dụng công cụ Javadoc để hỗ trợ ghi chú.

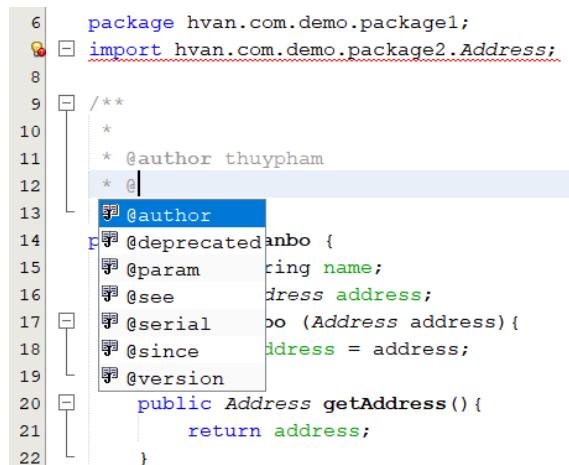
Cú pháp:

```

/**
 * <nội dung ghi chú>
 * @param args
 */

```

Điểm khác biệt của cách ghi chú này là nó hỗ trợ các Tags của Javadoc. Cụ thể khi ta gõ chữ @ trong phần viết ghi chú thì nó sẽ hiện ra các tag như hình sau:



Hình 0.1. Các Tags trong Javadoc.

Các tag này giúp giải thích rõ ràng về tham số, giá trị trả về... Ví dụ một số tag:

```

@Author: chú thích tên tác giả của ứng dụng.
@since: chú thích thời gian hoàn thành ứng dụng.
@return: chú thích nội dung trả về của phương thức trong ứng dụng.
@param: chú thích cụ thể về các tham số trong ứng dụng.
@exception: chú thích về các ngoại lệ sử dụng.

```

```

/*
 * To change this license header, choose License Headers in
Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package hvan.com.demo.package1;
/**
 *
 * @author thuypham
 */
public class Demo {
    /**
     * @param args là các đối số dòng lệnh
    */
    public static void main(String[] args) {

```

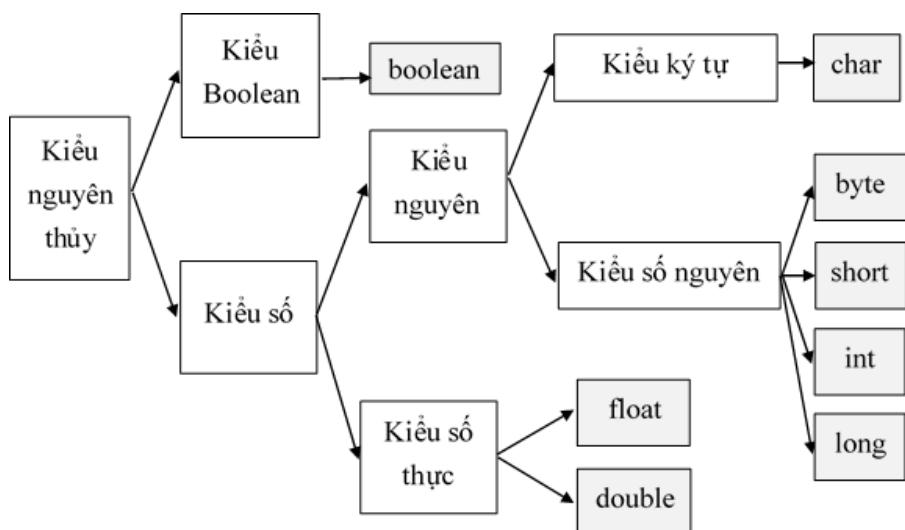
```
}
```

5. Các kiểu dữ liệu, biến và hằng

a) Các kiểu dữ liệu

Các ứng dụng luôn xử lý dữ liệu ở đầu vào và xuất dữ liệu kết quả ở đầu ra. Đầu vào, đầu ra và kết quả của các quá trình tính toán đều liên quan đến dữ liệu. Trong môi trường tính toán, dữ liệu được phân lớp theo các tiêu chí khác nhau phụ thuộc vào bản chất của nó. Ở mỗi tiêu chí, dữ liệu có một tính chất xác định và có một kiểu thể hiện riêng biệt. Kiểu dữ liệu trong Java dùng để xác định kích thước và loại giá trị có thể được lưu trữ trong một định danh (Định danh ở đây bao gồm tên biến, phương thức, tên lớp, interface và tên package). Các dữ liệu khác nhau cho phép chúng ta lựa chọn kiểu phù hợp với yêu cầu của bài toán đặt ra. Trong Java kiểu dữ liệu được chia thành hai loại: Các kiểu dữ liệu nguyên thủy (primitive) và các kiểu dữ liệu tham chiếu (reference).

- Các kiểu dữ liệu nguyên thủy (**Error! Reference source not found.**) bao gồm kiểu logic và kiểu số, trong kiểu số lại gồm kiểu nguyên và kiểu số thực. Trong kiểu nguyên có kiểu ký tự và kiểu số nguyên, gồm 4 kiểu là byte, short, int, long. Trong kiểu số thực gồm 2 loại là float và double.



Hình 1. 5. Các kiểu dữ liệu nguyên thủy.

+ Kiểu byte: Dùng để lưu dữ liệu kiểu số nguyên có kích thước một byte (8 bit). Giá trị nhỏ nhất là -128 (-2^7), giá trị lớn nhất là 127 ($2^7 - 1$) và giá trị mặc định là 0. Kiểu dữ liệu byte được sử dụng để lưu giữ khoảng trống trong các mảng lớn, chủ yếu là các số nguyên.

Ví dụ: byte a = 100, byte b = -50

+ Kiểu short: Dùng để lưu dữ liệu kiểu số nguyên kích thước 2 byte (16 bit). Giá trị nhỏ nhất là -32,768 (-2^{15}), giá trị lớn nhất là 32,767 ($2^{15} - 1$) và giá trị mặc định là 0. Kiểu dữ liệu short cũng có thể được sử dụng để lưu bộ nhớ như kiểu dữ liệu byte.

Ví dụ: short s = 10000, short r = -20000

+ Kiểu int: Dùng để lưu dữ liệu kiểu số nguyên kích cỡ 4 byte (32 bit). Giá trị nhỏ nhất là -2,147,483,648 (-2^{31}), giá trị lớn nhất là 2,147,483,647 ($2^{31} - 1$) và giá trị mặc định là 0. int được sử dụng như là kiểu dữ liệu mặc định cho các giá trị nguyên.

Ví dụ: int a = 100000, int b = -200000

+ Kiểu long: Dùng để biểu diễn các số nguyên có kích thước 8 byte. Giá trị nhỏ nhất là -9,223,372,036,854,775,808 (-2^{63}), giá trị lớn nhất là 9,223,372,036,854,775,807 ($2^{63} - 1$), và giá trị mặc định là 0L. Kiểu này được sử dụng khi cần một dải giá trị rộng hơn int.

Ví dụ: long a = 100000L, int b = -200000L

+ Kiểu float: Dùng để biểu diễn các số thực có kích thước 4 byte (32 bit). Kiểu Float được sử dụng chủ yếu để lưu bộ nhớ trong các mảng rộng hơn các số dấu chấm động. Giá trị nhỏ nhất: $-3.4028235 \times 10^{38}$, giá trị lớn nhất: 3.4028235×10^{38} , và giá trị mặc định là 0.0f.

Ví dụ: float f1 = 234.5f

+ Kiểu double: Kiểu dữ liệu double dùng để biểu diễn các số thực có kích thước 8 byte. Nói chung, kiểu dữ liệu này được sử dụng như là kiểu mặc định cho các giá trị thập phân. Giá trị nhỏ nhất: $-1.7976931348623157 \times 10^{308}$, giá trị lớn nhất: $1.7976931348623157 \times 10^{308}$, và giá trị mặc định là 0.0d.

Ví dụ: double d1 = 123.4

+ Kiểu boolean: Độ lớn 1 bit, dùng để lưu dữ liệu chỉ có hai trạng thái true hoặc false. Giá trị mặc định là false.

Ví dụ: boolean one = true

+ Kiểu char: Dùng để biểu diễn các số nguyên không âm có kích thước 2 byte (16 bit). Nó cũng được sử dụng để đại diện cho một ký tự Unicode, vì bản chất mỗi ký tự đều tương ứng với một số cụ thể (số này được hiểu như là mã của ký tự). Vì char là kiểu số nguyên không âm, kích thước 2 byte, nên phạm vi biểu diễn của nó là $[0, 2^{16}-1]$ ($[0, 65535]$). Khi char được hiểu như một ký tự Unicode thì ký tự nhỏ nhất là '\u0000' (mã 0), và ký tự lớn nhất là '\uffff' (mã 65535).

Ví dụ: char letterA ='A'

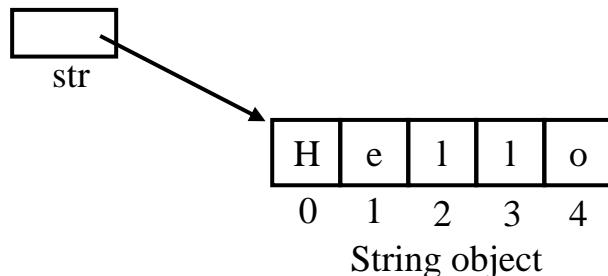
Error! Reference source not found. tóm tắt 8 kiểu dữ liệu nguyên thủy trong Java với các giá trị biểu diễn và kích thước tương ứng với từng kiểu.

Kiểu dữ liệu	Giá trị mặc định	Kích thước
boolean	false	1 bít
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Hình 1.6. 8 kiểu dữ liệu nguyên thủy trong Java.

- Các kiểu dữ liệu tham chiếu:

Trong Java một kiểu dữ liệu được tạo ra bởi sự kết hợp các kiểu nguyên thủy với nhau được gọi là kiểu tham chiếu. Kiểu tham chiếu thường được sử dụng nhất đó là String, nó là sự kết hợp của các ký tự.



Các kiểu dữ liệu tham chiếu được tạo ra dựa trên một lớp. Lớp giống như một bản thiết kế (blueprint) để định nghĩa một kiểu tham chiếu.

Ví dụ 1.19:

```
class Address {  
    String address;  
    String cityName;  
}  
  
class Student {  
    String fullName;  
    int age;  
    Address address;  
}
```

b) Biến

Trong java, biến là tên của vùng nhớ trong máy tính dùng để lưu trữ giá trị mà chương trình có thể tương tác. Có 2 cách khai báo biến trong Java:

- Cách 1: [kiểu_dữ_liệu] [tên_biến];

Ví dụ 1.20:

```
double d1;  
int n, m;
```

- Cách 2: [kiểu_dữ_liệu] [tên_biến] = [giá_trị];

Ví dụ 1.21:

```
double d1 = 100.04;  
int n = 10, m = 20;
```

Có 3 kiểu biến trong java, bao gồm biến cục bộ (local), biến thể hiện (instance) và biến tĩnh (static).

- Biến cục bộ: được khai báo trong các phương thức, phương thức khởi tạo hoặc trong khối lệnh (block). Các biến cục bộ được lưu trên vùng nhớ stack của bộ nhớ máy tính. Khi kết thúc các phương thức, phương thức khởi tạo và khối lệnh, biến cục bộ sẽ được giải phóng. Lưu ý không được sử dụng “access modifier” khi khai báo biến cục bộ và cần khởi tạo giá trị mặc định cho biến cục bộ trước khi sử dụng chúng.

Ví dụ 1.22:

```
package variable;  
  
public class LocalVariable {  
    public static void main(String[] args) {  
        int localVariable1 = 19;          // khai báo biến cục bộ  
        float localVariable2 = 5.4f;      // khai báo biến cục bộ  
        System.out.println("Giá trị của biến localVariable1 = " +  
localVariable1);  
        System.out.println("Giá trị của biến localVariable2 = " +  
localVariable2);  
    }  
}
```

- Biến thể hiện: được khai báo trong một lớp, bên ngoài các phương thức, constructor và block. Biến thể hiện được lưu trong bộ nhớ heap, được tạo khi một đối tượng được tạo bằng từ khóa “new” và biến sẽ được giải phóng khi đối tượng được giải phóng. Biến thể hiện có thể được sử dụng bởi các phương thức, constructor, block,... nhưng nó phải được sử dụng thông qua một đối tượng cụ thể. Được phép sử dụng “access modifier” khi khai báo biến thể hiện. Nếu khai báo biến thể hiện bên trong lớp ta có thể gọi nó trực tiếp bằng tên ở bên trong lớp đó.

Biến thể hiện có giá trị mặc định phụ thuộc vào kiểu dữ liệu của nó. Ví dụ nếu là kiểu int, short, byte thì giá trị mặc định là 0, kiểu double thì là 0.0d, ... Vì vậy, ta không cần khởi tạo giá trị cho biến thể hiện trước khi sử dụng.

Ví dụ 1.23:

```
package bienvadulieu1;

public class Sinhvien {
    // biến toàn cục "ten" kiểu String, có giá trị mặc định là null
    public String ten;

    // biến toàn cục "tuoi" kiểu Integer, có giá trị mặc định là 0
    private int tuoi;

    // sử dụng biến ten trong một constructor
    public Sinhvien(String ten) {
        this.ten = ten;
    }

    // sử dụng biến tuoi trong phương thức setTuoi
    public void setTuoi(int tuoi) {
        this.tuoi = tuoi;
    }

    public void showStudent() {
        System.out.println("Ten : " + ten);
        System.out.println("Tuoi : " + tuoi);
    }

    public static void main(String args[]) {
        Sinhvien sv = new Sinhvien("Nguyen Van A");
        sv.setTuoi(21);
        sv.showStudent();
    }
}
```

- Biến tĩnh:

Biến tĩnh được khai báo trong một lớp với từ khóa “static”, phía bên ngoài các phương thức, phương thức khởi tạo và khôi lệnh. Chỉ có duy nhất một bản sao của các biến tĩnh được tạo ra cho dù có bao nhiêu đối tượng từ lớp tương ứng được tạo ra đi nữa. Biến tĩnh được lưu trữ trong bộ nhớ static riêng. Biến tĩnh được tạo khi chương trình bắt đầu chạy và được giải phóng khi chương trình dừng. Giá trị mặc định của biến tĩnh phụ thuộc vào kiểu dữ liệu khai báo, tương tự như biến toàn cục.

Biến tĩnh được truy cập thông qua tên của lớp chứa nó, với cú pháp: TenClass.tenBien. Trong lớp, các phương thức sử dụng biến tĩnh bằng cách gọi tên của nó khi phương thức đó cũng được khai báo với từ khóa “static”.

Ví dụ 1.24:

```
package vn.viettuts.bienvadulieu2;

public class Sinhvien {
    // biến tĩnh 'ten'
    public static String ten = "Nguyen Van A";

    // biến tĩnh 'tuoi'
    public static int tuoi = 21;

    public static void main(String args[]) {
        // Sử dụng biến tĩnh bằng cách gọi trực tiếp
        System.out.println("Ten : " + ten);

        // Sử dụng biến tĩnh bằng cách gọi thông qua tên class
        System.out.println("Ten : " + Sinhvien.tuoi);
    }
}
```

c) Hằng

Hằng cũng tương tự như biến, nhưng đặc biệt ở chỗ nếu một biến được khai báo là hằng thì nó sẽ không thay đổi giá trị trong suốt chương trình. Hằng được khai báo như biến nhưng thêm từ khóa `final` vào trước khai báo.

Ví dụ 1.25:

```
final float PI = 3.14f;
final char FIRST_CHARACTER = 'a';
final int VIP_TYPE = 1;
```

Nên khai báo hằng bằng các ký tự viết hoa như ví dụ trên sẽ giúp chúng ta dễ dàng phân biệt được đâu là biến và đâu là hằng.

Hằng có thể là số nguyên: trường hợp giá trị hằng ở dạng long ta thêm tiếp vĩ ngữ “l” hay “L”. Hằng có thể là số thực: trường hợp giá trị hằng có kiểu float ta thêm tiếp vĩ ngữ “f” hay “F”, còn kiểu double thì thêm tiếp vĩ ngữ “d” hay “D”. Hằng Boolean: có 2 giá trị là true và false.

6. Các toán tử

a) Các toán tử cơ bản

Java cung cấp rất nhiều toán tử đa dạng để thao tác với các biến. Có thể chia các toán tử trong Java thành các nhóm sau:

- Toán tử số học
- Toán tử quan hệ
- Toán tử logic
- Toán tử thao tác bit
- Toán tử gán
- Toán tử instanceof
- Toán tử điều kiện
- Toán tử số học

Các toán tử số học được sử dụng trong các biểu thức toán học theo cách tương tự như chúng được sử dụng trong đại số. **Error! Reference source not found.** liệt kê các toán tử số học trong Java với 2 toán hạng A=10, B=20.

Toán tử	Mô tả	Ví dụ
+	Phép cộng	A + B sẽ cho kết quả 30
-	Phép trừ: trừ toán hạng trái cho toán hạng phải	A - B sẽ cho kết quả -10
*	Phép nhân	A * B sẽ cho kết quả 200
/	Phép chia: chia toán hạng trái cho toán hạng phải	B / A sẽ cho kết quả 2
%	Phép chia lấy phần dư: Lấy phần dư của phép chia toán hạng trái cho toán hạng phải	B % A sẽ cho kết quả 0

<code>++</code>	Tăng giá trị của biến lên 1.	Ví dụ <code>A++</code> sẽ là 11
<code>--</code>	Giảm giá trị của biến 1 đơn vị.	Ví dụ <code>A--</code> sẽ là 9

Bảng 1.2. Các toán tử số học trong Java.

Toán tử quan hệ

Các toán tử quan hệ được sử dụng để kiểm tra mối quan hệ giữa hai toán hạng. Kết quả của một biểu thức có dùng các toán tử quan hệ là những giá trị logic (“true” hoặc “false”). Các toán tử quan hệ được sử dụng trong các cấu trúc điều khiển. **Error! Reference source not found.** liệt kê các toán tử quan hệ trong Java.

Toán tử	Mô tả
<code>==</code>	So sánh bằng: Toán tử này kiểm tra sự tương đương của hai toán hạng
<code>!=</code>	So sánh khác: Toán tử này kiểm tra sự khác nhau của hai toán hạng
<code>></code>	Lớn hơn Kiểm tra giá trị của toán hạng bên phải lớn hơn toán hạng bên trái hay không
<code><</code>	Nhỏ hơn Kiểm tra giá trị của toán hạng bên phải có nhỏ hơn toán hạng bên trái hay không
<code>>=</code>	Lớn hơn hoặc bằng Kiểm tra giá trị của toán hạng bên phải có lớn hơn hoặc bằng toán hạng bên trái hay không
<code><=</code>	Nhỏ hơn hoặc bằng Kiểm tra giá trị của toán hạng bên phải có nhỏ hơn hoặc bằng toán hạng bên trái hay không

Bảng 1.3. Các toán tử quan hệ trong Java.

Toán tử logic

Các toán tử logic làm việc với các toán hạng Boolean. **Error! Reference source not found.** liệt kê các toán tử logic trong Java.

Toán tử	Mô tả
&&	Toán tử và (AND) Trả về giá trị True nếu chỉ khi cả hai toán tử có giá trị True
	Toán tử hoặc (OR) Trả về giá trị True nếu ít nhất một giá trị là True
^	Toán tử XOR Trả về giá trị True nếu và chỉ nếu chỉ một trong các giá trị là True, các trường hợp còn lại cho giá trị False
!	Toán tử phủ định (NOT) Toán hạng đơn từ NOT. Chuyển giá trị từ True sang False và ngược lại.

Bảng 1.4. Các toán tử logic trong Java.

Toán tử thao tác bít

Các toán tử dạng bit cho phép thực hiện thao tác trên từng bit riêng biệt trong các kiểu dữ liệu nguyên thuỷ. **Error! Reference source not found.** liệt kê các toán tử thao tác bít trong Java.

Toán tử	Mô tả
~	Phủ định NOT Trả về giá trị phủ định của một bít. Trả về một giá trị “Đúng” (True) nếu chỉ khi cả hai toán tử có giá trị “True”
&	Toán tử AND Trả về giá trị là 1 nếu các toán hạng là 1 và 0 trong các trường hợp khác
	Toán tử OR Trả về giá trị là 1 nếu một trong các toán hạng là 1 và 0 trong các trường hợp khác.
^	Toán tử Exclusive OR

	Trả về giá trị là 1 nếu chỉ một trong các toán hạng là 1 và trả về 0 trong các trường hợp khác.
>>	Dịch phải Chuyển toàn bộ các bít của một số sang phải một vị trí, giữ nguyên dấu của số âm. Toán hạng bên trái là số bị dịch còn số bên phải chỉ số vị trí mà các bít cần dịch.
<<	Dịch trái Chuyển toàn bộ các bít của một số sang trái một vị trí, giữ nguyên dấu của số âm. Toán hạng bên trái là số bị dịch còn số bên phải chỉ số vị trí mà các bít cần dịch.

Bảng 1.5. Toán tử thao tác bít trong Java.

Toán tử gán

Toán tử gán (=) dùng để gán một giá trị vào một biến và có thể gán nhiều giá trị cho nhiều biến cùng một lúc.

Ví dụ 1. 26:

```
int var = 20;
int p,q,r,s;
p=q=r=s=var;
```

Trong ví dụ trên, dòng lệnh `int var = 20` có nghĩa là gán một giá trị cho biến `var` và giá trị này lại được gán cho nhiều biến với dòng lệnh `p=q=r=s=var`. Dòng lệnh này được thực hiện từ phải qua trái. Đầu tiên giá trị ở biến `var` được gán cho ‘s’, sau đó giá trị của ‘s’ được gán cho ‘r’ và cứ tiếp như vậy. Toán tử gán còn được dùng phô biến để gán giá trị của các vế trong biểu thức với nhau, bao gồm:

= : Gán giá trị của toán hạng bên phải cho bên trái. Ví dụ $A = A + 16$.

+= : Gán giá trị của toán hạng bên phải cộng với chính giá trị toán hạng bên trái cho toán hạng bên trái. Ví dụ $A+=16$ (tương tự $A+16$).

`-=` : Gán giá trị của toán hạng bên phải trừ với chính giá trị toán hạng bên trái cho toán hạng bên trái. Ví dụ `A-=16` (tương tự `A-16`).

`*=` : Gán giá trị của toán hạng bên phải nhân với chính giá trị toán hạng bên trái cho toán hạng bên trái. Ví dụ `A*=16` (tương tự `A*16`).

`/=` : Gán giá trị của toán hạng bên phải chia cho chính giá trị toán hạng bên trái cho toán hạng bên trái. Ví dụ `A/=16` (tương tự `A/16`).

`%=` : Gán giá trị của toán hạng bên phải lấy dư với chính giá trị toán hạng bên trái cho toán hạng bên trái. Ví dụ `A%=16` (tương tự `A%16`).

`<<=` : Xử lý như toán tử bit, nhưng gán giá trị cho biến hiện tại.

`>>=` : Xử lý như toán tử bit, nhưng gán giá trị cho biến hiện tại.

`&=` : Xử lý như toán tử bit, nhưng gán giá trị cho biến hiện tại.

`^=` : Xử lý như toán tử bit, nhưng gán giá trị cho biến hiện tại.

`|=` : Xử lý như toán tử bit, nhưng gán giá trị cho biến hiện tại.

Toán tử *instanceof*

Toán tử này chỉ được sử dụng cho các biến tham chiếu đối tượng. Toán tử thường dùng để kiểm tra kiểu dữ liệu của biến. Toán tử *instanceof* được viết như sau:

```
( Object reference variable ) instanceof (class/interface type)
```

Ví dụ 1.27:

```
public class Test {  
    public static void main(String args[]){  
        String name = "gpcoder";  
        // Kiểm tra name có phải là kiểu chuỗi hay không  
        boolean result = name instanceof String;  
        System.out.println( result ); // kết quả: true  
    }  
}
```

Toán tử điều kiện

Toán tử điều kiện là một loại toán tử đặc biệt vì nó bao gồm ba thành phần câu thành biểu thức điều kiện. Cú pháp:

```
<biểu thức 1> ? <biểu thức 2> : <biểu thức 3>;
```

- Biểu thức 1: Biểu thức logic. Trả trả về giá trị True hoặc False.
- Biểu thức 2: Là giá trị trả về nếu biểu thức 1 xác định là True.
- Biểu thức 3: Là giá trị trả về nếu biểu thức 1 xác định là False.

Ví dụ 1.28:

```
public class Test {
    public static void main(String[] args) {
        int a = 20;
        int b = 3;

        String s = (a % b == 0) ? "a chia het cho b" : "a
khong chia het cho b";
        System.out.println(s);
    }
}
```

b) Thứ tự ưu tiên của các toán tử và quy tắc kết hợp

Thứ tự ưu tiên của các toán tử trong một biểu thức là thứ tự thực hiện các phép tính trong biểu thức đó. Trong một biểu thức, các toán tử có quyền ưu tiên cao nhất được tính toán đầu tiên. **Error! Reference source not found.** liệt kê thứ tự ưu tiên của các toán tử với quyền ưu tiên của các toán tử được xếp theo thứ tự từ cao tới thấp tương ứng với thứ tự hàng từ trên xuống dưới của bảng. Khi hai toán tử có cùng độ ưu tiên trong một biểu thức, các quy tắc kết hợp (thể hiện ở cột “**Thứ tự ưu tiên**” trong **Error! Reference source not found.**) sẽ cho trình biên dịch biết nên thực hiện phép tính cùng cấp độ ưu tiên theo thứ tự từ trái sang phải hay từ phải sang trái.

Loại	Toán tử	Thứ tự ưu tiên
Postfix	() [] . (toán tử dot)	Trái sang phải
Unary	++ - - ! ~	Phải sang trái
Tính nhân	* / %	Trái sang phải
Tính cộng	+ -	Trái sang phải
Dịch chuyển	>> >>> <<	Trái sang phải

Quan hệ	$> >= < <=$	Trái sang phải
Cân bằng	$== !=$	Trái sang phải
Phép AND bit	$\&$	Trái sang phải
Phép XOR bit	$^$	Trái sang phải
Phép OR bit	$ $	Trái sang phải
Phép AND logic	$\&\&$	Trái sang phải
Phép OR logic	$\ $	Trái sang phải
Điều kiện	$?:$	Phải sang trái
Gán	$= += -= *= /= \% = >>= <<= \&= ^= =$	Phải sang trái
Dấu phẩy	,	Trái sang phải

Bảng 1.6. Thứ tự ưu tiên của các toán tử.

Ví dụ 1.29:

```
int a = 27, b = 18, c = 20;
System.out.println(a % b * c + a / b);
```

Trong biểu thức trên, biểu thức $b*c$ sẽ được tính toán trước và cho ra kết quả là 360, sau đó a/b sẽ được tính toán với kết quả là 1.5, tiếp theo tới $a\%360$ với kết quả là 0.075. Kết quả cuối cùng là $1.5 + 0.075 = 1.575$.

Lưu ý: Để điều chỉnh độ ưu tiên mặc định của các toán tử như bảng trên, ta sử dụng dấu ngoặc đơn. Những biểu thức nằm trong dấu ngoặc đơn sẽ được Java tính toán trước. Sau khi tính xong những biểu thức này, thứ tự tính toán tiếp theo sẽ dựa vào độ ưu tiên mặc định.

Ví dụ 1.30:

```
int a = 27, b = 18, c = 20;
System.out.println((a % b) * (c + a) / b);
```

Trong ví dụ trên, kết quả cuối cùng là 23.

7. Nhập, xuất dữ liệu và các cú pháp điều khiển cơ bản

a) Nhập dữ liệu từ bàn phím và xuất ra màn hình

Java cung cấp lớp Scanner để nhập dữ liệu vào từ bàn phím. Cách khai báo để sử dụng lớp đó như sau:

```
Scanner sc = new Scanner(System.in);
```

Ví dụ 1.31: yêu cầu người dùng nhập tên và tuổi vào từ bàn phím

```
import java.util.Scanner;
public class InputValueFromKeyboard {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String ten;
        System.out.println("Moi ban nhap ten: ");
        ten = sc.nextLine();
        int tuoi;
        System.out.println("Moi ban nhap tuoi : ");
        tuoi = sc.nextInt();
    }
}
```

Sau khi khai báo lớp Scanner sc, để nhận giá trị nguyên ta dùng `sc.nextInt()`, nhận giá trị chuỗi dùng `sc.nextLine()`. Tương tự với giá trị `float` thì sẽ là `sc.nextFloat()`, `double` sẽ là `nextDouble()`, `long` sẽ là `nextLong()`.

Để hiển thị những thông tin vừa nhập ra ngoài màn hình, ta sử dụng câu lệnh:

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

Ví dụ 1.32:

```
package inputvaluefromkeyboard;
import java.util.Scanner;
public class InputValueFromKeyboard {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String ten;
        System.out.println("Moi ban nhap ten: ");
        ten = sc.nextLine();
        int tuoi;
        System.out.println("Moi ban nhap tuoi : ");
        tuoi = sc.nextInt();
        System.out.println("Thong tin cua ban vua nhap la : ");
        System.out.println("Ten cua ban : "+ ten);
        System.out.println("Tuoi cua ban : "+ tuoi);
    }
}
```

```
    }  
}
```

b) Các cú pháp điều khiển cơ bản

Cũng như các ngôn ngữ lập trình khác, Java hỗ trợ bộ cấu trúc điều khiển cơ bản gồm cấu trúc rẽ nhánh, cấu trúc vòng lặp, cấu trúc nhảy...

Cấu trúc rẽ nhánh

Cấu trúc rẽ nhánh còn gọi là cấu trúc lựa chọn hay cấu trúc quyết định, bao gồm các cấu trúc if, if-else, if-else-if, và switch-case.

- Cấu trúc rẽ nhánh cơ bản if:

Cú pháp:

```
if(biểu thức logic){  
    khối lệnh;  
}
```

Ý nghĩa: Nếu biểu thức logic trả về giá trị true thì thực thi khối lệnh. Nếu biểu thức logic trả về giá trị false thì bỏ qua khối lệnh này.

Ví dụ 1.33:

```
public class Test{  
    public static void main(String[] args) {  
        int diemTB = 10;  
        if (diemTB > 9){  
            System.out.print("Hoc luc Gioi");  
        }  
    }  
}
```

Ngoài cấu trúc cơ bản trên, có thể thực hiện cấu trúc lồng if trong Java:

Cú pháp:

```
if(biểu thức logic 1){  
    khối lệnh 1;  
    if(biểu thức logic 2){  
        khối lệnh 2;  
    }  
}
```

Ý nghĩa: Nếu biểu thức logic 1 trả về giá trị true thì thực hiện khối lệnh 1, ngược lại xét biểu thức logic 2 nếu trả về giá trị true thì thực hiện khối lệnh 2, nếu trả về giá trị false thì bỏ qua khối lệnh này.

Ví dụ 1.34:

```
public class Test{  
  
    public static void main(String args[]){  
        int time_s = 30;  
        int round_v = 10;  
  
        if( time_s == 30 ){  
            if( round_v == 10 ){  
                System.out.print("Phu hop tieu chuan chung");  
            }  
        }  
    }  
}
```

- Cấu trúc rẽ nhánh mở rộng dạng if-else:

Cú pháp:

```
if(biểu thức logic){  
    khối lệnh 1;  
}  
else{  
    khối lệnh 2;  
}
```

Ý nghĩa: Nếu biểu thức logic trả về giá trị true thì thực thi khối lệnh 1, ngược lại nếu biểu thức logic trả về giá trị false thì thực thi khối lệnh 2.

Ví dụ 1.35:

```
public class KiemTraChanLe{  
    public static void main(String[] args){  
        int a = 13;  
        if (a % 2 == 0) {  
            System.out.println("Số " + a + " là số chẵn.");  
        }  
        else{  
            System.out.println("Số " + a + " là số lẻ.");  
        }  
    }  
}
```

- Cấu trúc rẽ nhánh mở rộng dạng if-else-if:

Cú pháp:

```
if(biểu thức logic1){  
    khôi lệnh 1;  
} else if(biểu thức logic2){  
    khôi lệnh 2;  
} else if(biểu thức logic3){  
    khôi lệnh 3;  
}  
....  
else {  
    khôi lệnh n;  
}
```

Ý nghĩa: Nếu biểu thức sau if (biểu thức logic1) trả về giá trị true thì chỉ có khôi lệnh sau if (khôi lệnh 1) được thực hiện. Nếu bất kỳ biểu thức logic sau if else nào là true thì chỉ khôi lệnh sau else if đó được thực hiện... Nếu tất cả biểu thức logic của if và else if là false thì chỉ có khôi lệnh sau else (khôi lệnh n) sẽ được thực hiện.

Ví dụ 1.36:

```
public class XepLoai {  
    public static void main(String[] args) {  
        int diemTB = 6.5;  
  
        if (diemTB < 5) {  
            System.out.println("Xếp loại yếu!");  
        } else if (diemTB >= 5 && diemTB < 6) {  
            System.out.println("Xếp loại trung bình");  
        } else if (diemTB >= 6 && diemTB < 7) {  
            System.out.println("Xếp loại trung bình khá");  
        } else if (diemTB >= 7 && diemTB < 8) {  
            System.out.println("Xếp loại khá");  
        } else if (diemTB >= 8 && marks < 9) {  
            System.out.println("Xếp loại giỏi");  
        } else if (diemTB >= 9 && diemTB < 10) {  
            System.out.println("Xếp loại xuất sắc");  
        } else {  
            System.out.println("Giá trị không hợp lệ!");  
        }  
    }  
}
```

- Câu trúc switch-case

Khi cần xử lý các sự kiện liên quan tới nhiều trường hợp giá trị của biến, nếu dùng if-else nhiều thì chương trình sẽ dài, lặp lại và không mạch lạc. Do đó, câu trúc switch-case được sử dụng thay thế if-else.

Cú pháp:

```
switch (<biến>) {  
    case <giá trị_1> :  
        <khởi_lệnh_1>;  
        break;  
    case <giá trị_2>:  
        <khởi_lệnh_2>;  
        break;  
    ...  
    case <giá trị_n>:  
        <khởi_lệnh_n>;  
        break;  
    default:  
        <khởi_lệnh_default>;  
}
```

Ý nghĩa: Lệnh switch cho phép kiểm tra một biến bình đẳng với một danh sách các giá trị. Mỗi giá trị được gọi là một trường hợp (case). Nếu giá trị này trùng với case nào thì các lệnh tương ứng với case đó sẽ được thực thi.

Ví dụ 1.37:

```
public class SwitchDemo {  
    public static void main(String[] args) {  
        int a = 3;  
        switch (a) {  
            case 1: System.out.println("Xep hang A");  
            break;  
  
            case 2: System.out.println("Xep hang B");  
            break;  
            case 3: System.out.println("Xep hang C");  
            break;  
            default:  
                System.out.println("Bạn gán sai giá trị, chỉ được gán số nguyên từ 1 tới 3");  
                break;  
        }  
    }  
}
```

- Lưu ý khi sử dụng switch:

+ Biến được sử dụng trong một lệnh switch chỉ có thể là byte, short, int, char. Từ JDK 7, hỗ trợ thêm kiểu String.

+ Có thể có nhiều lệnh case bên trong một lệnh switch. Mỗi case được theo sau bởi giá trị để được so sánh và một dấu hai chấm.

+ Giá trị cho một case phải giống kiểu dữ liệu của biến trong switch và nó phải là hằng số hoặc ký tự.

+ Các lệnh sau case sẽ được thực thi tới khi gặp lệnh break. Khi gặp một lệnh break thì switch kết thúc và luồng điều khiển chuyển tới dòng tiếp theo của lệnh switch. Không phải case nào cũng cần một break. Nếu không có lệnh break xuất hiện, luồng điều khiển sẽ đi qua các case tới khi gặp một lệnh break.

+ Một lệnh switch có thể có một case mặc định (default) xuất hiện ở cuối lệnh switch. Case mặc định này có thể được sử dụng để thực thi một tác vụ trong trường hợp không có case nào là true. Trong trường hợp này, không cần lệnh break.

Cấu trúc lặp for

Sử dụng vòng lặp for khi biết số lần một tác vụ được lặp lại. Có 3 kiểu vòng lặp for trong Java: Vòng lặp for đơn giản; Vòng lặp for cải tiến; Vòng lặp for gán nhãn.

- Vòng lặp for đơn giản.

Cú pháp:

```
for(khoi_tao_bien; bieu_thuc_logic; tang/giam_bien){  
    khôi lệnh;  
}
```

Ý nghĩa: Bước *khoi_tao_bien* được thực thi đầu tiên, và chỉ một lần. Bước này cho phép khai báo và khởi tạo biến điều khiển vòng lặp. Sau đó, xét *biau_thuc_logic*, nếu nó là false, phần thân vòng lặp không thực thi và luồng điều khiển chuyển tới lệnh tiếp theo sau vòng lặp for; nếu nó là true, các lệnh trong thân vòng lặp được lặp lượt thực hiện; sau đó *tang/giam_bien* điều khiển vòng lặp và lặp lại các công việc trên từ khâu xét *biau_thuc_logic*.

Lưu ý: *khoi_tao_bien*, *bieu_thuc_logic*, *tang/giam_bien* có thể vắng mặt trong cấu trúc của vòng lặp for nhưng phải có các dấu chấm phẩy.

Ví dụ 1.38: Vòng lặp for có đầy đủ *khoi_tao_bien*, *bieu_thuc_logic* và *tang/giam_bien*

```
public class ForDemo {  
  
    public static void main(String args[]) {  
  
        for(int x = 1; x < 10; x = x+1) {  
            System.out.print("Gia tri cua x : " + x );  
            System.out.print("\n");  
        }  
    }  
}
```

Ví dụ 1.39: Vòng lặp for không có *khoi_tao_bien*

```
int sum = 0, k = 1;  
for (; sum < 100; k++) {  
    sum += k;  
}
```

Ví dụ 1.40: Vòng lặp for không có *khoi_tao_bien* và *tang/giam_bien*

```
int sum = 0, k = 1;  
for (; sum < 100; ) {  
    sum += k;  
    k++;  
}
```

Ví dụ 1.41: Vòng lặp for không có *khoi_tao_bien*, *bieu_thuc_logic* và *tang/giam_bien* khi đó thân của vòng lặp sẽ được thực hiện liên tục, do vậy trong thân của vòng lặp phải có lệnh để thoát ra ngoài vòng for.

```
for ( ; ; ) {  
    System.out.println("Vong lap khong xac dinh");  
}
```

Khi chạy đoạn mã trên, dòng chữ “Vong lap khong xac dinh” sẽ được in ra cho đến khi vòng lặp được ngắt bằng tay. Ta có thể sử dụng câu lệnh *break* để ngắt vòng lặp. Thông thường khi nhập giá trị đầu vào của chương trình không được biết trước ta có thể sử dụng vòng lặp không xác định, khi đó chương trình

sẽ chờ cho đến khi người dùng nhập dữ liệu. Khi người dùng nhập dữ liệu xong, hệ thống sẽ xử lý đầu vào, sau đó lại bắt đầu thực thi vòng lặp vô hạn.

- Vòng lặp for cải tiến:

Vòng lặp for cải tiến được sử dụng để lặp mảng (array) hoặc tập hợp (collection) trong Java. Sử dụng vòng lặp này dễ dàng hơn vòng lặp for đơn giản vì không cần tăng hay giảm giá trị của biến rồi kiểm tra điều kiện mà chỉ cần sử dụng ký hiệu hai chấm “::”.

Cú pháp:

```
for (Type var : array) {  
    Khối lệnh;  
}
```

Ví dụ 1.42:

```
public class ForEachExample {  
    public static void main(String[] args) {  
        int arr[] = { 12, 23, 44, 56, 78 };  
        for (int i : arr) {  
            System.out.println(i);  
        }  
    }  
}
```

- Vòng lặp for gán nhãn:

Ta có thể đặt tên cho mỗi vòng lặp for bằng cách gán nhãn trước vòng lặp for. Điều này rất hữu dụng khi muốn thoát/tiếp tục(break/continues) chạy vòng lặp for.

Cú pháp:

```
ten_nhan:  
for (khoi_tao_bien ; biểu_thức_logic ; tang/giam_bien) {  
    Khối lệnh;  
}
```

Ví dụ 1.43:

```
public class LabeledForExample {  
    public static void main(String[] args) {  
        aa: for (int i = 1; i <= 3; i++) {
```

```

        bb: for (int j = 1; j <= 3; j++) {
            if (i == 2 && j == 2) {
                break aa;
            }
            System.out.println(i + " " + j);
        }
    }
}

```

Cấu trúc lặp while, do-while

- Vòng lặp while: Được sử dụng trong trường hợp số lần lặp không được xác định trước.

Cú pháp:

```

while(biểu thức logic) {
    Khối lệnh;
}

```

Ý nghĩa: Trước tiên, biểu thức logic được xem xét. Nếu nó trả về giá trị true thì khối lệnh trong thân vòng lặp được thực hiện cho đến khi biểu thức logic trả về giá trị false.

Ví dụ 1.44:

```

public class WhileTest{
    public static void main(String[] args) {
        int i = 1;
        while (i <= 10){
            System.out.println(i);
            i++;
        }
    }
}

```

* Lưu ý: Nếu để biểu thức logic luôn trả về true thì vòng lặp while sẽ chạy đến vô tận cho đến khi ta dừng chương trình trên IDE (Eclipse, Netbean...) hoặc bấm Ctrl + C khi chạy trên giao diện dòng lệnh. Trong ví dụ trên để vòng lặp while dừng sau một số bước lặp nhất định ta thực hiện lệnh i++ trong thân vòng lặp.

Ví dụ 1.45: Vòng lặp while vô tận

```

public class WhileTestForever{
    public static void main(String[] args){
        while (true){
            System.out.println("Vòng lặp while vô tận...");
        }
    }
}

```

- Vòng lặp do-while: Được sử dụng tương tự như vòng lặp while, ngoại trừ do-while thực hiện lệnh ít nhất một lần ngay cả khi biểu thức logic là false.

Cú pháp:

```

do{
    Khởi lệnh;
}while(biểu thức logic);

```

Ví dụ 1.46:

```

public class DoWhileTest{
    public static void main(String[] args) {
        int a = 1, tong = 0;
        do{
            tong += a;
            a++;
        }while (a <= 5);
        System.out.println("Tong cac so tu 1 den 5 la:" + tong);
    }
}

```

- * Lưu ý: Nếu để biểu thức logic luôn trả về true thì vòng lặp do-while sẽ chạy đến vô tận cho đến khi ta dừng chương trình trên IDE (Eclipse, Netbean...) hoặc bấm Ctrl + C khi chạy trên giao diện dòng lệnh.

Ví dụ 1.47: Vòng lặp do-while vô tận

```

public class DoWhileTestForever{
    public static void main(String[] args) {
        do{
            System.out.println("Vòng lặp do-while vô tận...");
        }while (true);
    }
}

```

Các lệnh chuyển điều khiển

- Lệnh break

Lệnh break được sử dụng để dừng việc thực thi lệnh trong vòng lặp hoặc trong mệnh đề switch tại điều kiện đã được chỉ định. Đối với vòng lặp bên trong vòng lặp khác, lệnh break chỉ dừng vòng lặp bên trong đó.

Trong java, lệnh break được sử dụng theo hai cách là có nhãn (labeled) và không có nhãn (unlabeled). Lệnh break thường được sử dụng ở dạng không nhãn khi ta cần dừng vòng lặp tại vị trí nào đó và nó chỉ có tác dụng đối với vòng lặp chứa break. Lệnh break có nhãn được dùng để ngắt các vòng lặp lồng nhau.

Ví dụ 1.48: Lệnh break không có nhãn với vòng lặp for:

```
public class BreakTest1 {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break;
            }
            System.out.println(i);
        }
    }
}
```

Ví dụ 1.49: Lệnh break không có nhãn với vòng lặp bên trong vòng lặp for khác:

```
public class BreakTest12{
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2) {
                    break;
                }
                System.out.println(i + " " + j);
            }
        }
    }
}
```

Lúc này lệnh break chỉ có tác dụng ngắt vòng lặp với biến j.

Ví dụ 1.50: Lệnh break có nhãn

```
MyBreakLabel:
for (int i = 0; i < 5; i++) {
    System.out.println("i: " + i);
    for (int j = 0; j < 5; j++) {
```

```

        System.out.println("j : " + j);
        if (j == 3) {
            break MyBreakLabel;
        }
    }
}

```

Khi gặp câu lệnh `break MyBreakLabel;` thì cả vòng lặp `i` và `j` đều bị ngắt.

- Lệnh continue

Giống như lệnh `break`, lệnh `continue` cũng được dùng kết hợp với cấu trúc lặp. Khi gặp từ khóa `continue` thì lần lặp kế tiếp sẽ được thực hiện. Lệnh `continue` thường được dùng với một lệnh `if` bên trong vòng lặp để kiểm tra khi nào cần bỏ qua những lệnh sau nó để tiếp tục thực hiện vòng lặp mới.

Lưu ý: Nếu có nhiều cấu trúc lặp lồng nhau thì `continue` chỉ có tác dụng với cấu trúc lặp trong cùng chung nó.

Ví dụ 1.51: Chương trình in ra màn hình các dòng thông báo "*Chuỗi thứ 1*"... "*Chuỗi thứ 4*", sau đó in ra màn hình dòng "*Kết thúc chuỗi!*", rồi tiếp đó in ra thông báo "*Chuỗi thứ 5*" và cuối cùng in ra chuỗi "*Kết thúc chuỗi!*" rồi kết thúc chương trình.

```

public static void main(String[] args) {
    int count;
    for (count = 1; count <= 5; count++) {
        System.out.println("Chuỗi thứ " + count);
        /*
         * kiểm tra nếu count còn nhỏ hơn 4
         * thì còn quay lại vòng for kiểm tra điều kiện lặp
         */
        if (count < 4) {
            continue;
        }
        // Nếu count không nhỏ hơn 4 thì hiển thị "Kết thúc chuỗi!"
        System.out.println("Kết thúc chuỗi!");
    }
}

```

8. Mảng và kiểu liệt kê enum

a) *Mảng*

Mảng là kiểu dữ liệu có cấu trúc gồm một tập hợp cố định các phần tử có cùng kiểu dữ liệu, các phần tử của mảng có cùng tên và được phân biệt nhau bởi chỉ số. Mỗi phần tử của mảng được sử dụng như là một biến đơn, kiểu dữ liệu của mảng chính là kiểu dữ liệu của phần tử. Đối với mảng ta cần quan tâm đến các thành phần sau:

- Các thông tin liên quan đến mảng: Tên mảng, số phần tử của mảng, kiểu dữ liệu của mảng.

- Số chiều của mảng: Nếu mảng chỉ có một chỉ số để lưu trữ các giá trị vào trong các biến thành phần của mảng thì được gọi là mảng một chiều. Nếu mảng có 2 chỉ số để lưu trữ các giá trị (ví dụ giá trị của một bảng có m dòng, n cột) được gọi là mảng 2 chiều. Tương tự, ta có mảng 3 chiều, 4 chiều, ..., n chiều.

Trong lập trình, mảng thường xuyên được sử dụng không chỉ bởi tính đơn giản, dễ sử dụng của nó mà còn ở khả năng đáp ứng nhu cầu lưu trữ dữ liệu trong các bài toán thực tế. Chúng ta có thể sử dụng mảng khi cần lưu trữ nhiều giá trị, ví dụ như lưu trữ các số nguyên từ 1 đến 5; dãy 32 chuỗi ký tự, trong đó mỗi chuỗi lưu trữ tên của một sinh viên trong một lớp học...

Mảng một chiều: Là một tập hợp của nhiều phần tử có kiểu dữ liệu giống nhau.

- Cú pháp khai báo mảng một chiều có 2 dạng sau:

```
[Kiểu_dữ_liệu] tên_mảng[];
```

Hoặc:

```
[Kiểu_dữ_liệu][] tên_mảng;
```

trong đó, [Kiểu_dữ_liệu] mô tả kiểu của mỗi phần tử thuộc mảng (*như int, char, double, String,...*), tên_mảng là tên của mảng và quy tắc đặt tên phải tuân theo quy tắc đặt tên biến trong Java.

Ví dụ: `int[] a;` khai báo mảng có tên là `a` và có kiểu dữ liệu là `int`

- Cấp phát bộ nhớ cho mảng:

Bản chất của mảng là 1 đối tượng vì vậy mảng cần được cấp phát bộ nhớ trước khi sử dụng. Để cấp phát bộ nhớ cho mảng ta có 2 cách như sau:

+ Cách 1:

```
[Kiểu_dữ_liệu] tên_mảng[] = new [Kiểu_dữ_liệu]  
[Số_phần_tử_của_mảng];
```

+ Cách 2:

```
[Kiểu_dữ_liệu][] tên_mảng = new [Kiểu_dữ_liệu]  
[Số_phần_tử_của_mảng];
```

Trong đó, [Số_phần_tử_của_mảng] chỉ ra số lượng phần tử tối đa mà mảng có thể lưu trữ, giá trị này phải là một số nguyên dương. Ngoài ra, Java còn cho phép vừa khai báo mảng và vừa khởi tạo giá trị cho mảng.

Ví dụ: `int[] a = new int[] {2, 10, 4, 8, 5};`: khai báo mảng một chiều có tên là `a`, kiểu dữ liệu là `int` và mảng này chứa 5 phần tử có giá trị lần lượt là 2, 10, 4, 8, 5.

- Truy xuất các phần tử của mảng

Đối với mảng ta có thể truy xuất các phần tử của mảng thông qua các chỉ số của phần tử đó. Cú pháp như sau:

```
Tên_mảng[Chi_số_phần_tử];
```

trong đó, [Chi_số_phần_tử] là số thứ tự của các phần tử trong mảng và bắt đầu từ 0. Như vậy, mảng có n phần tử thì các phần tử của nó có chỉ số lần lượt là 0, 1, 2, ..., $n - 1$.

Ví dụ 1.52:

```
public static void main(String[] args) {
    // Khai báo và khởi tạo giá trị ban đầu cho mảng
    char[] kyTu = new char[] {'a', 'b', 'c', 'd', 'e'};

    // hiển thị ký tự tại vị trí thứ 2 trong mảng
    System.out.println("Ký tự tại vị trí thứ 2 trong mảng là " +
kyTu[2]);
}
```

- Nhập xuất các phần tử mảng

Ví dụ 1.53: Chương trình dưới đây sẽ minh họa cách nhập các phần tử cho mảng một chiều từ bàn phím và sau đó hiển thị các phần tử đó ra màn hình.

```
package com.javacaban.array;

import java.util.Scanner;

public class ArrayExample1 {
    public static Scanner scanner = new Scanner(System.in);

    /**
     * main
     *
     * @param args
     */
    public static void main(String[] args) {
        System.out.print("Nhập số phần tử của mảng: ");
        int n = scanner.nextInt();
        // khởi tạo mảng arr
        int[] arr = new int[n];
        System.out.print("Nhập các phần tử của mảng: \n");
        for (int i = 0; i < n; i++) {
            System.out.printf("a[%d] = ", i);
            arr[i] = scanner.nextInt();
        }
        System.out.print("Các phần tử của mảng: ");
        show(arr);
    }

    /**
     * in các phần tử của mảng ra màn hình
     *
     * @param arr: mảng các số nguyên
     * @param n: số phần tử của mảng
     */
    public static void show(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

```
}
```

Trong ví dụ trên, ta sử dụng vòng lặp for để nhập từ bàn phím và in ra màn hình các phần tử của mảng. Ngoài ra, ta có thể sử dụng vòng lặp for cải tiến cho mảng.

Ví dụ 1.54:

```
public class Test {  
  
    public static void main(String args[]){  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ){  
            System.out.print( x );  
            System.out.print(",");  
        }  
        System.out.print("\n");  
        String [] names = {"James", "Larry", "Tom", "Lacy"};  
        for( String name : names ) {  
            System.out.print( name );  
            System.out.print(",");  
        }  
    }  
}
```

Mảng đa chiều: Trong mảng đa chiều, dữ liệu được lưu trữ theo hàng và cột theo chỉ mục (hay còn gọi là dạng ma trận).

- Cú pháp khai báo mảng đa chiều:

```
[Kiểu_dữ_liệu] [][] tên_mảng[];
```

hoặc:

```
[Kiểu_dữ_liệu] [][]tên_mảng[];
```

hoặc:

```
[Kiểu_dữ_liệu] tên_mảng[][];
```

hoặc:

```
[Kiểu_dữ_liệu] []tên_mảng[];
```

Ví dụ 1.55: Khởi tạo và gán giá trị cho mảng đa chiều:

```
int[][] arr=new int[3][3]; // 3 hàng 3 cột  
arr[0][0]=1;
```

```

arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;

```

Ví dụ 1.56: Mảng đa chiều

```

public class TestArray3 {
    public static void main(String args[]) {

        // khai báo và khởi tạo mảng 2 chiều
        int arr[][] = { { 1, 2, 3 }, { 2, 4, 5 }, { 4, 4, 5 } };

        // in mảng 2 chiều ra màn hình
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }

    }
}

```

b) Kiểu liệt kê enum

Kiểu liệt kê enum trong Java đã được giới thiệu trong JDK 1.5. Nó định nghĩa một tập hợp các hằng số.

- Khai báo enum:

Trong Java, enum có thể được định nghĩa bên trong hoặc bên ngoài một lớp, vì nó tương tự như lớp trong java.

Ví dụ 1.57: enum định nghĩa bên trong một lớp:

```

public class EnumExample {
    enum WeekDay {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
        SUNDAY;
    }

    public static void main(String[] args) {
        WeekDay d = WeekDay.MONDAY;
        System.out.println(d);
    }
}

```

```
}
```

Ví dụ 1.58: enum định nghĩa bên ngoài một lớp:

```
enum WeekDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY;
}

public class EnumExample {
    public static void main(String[] args) {
        WeekDay d = WeekDay.MONDAY;
        System.out.println(d);
    }
}
```

Lưu ý: enum định nghĩa bên ngoài một lớp không thể dùng access modifier là public, protected hay private.

Ví dụ 1.59: enum định nghĩa trong một file riêng biệt:

WeekDay.java

```
public enum WeekDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY;
}
```

EnumExample.java

```
public class EnumExample {
    public static void main(String[] args) {
        WeekDay d = WeekDay.MONDAY;
        System.out.println(d);
    }
}
```

- Duyệt các phần tử trong enum

Chúng ta có thể duyệt trên tất cả các phần tử của enum, thông qua phương thức **values()**. Trình biên dịch trong java tự động thêm phương thức values() vào enum khi nó được biên dịch. Phương thức values() trả về một mảng chứa tất cả các giá trị của enum.

Ví dụ 1.60:

```
public class EnumExample {
```

```
enum WeekDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY;
}
public static void main(String[] args) {
    for (WeekDay d : WeekDay.values()) {
        System.out.println(d);
    }
}
```

Câu hỏi và bài tập cuối chương

Câu 1. Tải và cài đặt phiên bản JDK mới nhất. Cấu hình, cài đặt biến môi trường cho Java. Cài đặt Java Runtime Environment để lập trình Java và kiểm tra phiên bản Java trên giao diện dòng lệnh.

Câu 2. Cài đặt và sử dụng thành thạo các IDE cho Java, bao gồm NetBean, IntelliJ IDEA, Eclipse IDE.

Câu 3. Sử dụng IDE để tạo và chạy chương trình Java in ra dòng chữ “Hello Java”.

Câu 4. Bài tập cơ bản:

- Nhập vào một số thực là bán kính của một hình tròn. In ra chu vi, diện tích của hình tròn.

- Nhập vào 3 số thực a, b, c là 3 cạnh của một tam giác. In ra chu vi, diện tích của tam giác đó.

- Viết chương trình tính giai thừa của một số nguyên nhập vào từ bàn phím.

- Viết chương trình nhập số nguyên n và kiểm tra n có phải số nguyên tố hay không?

- Viết chương trình tìm ước số chung lớn nhất của hai số nguyên dương a, b nhập từ bàn phím.

- Viết chương trình phân tích một số nguyên thành các thừa số nguyên tố.

- Viết chương trình liệt kê n số nguyên tố đầu tiên.

- Nhập dãy số thực gồm n phần tử. Sắp xếp dãy số theo thứ tự tăng dần và in kết quả ra màn hình.

Câu 5. Bài tập về các cú pháp điều khiển:

- Viết chương trình giải phương trình bậc 2.

- Viết chương trình in ra tổng của 10 số chẵn đầu tiên

- Viết chương trình nhập vào 2 số nguyên, tìm số lớn nhất của 2 số.

- Viết chương trình tính lương của nhân viên dựa theo thời gian công tác (TNCT) như sau:

Lương = hệ số * lương căn bản, trong đó lương căn bản là 650000 đồng.

Nếu TNCT < 12 tháng: hệ số = 1.92

Nếu 12 <= TNCT < 36 tháng: hệ số = 2.34

Nếu 36 <= TNCT < 60 tháng: hệ số = 3

Nếu TNCT >= 60 tháng: hệ số = 4.5

- Viết chương trình đếm và in ra số lượng các số nguyên chia hết cho 3 hoặc 7 nằm trong khoảng từ 1 đến 100.

- Viết chương trình nhập vào số nguyên n. Tính và in ra tổng sau:

$$S = 1 + 2 + 3 + \dots + n$$

- Viết chương trình nhập vào các số nguyên và tính tổng các số đó, nếu tổng lớn hơn > 100 thì kết thúc vòng lặp và hiển thị thông báo tổng của các số đã nhập.

- Viết chương trình hiển thị ra màn hình các số chẵn nhỏ hơn hoặc bằng 20, mỗi số được in ra phải nằm trên 1 dòng.

- Viết chương trình nhập vào 2 số nguyên dương, tìm và in ra màn hình ước số chung lớn nhất và bội số chung nhỏ nhất của 2 số nguyên dương nhập vào từ bàn phím.

- Viết chương trình hiển thị ra màn hình các số chẵn nhỏ hơn hoặc bằng 20, mỗi số được in ra phải nằm trên 1 dòng.

- Viết chương trình nhập vào 2 số nguyên dương, tìm và in ra màn hình ước số chung lớn nhất và bội số chung nhỏ nhất của 2 số nguyên dương nhập vào từ bàn phím.

- Viết chương trình nhập vào 1 số nguyên dương n ($0 < n \leq 10$), tính và in ra màn hình n giai thừa.

Câu 6. Bài tập về mảng:

- Viết chương trình nhập vào một mảng số nguyên có n phần tử và thực hiện các công việc sau:

+ Xuất giá trị các phần tử của mảng.

+ Tìm phần tử có giá trị lớn nhất, nhỏ nhất.

+ Đếm số phần tử là số chẵn.

- Nhập một mảng số nguyên $a_0, a_1, a_2, \dots, a_{n-1}$. In ra màn hình các phần tử xuất hiện trong mảng đúng 1 lần.

- Nhập một mảng số nguyên $a_0, a_1, a_2, \dots, a_{n-1}$. In ra màn hình số lần xuất hiện của các phần tử.

- Nhập một mảng số nguyên $a_0, a_1, a_2, \dots, a_{n-1}$. Hãy sắp xếp mảng theo thứ tự tăng dần.

- Viết chương trình nhập vào mảng A có n phần tử, các phần tử là số nguyên lớn hơn 0 và nhỏ hơn 100. Thực hiện:

+ Tìm phần tử lớn thứ nhất và lớn thứ 2 trong mảng với các chỉ số của chúng (chỉ số đầu tiên tìm được).

+ Sắp xếp mảng theo thứ tự tăng dần.

+ Nhập số nguyên x và chèn x vào mảng A sao cho vẫn đảm bảo tính tăng dần cho mảng A.

Chương 2. LỚP VÀ ĐỐI TƯỢNG

I. TRÙU TƯỢNG HÓA ĐỐI TƯỢNG VÀ ĐÓNG GÓI DỮ LIỆU VÀO LỚP

1. Trùu tượng hóa đối tượng

Như đã đề cập ở Chương 1, trùu tượng hóa là cơ chế giúp giảm thiểu và bỏ qua các chi tiết không cần thiết của đối tượng để tập trung vào các đặc tính chính của chúng sao cho phù hợp với chương trình. Trùu tượng hóa có thể là trùu tượng hóa dữ liệu hoặc trùu tượng hóa chức năng. Ví dụ, trùu tượng hóa chức năng trong lập trình cấu trúc là sử dụng các phương thức và luồng điều khiển được định dạng. Trùu tượng hóa dữ liệu cho phép kiểm soát dữ liệu hiệu quả, chẳng hạn sử dụng kiểu dữ liệu. Trong lập trình hướng đối tượng, nguyên lý trùu tượng hóa được thực hiện thông qua việc trùu tượng hóa đối tượng theo chức năng và trùu tượng hóa đối tượng theo dữ liệu.

Một đối tượng là một thực thể đang tồn tại trong hệ thống bao gồm:

- Định danh: Tên gọi duy nhất của đối tượng, phân biệt đối tượng với các đối tượng khác.
- Trạng thái: là tổ hợp các giá trị thuộc tính mà đối tượng đang có.
- Hoạt động: là các hành động mà đối tượng có khả năng thực hiện được.

Ví dụ 2.1:

Trong bài toán quản lí Thư viện điện tử, mỗi độc giả (mỗi người dùng của hệ thống) được coi là một đối tượng. Chẳng hạn, một độc giả là sinh viên nam có mã số SV0001, tên “Trần Văn A”, sinh ngày 20/05/1990, quê ở Thái Bình, học lớp D46CNTT, thuộc khoa CN&ANTT là một đối tượng. Một độc giả khác là một giáo viên nữ có mã số GV0001, tên “Lê Thị B”, sinh ngày 23/03/1983, quê ở Hà Nội, công tác ở khoa CN&ANTT cũng là một đối tượng... Mỗi độc giả trên là một đối tượng có các trạng thái và hoạt động như sau:

	Trạng thái	Hoạt động
Độc giả sinh viên	<ul style="list-style-type: none"> • <i>Mã số</i> là SV0001 • <i>Mật khẩu</i> là SV0001 • <i>Tên</i> là Trần Văn A • <i>Ngày sinh</i> 20/05/1990 • <i>Giới tính</i> là Nam • <i>Quê quán</i> là Thái Bình • <i>Lớp</i> là D46CNTT • <i>Khoa</i> là CN&ANTT 	<ul style="list-style-type: none"> • <i>Đăng nhập</i> • <i>Xem thông tin độc giả</i> • <i>Tìm kiếm tài liệu</i> • <i>Xem tài liệu</i> • <i>Mượn tài liệu</i> • <i>Trả tài liệu</i>
Độc giả giáo viên	<ul style="list-style-type: none"> • <i>Mã số</i> là GV0001 • <i>Mật khẩu</i> là GV0001 • <i>Tên</i> là “Lê Thị B” • <i>Ngày sinh</i> 23/03/1983 • <i>Giới tính</i> là Nữ • <i>Quê quán</i> là Hà Nội • <i>Đơn vị</i> là khoa CN&ANTT • <i>Cấp hàm</i> là Thượng úy 	<ul style="list-style-type: none"> • <i>Đăng nhập</i> • <i>Xem thông tin độc giả</i> • <i>Tìm kiếm tài liệu</i> • <i>Xem tài liệu</i> • <i>Mượn tài liệu</i> • <i>Trả tài liệu</i>

- Trùu tượng hóa đối tượng theo dữ liệu

Trùu tượng hóa đối tượng theo dữ liệu chính là quá trình mô hình hóa các thuộc tính của lớp dựa trên trạng thái dữ liệu của các đối tượng tương ứng. Quá trình trùu tượng hóa đối tượng theo dữ liệu được tiến hành như sau:

- + Tập hợp tất cả các thuộc tính có thể có của các đối tượng.
- + Nhóm các đối tượng có các thuộc tính tương tự nhau, loại bỏ bớt các thuộc tính cá biệt, tạo thành một nhóm chung.
- + Mỗi nhóm đối tượng đề xuất một lớp tương ứng.
- + Các thuộc tính chung của nhóm đối tượng sẽ cấu thành các thuộc tính tương ứng của lớp được đề xuất.

- Trùu tượng hóa đối tượng theo chức năng

Trùu tượng hóa đối tượng theo chức năng là quá trình mô hình hóa phương thức của lớp dựa trên các hành động của các đối tượng. Quá trình trùu tượng hóa đối tượng theo chức năng được tiến hành như sau:

- + Tập hợp tất cả các hành động có thể có của các đối tượng.
- + Nhóm các đối tượng có các hoạt động tương tự nhau, loại bỏ bớt các hoạt động cá biệt, tạo thành một nhóm chung.
- + Mỗi nhóm đối tượng đề xuất một lớp tương ứng.
- + Các hành động chung của nhóm đối tượng sẽ cấu thành các phương thức của lớp tương ứng.

Trong **Error! Reference source not found.**, ta có thể thực hiện hai mức trùu tượng hóa như sau:

Mức 1: Nhóm các đối tượng vào một nhóm gọi là nhóm “*Độc Giả*”.

Lớp Độc Giả	Lớp Độc Giả
Thuộc tính	Phương thức
Mã số	Đăng nhập
Mật khẩu	Xem thông tin độc giả
Tên	Tìm kiếm tài liệu
Ngày sinh	Xem tài liệu
Giới tính	Mượn tài liệu
Quê quán	Trả tài liệu

Mức 2: Nhóm các đối tượng vào nhóm “*Sinh Viên*” và nhóm “*Giáo Viên*”

Khi đó, kết quả trùu tượng hóa các đối tượng theo dữ liệu sẽ như sau:

Lớp Sinh Viên	Lớp Giáo Viên
Thuộc tính	Thuộc tính
Mã số	Mã số
Mật khẩu	Mật khẩu

Tên	Tên
Ngày sinh	Ngày sinh
Giới tính	Giới tính
Quê quán	Quê quán
Lớp	Đơn vị
Khoa	Cấp hàm

Kết quả trừu tượng hóa các đối tượng theo chức năng như sau:

Lớp Sinh Viên	Lớp Giáo Viên
Phương thức	Thuộc tính
Đăng nhập	Đăng nhập
Xem thông tin độc giả	Xem thông tin độc giả
Tìm kiếm tài liệu	Tìm kiếm tài liệu
Xem tài liệu	Xem tài liệu
Mượn tài liệu	Mượn tài liệu
Trả tài liệu	Trả tài liệu

Lưu ý:

Các phương thức của hai lớp Sinh viên và Giáo viên tuy có cùng tên, nhưng việc thực thi các phương thức này sẽ có những điểm khác nhau. Vấn đề này sẽ được lý giải và thảo luận kỹ hơn ở phần đa hình.

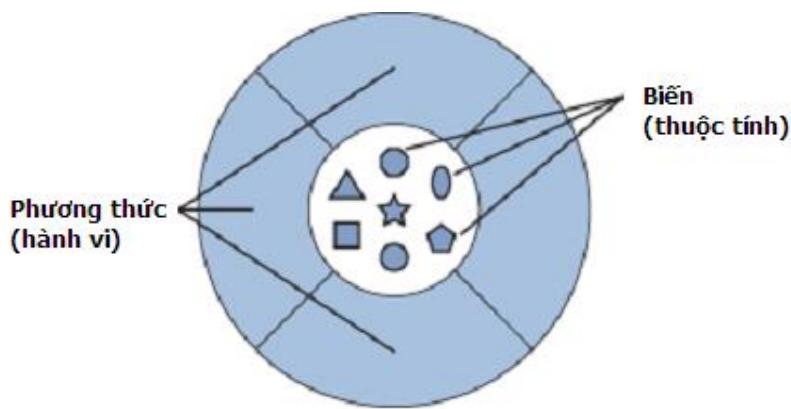
Như vậy, trừu tượng hóa là cần thiết, bởi vì không thể mô tả tất cả các hành động và các thuộc tính của một thực thể. Vấn đề mấu chốt là tập trung vào vấn đề cần quan tâm trong ứng dụng, xác định những đặc tính thiết yếu và những hành động cần thiết, giảm thiểu những chi tiết không cần thiết.

Trong lập trình hướng đối tượng, trừu tượng hóa các đối tượng theo dữ liệu và theo chức năng là bước đầu tiên trong quá trình hình thành nên kiểu dữ liệu trừu tượng lớp.

2. Đóng gói dữ liệu vào lớp

Sau khi trừu tượng hóa các đối tượng theo dữ liệu và theo chức năng, để tạo nên các thuộc tính và phương thức, chúng ta tiến hành đóng gói các thuộc tính và

phương thức này vào một lớp. Đóng gói trong lập trình hướng đối tượng mô phỏng đóng gói trong thế giới thực, nhằm mục đích che giấu chi tiết về cài đặt của đối tượng khỏi thế giới bên ngoài. Chi tiết về các thuộc tính của một đối tượng cần được đóng gói bên trong đối tượng để từ đó truy nhập đến chúng qua các hành vi của đối tượng. Tương tự, chi tiết về cách thực hiện các hành vi của một đối tượng cũng nên được đóng gói để bên ngoài không biết chi tiết. Vì vậy, đóng gói trong lập trình hướng đối tượng bao gồm: đóng gói chi tiết về cách thức lưu trữ các thuộc tính của đối tượng và đóng gói chi tiết về cách thực hiện các hành vi của đối tượng.



Hình 2.1. Đóng gói thuộc tính và hành vi của đối tượng

Mô hình đối tượng ở trên chỉ ra rằng các biến (thuộc tính) của đối tượng nằm ở trung tâm (hạt nhân) của đối tượng. Các phương thức (hành vi) bao quanh và che giấu hạt nhân của đối tượng. Việc đóng gói các thuộc tính và phương thức vào một đơn vị cấu trúc (gọi là lớp) và việc bao bọc các thuộc tính trong vòng bảo vệ của các phương thức chính là nguyên tắc đóng gói thông tin (*encapsulation*) trong lập trình hướng đối tượng. Tuy nhiên, khi hiện thực, tùy yêu cầu từng ứng dụng mà mức độ bảo vệ dữ liệu là khác nhau, công khai hoặc bí mật đối với bên ngoài. Điều này sẽ được thảo luận sâu hơn trong phần kiểm soát truy nhập.

Đóng gói thông tin có hai lợi ích chính:

- Mô-đun hóa: Mã nguồn của một lớp có thể được viết lại và bảo trì độc lập với mã nguồn của các lớp khác nên đơn giản hóa lập trình (miễn là giao diện của lớp với bên ngoài không đổi).

- Che giấu thông tin: Giá trị của các thuộc tính và chi tiết cài đặt bên trong của phương thức có thể được che giấu nhờ sự kết hợp của đóng gói và kiểm soát truy nhập.

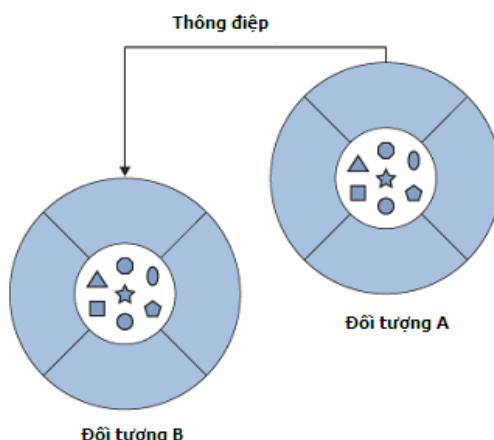
Lớp Độc Giả
Thuộc tính
Mã số
Mật khẩu
Tên
Ngày sinh
Giới tính
Quê quán
Phương thức
Đăng nhập
Xem thông tin độc giả
Tìm kiếm tài liệu
Xem tài liệu
Mượn trả tài liệu

Như vậy, ta có thể trùu tượng hóa mức 1 các đối tượng độc giả cụ thể theo dữ liệu và theo chức năng để xây dựng nên các thuộc tính và phương thức chung, sau đó đóng gói các thuộc tính và phương thức này vào lớp *Độc giả*.

Một chương trình hay ứng dụng lớn khi thực thi thường tạo nhiều đối tượng khác nhau. Các đối tượng tương tác và giao tiếp với nhau bằng cách gửi các thông điệp (*message*). Khi đối tượng A muốn đối tượng B thực hiện các phương thức của đối tượng B thì đối tượng A gửi một thông điệp tới đối tượng B để yêu cầu

đối tượng B thực hiện phương thức của nó (Minh họa ở Hình 2.2). Một thông điệp (*message*) là một lời yêu cầu thực hiện phương thức. Một thông điệp được truyền khi một đối tượng triệu gọi một hay nhiều phương thức của đối tượng khác. Một thông điệp gồm có: đối tượng nhận thông điệp, tên của phương thức thực hiện, các tham số mà phương thức cần. Khi một đối tượng nhận được một thông điệp sẽ thực hiện một phương thức tương ứng của nó. Phương thức còn được gọi là “hộp đen”, vì để sử dụng phương thức, cũng giống như để sử dụng hộp đen, không cần quan tâm thực thi phức tạp bên trong phương thức mà chỉ cần quan tâm đến giao diện của phương thức đó. Trong lập trình hướng đối tượng, lớp cũng được coi là “hộp đen”, vì lớp có phần giao diện công khai và phần thực thi bên trong bí mật.

Như vậy, thông tin đã được đóng gói vào lớp. Việc truy nhập dữ liệu về cơ bản không được thực hiện trực tiếp mà cần thông qua các phương của lớp. Khi có thay đổi trong dữ liệu của đối tượng, ta chỉ cần thay đổi các phương thức truy nhập thuộc tính của lớp, mà không cần phải thay đổi mã nguồn của các chương trình sử dụng lớp tương ứng.



Hình 2.2. Đối tượng A gửi thông điệp tới đối tượng B.

II. KHAI BÁO LỚP, TẠO VÀ SỬ DỤNG ĐỐI TƯỢNG

1. Khai báo lớp và tạo đối tượng

Lớp gồm các thuộc tính và các phương thức tác động lên thành phần thuộc tính đó. Đối tượng được xây dựng bởi lớp nên được gọi là các thể hiện của lớp (*class instance*). Lớp trong Java là sự mở rộng khái niệm bản ghi (record) trong ngôn ngữ lập trình Pascal, hay cấu trúc (*struct*) trong ngôn ngữ lập trình C. Lớp được coi là một kiểu dữ liệu. Sau khi khai báo lớp (định nghĩa kiểu dữ liệu), ta có thể tạo các đối tượng của lớp.

Lớp nên được xây dựng từ phạm vi bài toán cần giải quyết, vì vậy tên của lớp cũng nên đặt trùng tên các đối tượng thực được biểu diễn.

Ví dụ 2.2:

Trong bài toán quản lý sinh viên có yêu cầu thống kê cuối năm học cần lập danh sách khen thưởng cho các sinh viên xuất sắc (có điểm học trung bình từ 8.0 trở lên, điểm rèn luyện từ 9.0 trở lên) đồng thời phải có thành tích nghiên cứu khoa học từ cấp Học viện trở lên. Giả sử có danh sách sinh viên như sau:

Mã SV	Họ tên	Giới tính	Quê quán	Lớp	Khoa	Điểm học TB	Điểm rèn luyện	Nghiên cứu khoa học
SV0001	Trần Văn A	Nam	Thái Bình	D46 Tin	CN& ANTT	8,2	10	Giải nhì nghiên cứu khoa học cấp Học viện
SV0002	Phan Thị B	Nữ	Hải Phòng	D46 AV	Ngoại ngữ	7,0	9	
...

Với danh sách này, khi lập trình theo hướng đối tượng ta quan niệm mỗi sinh viên là một đối tượng. Căn cứ yêu cầu ứng dụng, thực hiện trừu tượng hóa đối tượng theo dữ liệu và theo chức năng, sau đó tiến hành đóng gói thông tin, ta được lớp Sinh Viên như sau:

Lớp Sinh Viên
Thuộc tính
Mã sinh viên
Họ tên
Giới tính
Quê quán
Lớp
Khoa
Điểm học TB
Điểm rèn luyện
Nghiên cứu khoa học
Phương thức
Xem thông tin
Được khen thưởng

Trong đó *Mã sinh viên*, *Họ tên*, *Giới tính*, *Quê quán*, *Lớp*, *Khoa*, *Điểm học TB*, *Điểm rèn luyện*, *Nghiên cứu khoa học* trở thành thuộc tính của lớp. Còn *Xem thông tin*, *Được khen thưởng* là các phương thức của lớp. Sử dụng ngôn ngữ mô hình hóa và sơ đồ, ta có thể mô hình hóa lớp Sinh viên trên như sau:

Lớp Sinh Viên
- maSV: String
- hoTen: String
- gioiTinh: boolean
- queQuan: String
- lop: String
- khoa: String
- diemHocTB: float
- diemRenLuyen: float
- nghienCuuKhoaHoc: String
+ xemThongTin(): void

```
+duocKhenThuong(): boolean
```

Khi đã mô hình hóa được lớp cần tạo với các thuộc tính và phương thức, ta sẽ xây dựng lớp sử dụng ngôn ngữ Java.

a) Khai báo lớp

Cú pháp khai báo một lớp trong Java như sau:

```
[<Chỉ thị truy nhập>] [<Tính chất>] class <Tên lớp>  
[extends <Tên lớp cha>] [implements <Tên giao diện>] {  
    <Các thành phần của lớp>  
}
```

Trong đó **class**, **extends**, **implements** là các từ khóa. Những phần trong cặp [] là tùy chọn (có thể có hoặc không) bao gồm các nội dung thuộc phần kiểm soát truy nhập, thừa kế và giao diện, sẽ được đề cập chi tiết ở các phần sau.

Trước tiên chúng ta tìm hiểu cú pháp định nghĩa cơ bản không bao gồm các tùy chọn trên:

```
class <Tên lớp>{  
    <Các thành phần của lớp>  
}
```

- <Tên lớp>: Tên của lớp phải được đặt theo quy ước định danh trong Java, ví dụ SinhVien, GiaoVien.

- <Các thành phần của lớp>: bao gồm các *biến*, các *phương thức* thành phần và các *phương thức khởi tạo* (*constructor*). Thân của một lớp có thể chứa các khai báo của các lớp, giao diện (*interface*) khác và xem chúng như là các thành phần của lớp. Các thành phần của lớp được khai báo như sau:

+ Khai báo thuộc tính

Cú pháp khai báo thuộc tính trong Java như sau:

```
[<Chỉ thị truy nhập>] [<Tính chất>] kieuThuocTinh tenThuocTinh;
```

Trong đó, các phần trong [] là tùy chọn, có thể có hoặc không (sẽ được trình bày cụ thể ở các phần sau), kieuThuocTinh là kiểu dữ liệu của thuộc tính, tenThuocTinh là tên của thuộc tính đặt theo quy tắc đặt định danh.

Ví dụ 2.3: Khai báo các thuộc tính trong lớp SinhVien:

```
String maSV;
String hoTen;
Date ngaySinh;
boolean gioiTinh;
String queQuan;
String lop;
String khoa;
```

+ Khai báo phương thức khởi tạo

Khi khai báo biến thuộc các kiểu dữ liệu trong Java, giá trị chứa trong biến đó thường là không dự đoán trước được. Các biến cần được đảm bảo đã được khởi tạo giá trị trước khi sử dụng. Tương tự đối với các lớp trong Java, các thành phần dữ liệu (thuộc tính) của một thể hiện (đối tượng) cần được khởi tạo giá trị cho trước khi bắt đầu sử dụng các phương thức của đối tượng đó. Phương thức khởi tạo là một loại phương thức đặc biệt dùng để khởi tạo thể hiện của lớp. Mỗi khi một thể hiện của lớp được tạo, một hàm khởi tạo của lớp sẽ được gọi.

Cú pháp khai báo phương thức khởi tạo

```
[<Chỉ thị truy nhập>]<Tên lớp>([<Ds tham biến hình thức>]) {
    <Nội dung cần tạo lập>
}
```

Trong đó, các phần trong [] là tùy chọn sẽ được trình bày chi tiết sau.

- * <Tên lớp> luôn trùng với tên của lớp chứa phương thức khởi tạo.
- * <Nội dung cần tạo lập>: Phần này có thể bao gồm khối lệnh hoặc không. Nội dung cần tạo lập bao gồm khởi gán giá trị cho các thuộc tính của đối tượng và có thể thực hiện một số công việc cần thiết khác chuẩn bị cho đối tượng mới.

Lưu ý:

- * Phương thức khởi tạo có tên trùng với tên của lớp.

- * Phương thức khởi tạo không có kiểu trả lại, kiểu void cũng không được sử dụng.
- * Phương thức khởi tạo chỉ sử dụng được với toán tử *new*
- * Phương thức khởi tạo có thể có hoặc không có tham số như các phương thức thông thường khác

Trong một lớp có thể có nhiều phương thức khởi tạo.

+ Phương thức khởi tạo mặc định

Khi xây dựng một lớp mà không xây dựng phương thức khởi tạo thì Java sẽ cung cấp cho ta một phương thức khởi tạo không có tham số mặc định. Phương thức khởi tạo này thực chất không làm gì.

```
<Tên lớp> () { }
```

Ví dụ phương thức khởi tạo mặc định của lớp SinhVien

```
SinhVien () { };
```

Khi sử dụng phương thức khởi tạo với toán tử *new* để tạo đối tượng thì các biến kiểu nguyên thủy như *masv*, *hoTen*, *gioiTinh*, ... sẽ được khởi tạo các giá trị mặc định tương ứng là *null*, *null*, *false*, ...

Lưu ý:

Nếu trong lớp đã có ít nhất một phương thức khởi tạo thì phương thức khởi tạo mặc định sẽ không được tạo ra. Khi ta tạo ra một đối tượng thì sẽ có một phương thức khởi tạo nào đó được gọi. Nếu trình biên dịch không tìm thấy phương thức khởi tạo tương ứng sẽ thông báo lỗi.

Bên cạnh phương thức khởi tạo mặc định do Java cung cấp, ta có thể tạo thêm nhiều phương thức khởi tạo khác nhau tùy theo danh sách tham số. Java phân biệt các phương thức khởi tạo dựa trên số lượng và loại tham số trong danh sách tham số.

Ví dụ 2.4: Phương thức khởi tạo Sinhvien

```

    SinhVien(String maSV, String hoTen, boolean gioiTinh, String
queQuan, String lop, String khoa, float diemTB, float diemRL,
String nckh) {
    this.maSV = maSV;
    this.hoTen = hoTen;
    this.gioiTinh = gioiTinh;
    this.queQuan = queQuan;
    this.lop = lop;
    this.khoa = khoa;
    this.diemTB = diemTB;
    this.diemRL = diemRL;
    this.nckh = nckh;
}

```

+ Khai báo phương thức

Các đối tượng trong chương trình Java trao đổi với nhau bằng các thông điệp. 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.

Cú pháp khai báo phương thức

```

[<Chỉ thị truy nhập>] [<Tính chất>] kieuHam tenHam([<Ds tham biến
hình thức>]);

```

Trong đó, các phần trong [] là tùy chọn có thể có hoặc không (sẽ được trình bày cụ thể ở các phần sau), kieuHam là kiểu dữ liệu mà giá trị của nó được phương thức trả về, có thể là kiểu nguyên thủy hoặc kiểu tham chiếu. Nếu không có một giá trị nào được trả về, kiểu dữ liệu có thể là void, tenHam là tên của biến. <Ds tham biến hình thức> chứa tên của các tham số được sử dụng trong phương thức và kiểu dữ liệu tương ứng với từng tham số. Dấu phẩy được dùng để phân cách các tham số.

Ví dụ 2.5:

```

void xemThongtin(){
    System.out.println("Mã sinh viên: " + maSV);
    System.out.println("Họ tên: " + hoTen);
    if(gioiTinh)
        System.out.println("Giới tính nam");
    else
        System.out.println("Giới tính nữ");
    System.out.println("Quê quán: " + queQuan);
    System.out.println("Lớp: " + lop);
    System.out.println("Khoa: " + khoa);
}

```

```

        System.out.println("Điểm học trung bình: " + diemTB);
        System.out.println("Điểm rèn luyện: " + diemRL);
        System.out.println("Nghiên cứu khoa học: " + nckh);
    }
    boolean duocKhenThuong(){
        if(diemTB >= 8.0 && diemRL >= 9.0 && !nckh.endsWith(""))
            return true;
        return false;
    }
}

```

Khai báo lớp SinhVien sử dụng cú pháp định nghĩa lớp đóng gói các thuộc tính và phương thức đã khai báo ở trên.

Ví dụ 2.6: Thêm phương thức vào lớp SinhVien

```

class SinhVien {
    String maSV;
    String hoTen;
    boolean gioiTinh;
    String queQuan;
    String lop;
    String khoa;
    float diemTB;
    float diemRL;
    String nckh;
    SinhVien();
    SinhVien(String maSV, String hoTen, boolean gioiTinh, String
queQuan, String lop, String khoa, float diemTB, float diemRL,
String nckh) {
        this.maSV = maSV;
        this.hoTen = hoTen;
        this.ngaySinh = ngaySinh;
        this.gioiTinh = gioiTinh;
        this.queQuan = queQuan;
        this.lop = lop;
        this.khoa = khoa;
        this.diemTB = diemTB;
        this.diemRL = diemRL;
        this.nckh = nckh;
    }
    void xemThongtin(){
        System.out.println("Mã sinh viên: " + maSV);
        System.out.println("Họ tên: " + hoTen);
        System.out.println("Ngày sinh: " + ngaySinh.toString());
        if(gioiTinh)
            System.out.println("Giới tính nam");
        else
            System.out.println("Giới tính nữ");
        System.out.println("Quê quán: " + queQuan);
        System.out.println("Lớp: " + lop);
        System.out.println("Khoa: " + khoa);
        System.out.println("Điểm học trung bình: " + diemTB);
        System.out.println("Điểm rèn luyện: " + diemRL);
        System.out.println("Nghiên cứu khoa học: " + nckh);
    }
}

```

```

    }
    boolean duocKhenThuong(){
        if(diemTB >= 8.0 && diemRL >= 9.0 && !nckh.endsWith(""))
            return true;
        return false
    }
}

```

Ngoài cú pháp khai báo lớp cơ bản như trên, Java cho phép khai báo một lớp trong một lớp khác. Lớp như vậy được gọi là *nested class* và được khai báo:

```

class LopNgoai{
    ...
    class LopTrong{
        ...
    }
}

```

Các lớp nested class được chia thành hai nhóm: static và non-static. Các lớp nested class không được khai báo với từ khóa static còn được gọi là lớp inner class (lớp trong) phân biệt với outer class (lớp ngoài). Các lớp nested class được khai báo với từ khóa static còn được gọi là lớp static nested class.

b) Tạo đối tượng

Khai báo lớp cung cấp khuôn mẫu để tạo đối tượng. Ta sử dụng lớp để tạo đối tượng. Tạo một đối tượng đồng nghĩa với tạo một thể hiện của một lớp.

Cú pháp tạo đối tượng:

```
tenLop tenBien = new tenLop([<danh sách tham biến hiện thời>]);
```

Mỗi câu lệnh tạo đối tượng gồm có 3 phần:

- Khai báo biến tham chiếu tới một đối tượng

```
type name;
```

Với biến kiểu nguyên thủy, khai báo này phân bổ chính xác vùng nhớ cho biến. Với biến kiểu tham chiếu nếu khai báo như vậy biến sẽ không tham chiếu tới một đối tượng nào. Nói cách khác, khai báo này sẽ không xác định được giá trị của biến chừng nào đối tượng chưa được thực sự tạo ra và khởi gán. Vì vậy, ta cần sử dụng toán tử *new* và khởi gán một đối tượng cho biến tham chiếu trước khi

sử dụng nó, nếu không trình biên dịch sẽ báo lỗi. Toán tử *new* thể hiện hóa một lớp bằng cách phân bổ vùng nhớ cho một đối tượng mới và trả về một tham chiếu tới vùng nhớ đó. Toán tử *new* gọi tới hàm khởi tạo đối tượng.

- Khởi tạo một đối tượng

Java sử dụng các phương thức khởi tạo để khởi tạo đối tượng mới.

Ví dụ 2.7:

```
//tao doi tuong
SinhVien sv1 = new SinhVien();
SinhVien sv2;
sv2 = new SinhVien("SV0001", "Trần Văn A", true, "Thái Bình",
"D41Tin", "Toán Tin", 8.2f, 10.0f, "Giải nài nghiên cứu khoa học
cấp Học viện");
```

2. Sử dụng đối tượng

a) Tham chiếu

Sau khi tạo đối tượng, ta có thể sử dụng đối tượng với các thao tác như lấy giá trị hoặc thay đổi giá trị các thuộc tính của đối tượng, hay gọi một trong số các phương thức của đối tượng.

Giả sử đối tượng A muốn truy nhập tới các thuộc tính và phương thức của đối tượng B. Khi đó, đối tượng A cần thực hiện các câu lệnh tham chiếu sau:

- Tham chiếu các thuộc tính của một đối tượng

```
<Tham chiếu đối tượng>.<Tên thuộc tính>([<danh sách tham biến
hiện thời>]);
```

Các thuộc tính của đối tượng được truy nhập bởi tên. Mã lệnh nằm ngoài lớp phải sử dụng một tham chiếu đối tượng hoặc một biểu thức, theo sau bởi dấu chấm (.) và tên trường.

Ví dụ 2.8:

```
//tham chieu thuoc tinh doi tuong sv2
System.out.println("Mã sinh viên: " + sv2.maSV);
```

```
System.out.println("Họ tên: " + sv2.hoTen);  
System.out.println("Lớp: " + sv2.lop);
```

Mã tham chiếu thuộc tính đối tượng sv2 nằm ngoài lớp SinhVien. Vì vậy, để liên hệ tới các thuộc tính maSV, hoTen, lop ... trong đối tượng sinhVien1 thuộc lớp SinhVien phải sử dụng sinhVien1.maSV, sinhVien1.hoTen, sinhVien1.lop ... tương ứng.

Ta cũng có thể truy nhập các thuộc tính bằng cách sử dụng tham chiếu trả về từ toán tử *new* để truy nhập các trường của một đối tượng mới.

```
String hoTen = new SinhVien().hoTen;
```

Câu lệnh trên truy nhập giá trị mặc định của thuộc tính hoTen. Tuy nhiên cần lưu ý, sau khi thực thi câu lệnh trên, chương trình sẽ không còn tham chiếu tới đối tượng kiểu SinhVien vừa được tạo, vì chương trình không lưu tham chiếu này.

- Tham chiếu các phương thức của một đối tượng

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

Ví dụ:

```
sv2.xemThongtin();
```

Việc gọi các phương thức của một đối tượng thực hiện tương tự như cách truy cập các thuộc tính của đối tượng. Tuy nhiên cần lưu ý vấn đề truyền tham số trong lời gọi hàm.

Tuy nhiên, các câu lệnh tham chiếu tới các thuộc tính và phương thức trên có thực hiện được hay không còn phụ thuộc vào quyền truy cập của đối tượng A tới thuộc tính và phương thức đó của đối tượng B. Đặc trưng cơ bản của lập trình hướng đối tượng là tính đóng gói và che giấu thông tin, nghĩa là có cơ chế kiểm soát để đối tượng A được phép hay không được phép truy nhập (tham chiếu) tới các thuộc tính và phương thức của đối tượng B. Phản tiếp theo sẽ phân tích rõ hơn về cơ chế này của lập trình hướng đối tượng với minh họa trên ngôn ngữ Java.

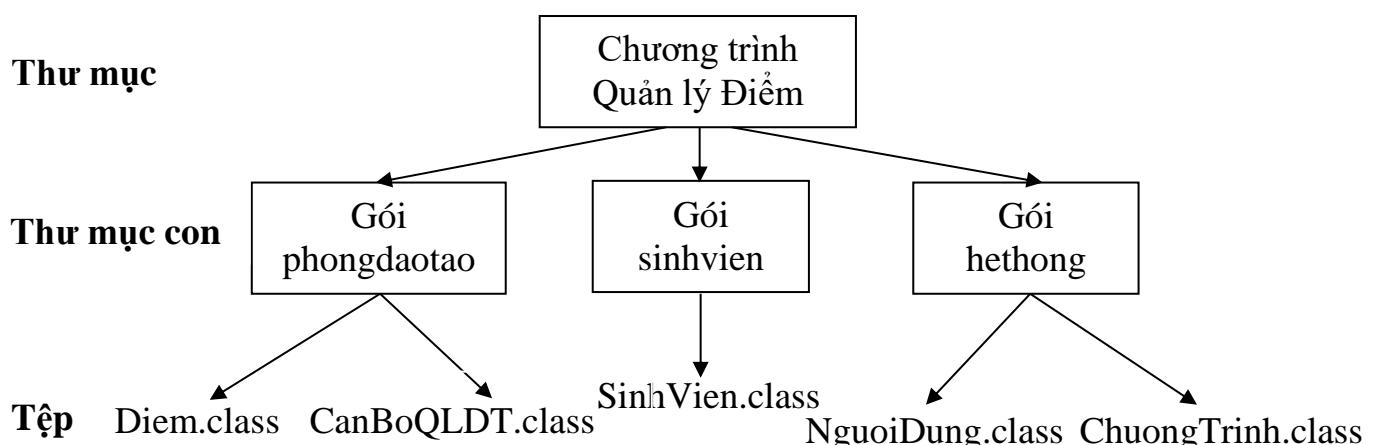
- Tham chiếu đối tượng hiện thời

Trong một phương thức thể hiện hoặc một toán tử tạo lập, this là một tham chiếu tới đối tượng hiện thời - đối tượng mà phương thức hoặc toán tử tạo lập của nó được gọi. Ta có thể sử dụng this với thuộc tính hoặc với một toán tử tạo lập.

b) Kiểm soát truy nhập

Java quản lý lớp theo gói, đồng thời cung cấp các chỉ thị truy nhập (access modifiers) trước định nghĩa lớp và các thành phần lớp, bao gồm: *public*, *protected*, *private*. Các mức độ kiểm soát truy nhập từ “truy nhập nhiều nhất” đến “truy nhập ít nhất” là: công khai (*public*), được bảo vệ (*protected*), truy nhập gói (không sử dụng chỉ thị truy nhập), và riêng tư (*private*). Trong phần này chúng ta sẽ tìm hiểu việc điều khiển truy nhập tới các lớp và các thành phần (thuộc tính, phương thức) của lớp trong Java.

Chúng ta sử dụng một số ký hiệu của UML để minh họa. UML ký hiệu ‘+’ cho truy nhập *public*, ‘#’ cho truy nhập *protected*, ‘~’ cho truy nhập mặc định và ‘-’ cho truy nhập *private*. Xét cấu trúc tổng quát của một chương trình Java thông qua một chương trình Quản lý điểm đơn giản (Hình 2.3).



Hình 2.3. Cấu trúc tổ chức chương trình Quản lý điểm.

Chương trình này có các đặc điểm như sau:

- Đối tượng sử dụng chương trình: cán bộ phòng quản lý đào tạo, sinh viên.

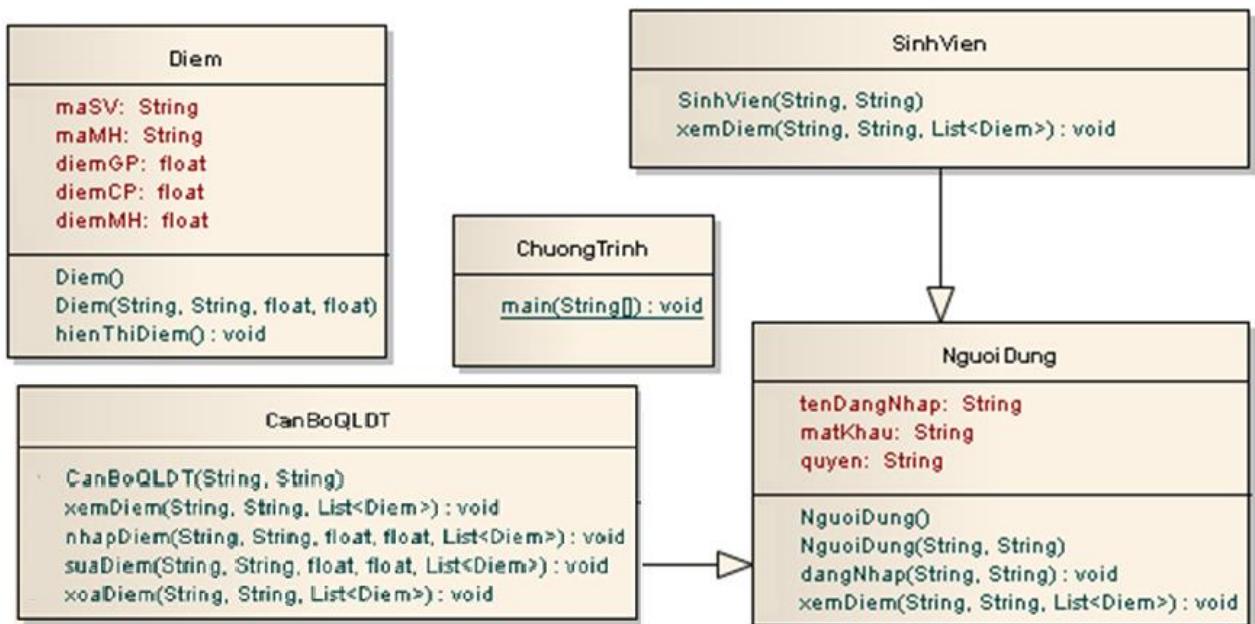
- Chương trình quản lý điểm giữa phần, điểm cuối phần và điểm tổng hợp theo môn học của sinh viên.

- Chương trình chỉ cho phép sinh viên xem điểm môn học.

- Chương trình cho phép cán bộ phòng quản lý đào tạo thực hiện các thao tác xem và cập nhập điểm giữa phần và điểm cuối phần theo môn học của sinh viên.

- Chương trình tính điểm tổng hợp môn học của sinh viên theo công thức tỉ lệ điểm giữa phần và điểm cuối phần của từng môn học.

Với chương trình trên, ta có thể thiết kế các lớp trong chương trình như sau:



Hình 2.4. Các lớp trong chương trình Quản lý điểm.

Đến đây ta mới chỉ xây dựng được các lớp riêng lẻ mà chưa quản lý được các lớp này trong mối quan hệ với nhau và với ứng dụng. Để quản lý các lớp theo cấu trúc chương trình như trên, chúng ta cần thực hiện các bước sau:

- Tạo và sử dụng gói

Gói là một nhóm các lớp, giao diện có quan hệ với nhau nhằm cung cấp sự bảo vệ truy nhập và quản lý không gian tên. Bảo vệ truy nhập cho phép các lớp, giao diện trong cùng gói có được truy nhập không hạn chế tới nhau trong khi vẫn

hạn chế được truy nhập tới các lớp và giao diện bên ngoài gói. Quản lý không gian tên giúp tổ chức các lớp và giao diện một cách logic và chặt chẽ theo chức năng và mối quan hệ giữa chúng, từ đó tránh đụng độ tên với lớp, giao diện thuộc gói khác và giúp dễ dàng tìm kiếm và sử dụng.

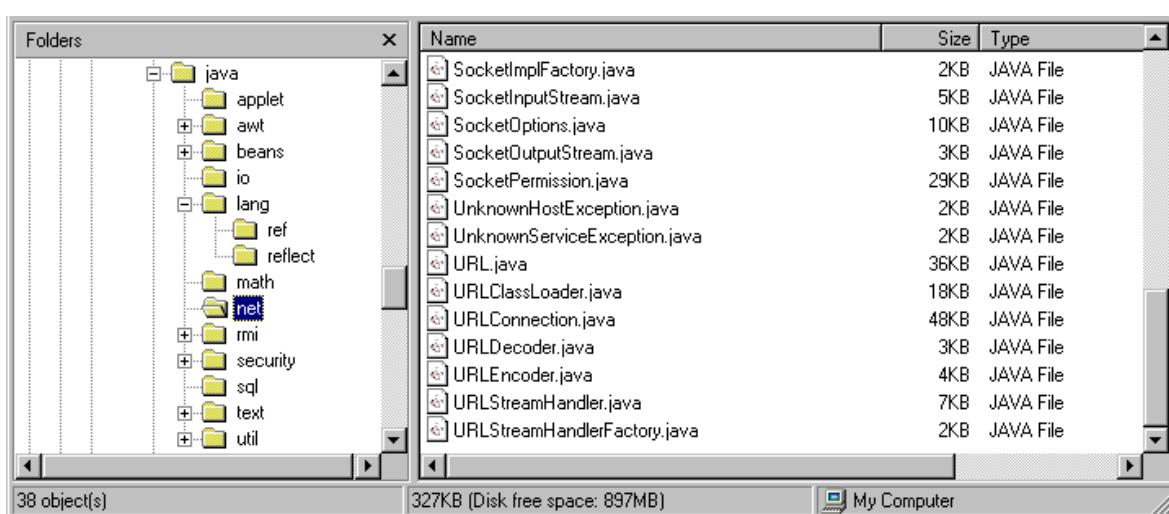
Java có 2 loại gói: Gói API chuẩn (*Standard API package*) và gói người dùng định nghĩa (*user-defined package*). Bên cạnh việc sử dụng thư viện các gói API chuẩn JDK có sẵn, Java cung cấp cú pháp cho phép lập trình viên tự tạo các gói thư viện của riêng mình gọi là các gói người dùng định nghĩa (*user-defined package*). Câu lệnh tạo gói như sau:

```
package <packageName>;
```

Câu lệnh này được đặt tại dòng lệnh đầu tiên của mỗi file chứa các lớp, giao diện ta muốn đưa vào gói. Chỉ có một câu lệnh gói trong mỗi file nguồn, nó áp dụng cho tất cả các lớp, giao diện trong file đó. Trong câu lệnh tạo gói trên, ta có:

+ package là từ khóa tạo gói

+ <packageName> là tên gói. Có thể đặt tên gói như tên thư mục trên ổ đĩa. Nghĩa là bắt đầu bằng tên có phạm vi lớn, cho đến các tên có phạm vi nhỏ, cuối cùng là tên các gói trực tiếp chứa các lớp, ngăn cách giữa các tên là dấu ‘.’.



Hình 2.5. Gói thư viện Java.

Khi không sử dụng câu lệnh tạo gói, mặc định các lớp và giao diện thuộc gói mặc định (*default*), tức gói thư mục hiện thời.

Ví dụ 2.9:

```
public class PackageTest{
    public static void test(){
        System.out.println("This is package test program");
    }
}
```

Sử dụng câu lệnh tạo gói, các lớp giao diện thuộc gói người dùng định nghĩa

Ví dụ 2.10:

```
package oop.chapter2;

public class PackageTest{
    public static void test(){
        System.out.println("This is package test program");
    }
}
```

Để sử dụng thành phần gói công khai từ bên ngoài gói đó, có thể thực hiện một trong ba cách sau:

+ Chỉ dẫn đến thành phần với tên chuẩn đầy đủ
<packageName>.<memberName>

```
public class PackageUsing{
    public static void main(String args) {
        oop.chapter2.PackageTest.test();
    }
}
```

+ Chèn thành phần gói : import <packageName>.<memberName>;

```
import oop.chapter2.PackageTest;
public class PackageUsing{
    public static void main(String[] args) {
        PackageTest.test();
    }
}
```

+ Chèn toàn bộ gói của thành phần đó : import <packageName>.*;

```
import oop.chapter2.*;
```

```
public class PackageUsing{
    public static void main(String[] args) {
        PackageTest.test();
    }
}
```

Lưu ý:

+ Để thuận tiện, trình biên dịch Java tự động import toàn bộ gói `java.lang` và gói mặc định (gói thư mục hiện thời) vào mỗi file nguồn.

+ Phân cấp các gói: Các gói xuất hiện có vẻ như phân cấp, nhưng không hẳn vậy. Ví dụ, Java API bao gồm một gói `java.awt`, một gói `java.awt.color`, một gói `java.awt.font`, và nhiều gói khác bắt đầu với `java.awt`. Tuy nhiên, gói `java.awt.color`, `java.awt.font` và những gói khác `java.awt.xxxx` không được bao gồm trong gói `java.awt`. Câu lệnh `import java.awt.*` sẽ import tất cả các “loại” trong gói `java.awt`, nhưng không import `java.awt.color`, `java.awt.font` hay bất kỳ gói `java.awt.xxxx` nào khác. Nếu muốn sử dụng các “loại” trong `java.awt.color` và `java.awt`, ta phải import toàn bộ cả hai gói.

```
import java.awt.*;
import java.awt.color.*;
```

+ Xung đột tên: Nếu một thành phần trong một gói có cùng tên với một thành phần trong gói khác cũng được import, ta cần chỉ dẫn tới mỗi thành phần bởi tên được chuẩn hóa của nó. Chẳng hạn, gói `oop.chapter2` định nghĩa một lớp tên `Date`. Gói `java.util` cũng chứa một lớp `Date`. Nếu cả hai gói `oop.chapter2` và `java.util` đều được import, câu lệnh `Date date;` sẽ dẫn đến xung đột. Với tình huống này, ta cần sử dụng tên chuẩn hóa đầy đủ của lớp `Date` với câu lệnh `oop.chapter2.Date date;`

- Chỉ thị truy nhập

Các chỉ thị truy nhập trong Java được đặt trước định nghĩa lớp và thành phần của lớp (thuộc tính, phương thức). Nếu không có chỉ thị truy nhập, nghĩa là “mức truy nhập gói”.

+ Truy nhập đối tượng

Mức truy nhập có thể đối với toàn bộ đối tượng là công khai (chỉ thị truy nhập “*public*”) hoặc mức gói (không dùng chỉ thị truy nhập). Để kiểm soát truy nhập tới toàn bộ đối tượng, chỉ thị truy nhập phải xuất hiện trước từ khóa class. Tuy nhiên, có một số ràng buộc như sau:

- * Chỉ có một lớp *public* trong một đơn vị biên dịch (một file). Ý tưởng là mỗi đơn vị biên dịch chỉ có một giao diện công khai, chính là lớp *public*.
- * Tên của lớp *public* phải trùng chính xác tên của file chứa đơn vị biên dịch, phân biệt chữ viết hoa.
- * Một đơn vị biên dịch có thể không cần có lớp *public*. Khi đó ta có thể đặt tên file bất kỳ.

Xét chương trình Quản lý Điểm, tại thời điểm chạy chương trình, ta có các đối tượng thuộc các lớp NguoiDung, SinhVien, CanBoQLDT đều có thể xem điểm. Cụ thể, trong các đối tượng thuộc 3 lớp này đều có phương thức *xemDiem()* trong đó có một tham số đầu vào là danh sách Diem. Điều này có nghĩa là các đối tượng thuộc những lớp này đều phải nhìn thấy đối tượng thuộc lớp Diem và có khả năng truy nhập được tới đối tượng thuộc lớp Diem. Trong ChuongTrinh (được coi là đầu vào - hay giao diện của ứng dụng), chúng ta cũng cần tham chiếu tới đối tượng thuộc lớp Diem này. Vậy đối tượng lớp Diem cần được nhìn thấy và được truy nhập trong toàn bộ hệ thống. Với yêu cầu đó, chúng ta sẽ khai báo lớp Diem với chỉ thị truy nhập lớp là *public*: `public class Diem`.

Nếu chúng ta chỉ khai báo `class Diem` mà không khai báo chỉ thị truy nhập lớp là *public* thì trình biên dịch sẽ báo lỗi ở các lớp nằm ngoài gói với lớp Diem, tức sẽ báo lỗi không thể truy nhập được tới lớp Diem ở các lớp SinhVien, NguoiDung và ChuongTrinh.

Bảng dưới đây mô tả khả năng truy nhập đối với đối tượng.

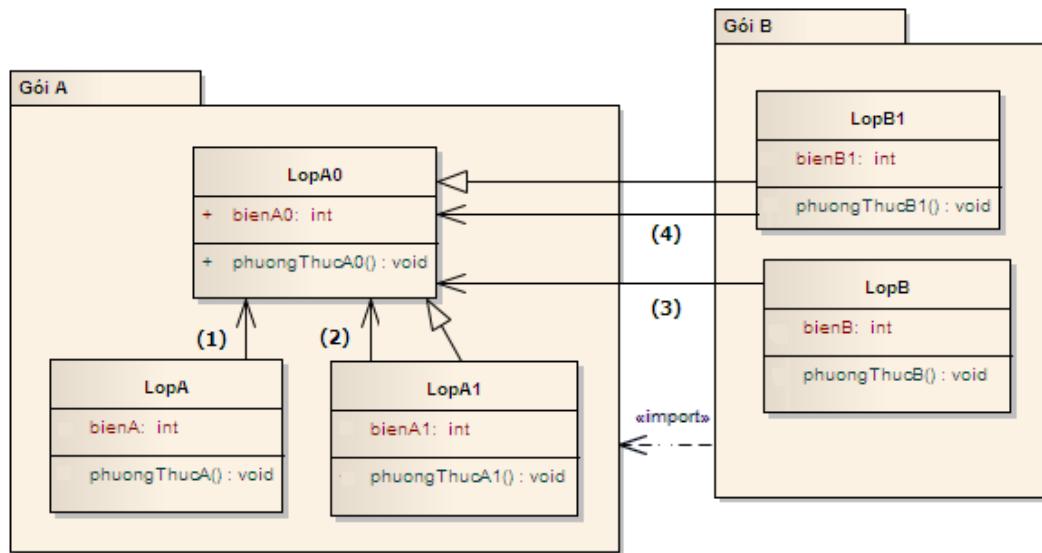
Chỉ thị truy nhập	Cùng gói	Khác gói
Public	+	+
no modifier	+	-

Bảng 2.1. Các mức truy nhập đối tượng.

Trong đó, ‘+’: có khả năng truy nhập; ‘-’: không có khả năng truy nhập

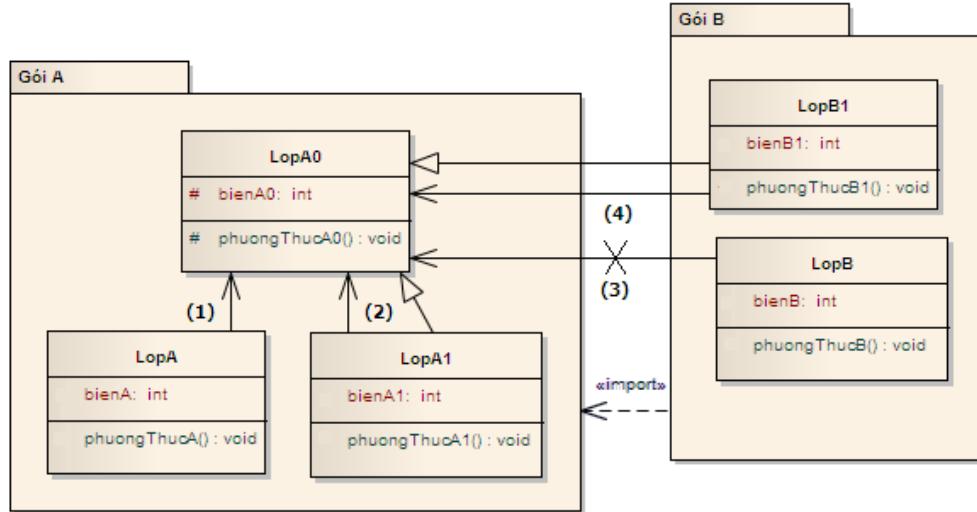
+ Truy nhập các thành phần của lớp :

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



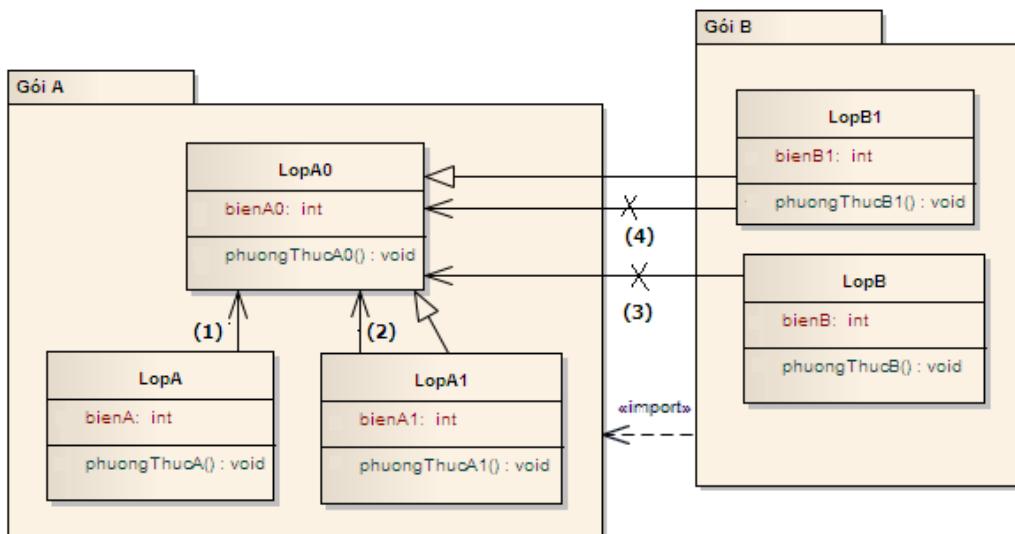
Hình 2.6. Minh họa truy cập các thành phần public.

+ Các thành phần được bảo vệ (*protected*): Những thành phần *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*.



Hình 2.7. Minh họa truy cập các thành phần protected.

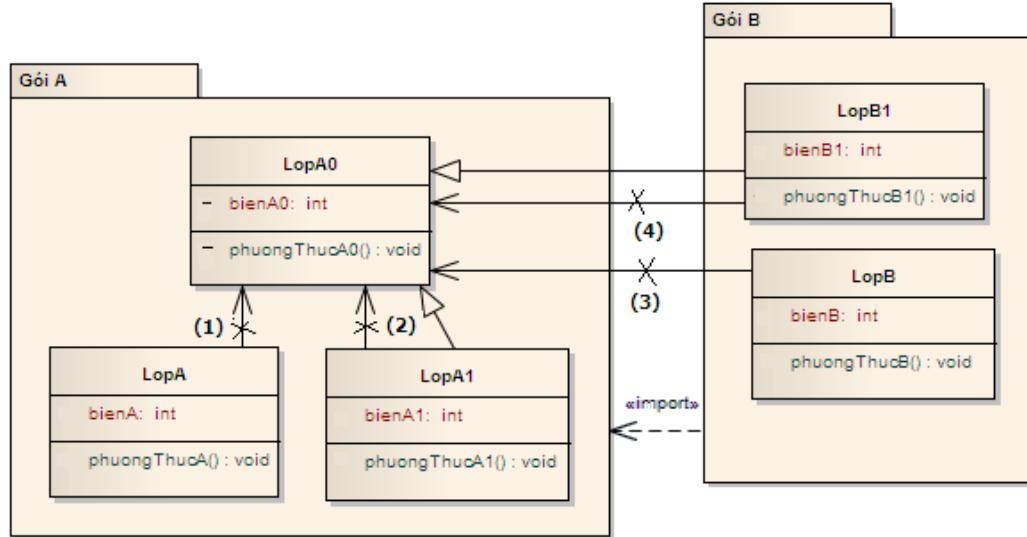
* Các thành phần mặc định (không sử dụng chỉ thị truy nhập): Truy nhập mặc định (không có từ khóa) là truy nhập gói, nghĩa là tất cả các lớp khác trong cùng gói đều có thể truy nhập tới thành phần có mức truy nhập gói nhưng tất cả các lớp bên ngoài gói này thì không thể truy nhập được. Các lớp trong các file khác nhau (không có định nghĩa tên gói) nhưng cùng trong một thư mục thì coi như có mức truy nhập gói mặc định, vì vậy, cung cấp truy nhập gói tới tất cả các file khác trong thư mục đó.



Hình 2.8. Minh họa truy cập các thành phần mặc định.

* Các thành phần sở hữu riêng (*private*): Những thành phần *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 kế thừa cũng không được phép truy nhập trực tiếp tới các thành phần *private*.



Hình 2.9. Minh họa truy cập các thành phần *private*.

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.

Chỉ thị truy nhập	Trong lớp	Lớp cùng gói	Lớp con khác gói	Lớp khác gói
Public	+	+	+	+
Protected	+	+	+	-
no modifiers	+	+	-	-
Private	+	-	-	-

Bảng 2.2. Các mức truy nhập thành phần đối tượng.

Dấu ‘+’ nghĩa là có khả năng truy nhập. Dấu ‘-’ nghĩa là không có khả năng truy nhập.

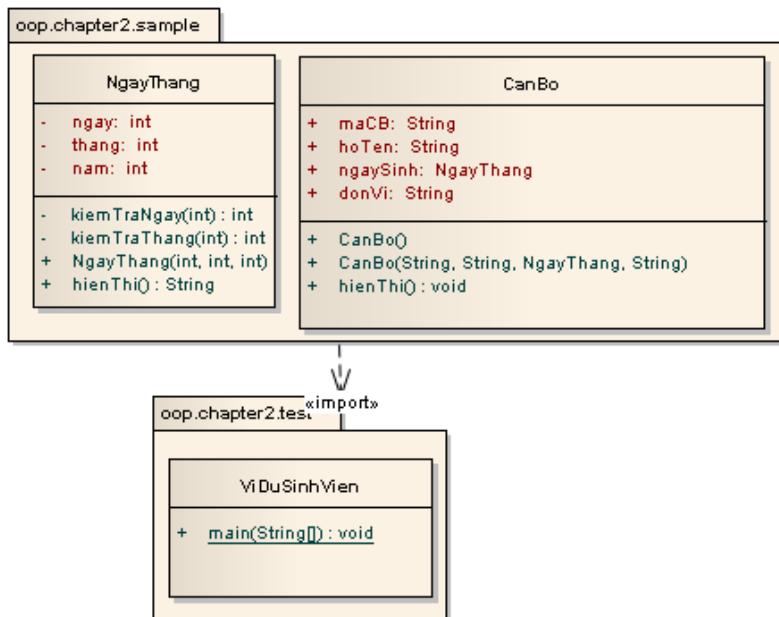
Trong Bảng 2.2. Các mức truy nhập thành phần đối tượng., theo thứ tự từ trái sang phải, từ trên xuống dưới là mức truy nhập từ cao nhất đến thấp nhất. Cột ‘Trong lớp’ chỉ ra rằng một lớp luôn truy nhập được tới mọi thành phần của lớp

đó, bất kể chỉ thị truy nhập được sử dụng là gì. Cột ‘Lớp cùng gói’ chỉ ra rằng các lớp trong cùng một gói có khả năng truy nhập tới mọi thành phần của nhau, trừ các thành phần có chỉ thị truy nhập private. Cột ‘Lớp con khác gói’ chỉ ra rằng các lớp con được phép truy nhập các thành phần public và các thành phần protected của lớp cha. Cột ‘Lớp khác gói’ chỉ ra rằng các lớp bên ngoài gói (không có mối quan hệ kế thừa) chỉ có thể truy cập được các thành phần khai báo public.

Như vậy, mức truy nhập tác động theo hai khía cạnh: Thứ nhất, khi sử dụng thư viện các lớp trong mã nguồn khác, chẳng hạn các lớp trong nền tảng Java, các mức truy nhập quyết định các lớp đang viết có thể sử dụng thành phần nào của những lớp thư viện đó; thứ hai, khi viết thư viện (các lớp), ta cần quyết định mức truy nhập cho mỗi thuộc tính và mỗi phương thức trong lớp.

Ví dụ 2.11:

Tạo 2 gói oop.chapter2.sample và oop.chapter2.test. Trong gói oop.chapter2.sample ta xây dựng 2 lớp NgayThang và CanBo với thiết kế như dưới đây. Trong gói oop.chapter2.test ta xây dựng lớp ViDuCanBo.



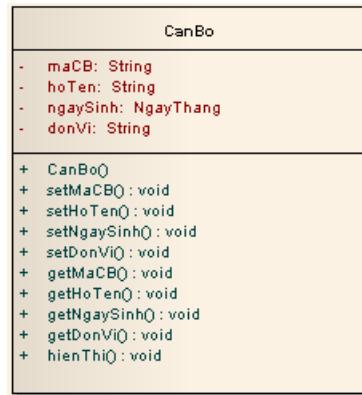
Hình 2.10. Ví dụ thiết kế gói.

Trong ví dụ này, các thuộc tính của lớp CanBo đều để mức truy nhập public. Điều này dẫn đến việc tự do truy nhập và tự do thiết lập giá trị cho các thuộc tính này, ngay cả khi giá trị đó không hợp lệ.

```
package oop.chapter2.test;
import oop.chapter2.sample.*;
public class ViDuCanBo{
    public static void main(String[] args) {
        NgayThang ngaySinh = new NgayThang(30, 6, 1987);
        CanBo canBo = new CanBo("123", "", ngaySinh, "");
        canBo.hienThi();
    }
}
```

Chẳng hạn, dữ liệu cán bộ nhập vào phải thỏa mãn những quy định sau: mã số cán bộ phải là một chuỗi độ dài 6 ký tự, bắt đầu bằng “cb” và theo sau bởi 4 chữ số; ngày sinh phải là ngày hợp lệ; họ tên và đơn vị phải nhập đầy đủ, không được để trống. Ở đây, maCB không được thiết lập theo đúng quy cách, hoTen và đơn vị không được nhập vào. Rõ ràng việc thiết lập các thuộc tính của lớp CanBo ở mức truy nhập public đã dẫn đến việc thiết lập giá trị tự do, không đảm bảo tính hợp lệ của dữ liệu trong trường hợp này. Giải pháp cho vấn đề trên là sử dụng mức truy cập hạn chế nhất (mức private) cho các thuộc tính của lớp CanBo, để bên ngoài không thể tự do truy nhập hay thiết lập giá trị cho các thuộc tính này của lớp. Trong trường hợp lớp khác cần thiết lập hay lấy giá trị của các thuộc tính private của lớp thì cần sử dụng phương thức public được đặt tên bắt đầu với *set* hay *get*.

Xét một thiết kế khác của lớp SinhVien. Ở thiết kế này, các thuộc tính của lớp CanBo đều để ở mức truy nhập private. Do đó, bên ngoài không thể tự do truy nhập hay thiết lập giá trị cho các thuộc tính này. Để truy nhập và thiết lập giá trị cho các thuộc tính, lớp có các phương thức công khai (giao diện) bắt đầu với từ *get* và *set*.



Hình 2.11. Thiết kế lớp CanBo.

```

public void setMaCB(String maCB) {
    this.maCB = maCB;
}
public void setHoTen (String hoTen) {
    this.hoTen = hoTen;
}
public String getMaCB () {
    return maCB;
}
public String getHoTen () {
    return hoTen;
}

```

Tuy nhiên, nếu chỉ dùng lại với các phương thức *set* được viết như trên thì hoàn toàn không có ý nghĩa đối với việc kiểm tra dữ liệu trước khi thiết lập giá trị cho thuộc tính. Xét các phương thức set dưới đây:

```

public void setMaCB(String maCB) {
    if (maCB.matches("cb\\d{4}"))
        this.maCB = maCB;
    else
        System.out.println("Ma can bo khong hop le");
}

public void setHoTen(String hoTen) {
    if (!hoTen.equals(""))
        this.hoTen = hoTen;
    else
        System.out.println("Phai nhap ho ten");
}

```

Chẳng hạn, phương thức setMaCB ở trên không chỉ đơn thuần thiết lập giá trị maCB từ tham biến đầu vào mà còn kiểm tra giá trị đó có hợp lệ hay không (qua việc đối chiếu với mẫu) trước khi thiết lập giá trị cho nó.

Như vậy, khi quyết định mức truy nhập cần lưu ý:

* Tránh sử dụng mức truy nhập *public* cho các trường thuộc tính vì các thuộc tính *public* có khuynh hướng dẫn tới mức tự do truy nhập, tự do thiết lập và hạn chế tính khả chuyển mã chương trình. Nên sử dụng mức truy nhập hạn chế nhất (*private*) đối với các thành phần, đặc biệt là dữ liệu để nâng cao tính an toàn dữ liệu của chương trình.

* Thao tác gián tiếp với các thuộc tính *private* thông qua các phương thức (giao diện) công khai là các phương thức *set* và *get*.

Chúng ta thường để các thuộc tính ở mức truy nhập hạn chế nhất là *private*. Đối với một số phương thức, chúng ta vẫn sử dụng mức truy nhập *private*. Chẳng hạn, trong lớp *NgayThang*, chúng ta có hai phương thức *private* là *kiemTraThang(int thang)* và *kiemTraNgay(int ngay)*.

Ví dụ 2.12:

```
private int kiemTraThang(int thang){  
    if (thang > 0 && thang <= 12)  
        return thang;  
    System.out.println("Thang khong hop le!");  
    return 0;  
}  
  
private int kiemTraNgay(int ngay){  
    int ngayTrongThang[] = {0, 31, 28, 31, 30, 31, 30, 31, 31,  
30, 31, 30, 31};  
    if (ngay > 0 && ngay <= ngayTrongThang[thang])  
        return ngay;  
    if (thang == 2 && ngay == 29 && (nam % 400 == 0 || (nam % 4  
== 0 && nam % 100 != 0)))  
        return ngay;  
    System.out.println("Ngay khong hop le!");  
    return 0;  
}
```

Chúng ta sử dụng mức truy nhập *private* cho hai phương thức này nhằm che giấu chi tiết thực hiện của toán tử tạo lập *NgayThang*.

```
public NgayThang(int ngay, int thang, int nam){  
    this.nam = nam;  
    this.thang = kiemTraThang(thang);  
    this.ngay = kiemTraNgay(ngay);
```

```
}
```

Với thiết kế mới của lớp CanBo, chương trình kiểm soát được dữ liệu nhập vào theo đúng định dạng yêu cầu.

Ví dụ 2.13:

```
package oop.chapter2.test;
import oop.chapter2.sample.*;

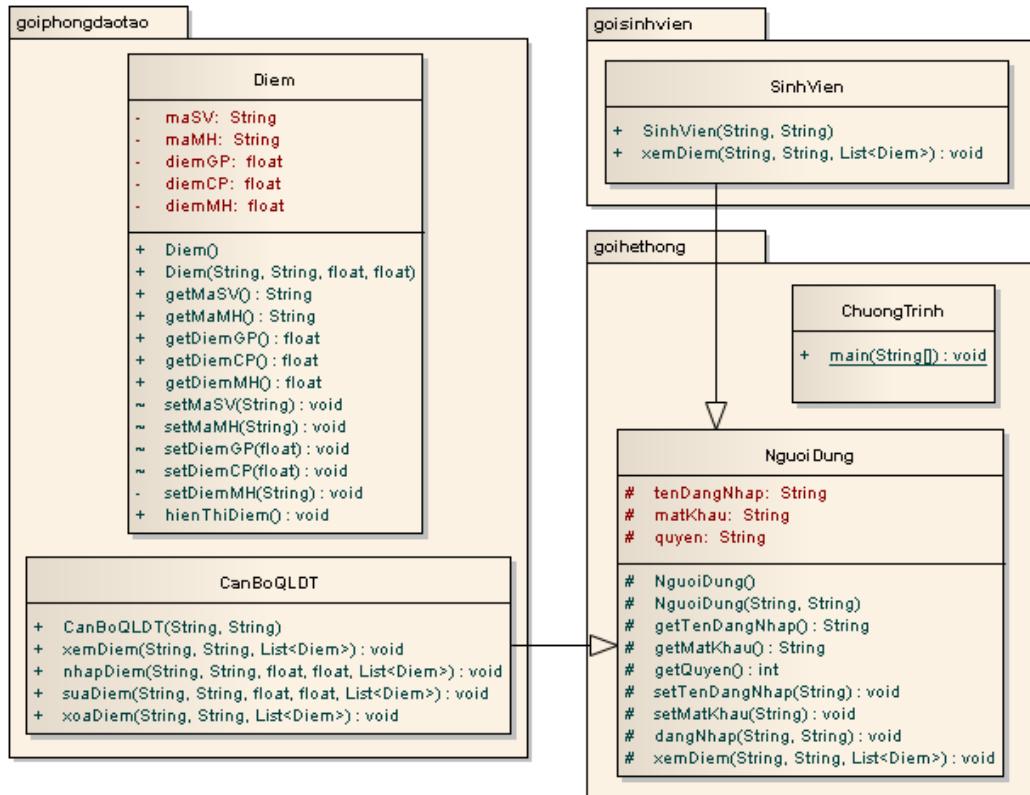
public class ViDuCanBo{
    public static void main(String[] args) {
        NgayThang ngaySinh = new NgayThang(29, 2, 1987);
        CanBo canBo = new CanBo();
        CanBo.setMaCB("123");
        CanBo.setHoTen("");
        CanBo.setNgaySinh(ngaySinh);
        CanBo.setDonVi("");
    }
}
```

Kết quả chạy chương trình như sau:

```
Ngay khong hop le!
Ma can bo khong hop le.
Phai nhap ho ten.
Phai nhap don vi

Process completed
```

Trở lại với chương trình Quản lý điểm, chúng ta sẽ sử dụng các kiến thức vừa học để thiết kế cải tiến các lớp của chương trình, thực hiện kiểm soát truy nhập tới các lớp và các thành phần lớp một cách hợp lý dựa trên các yêu cầu nghiệp vụ đưa ra. Kết quả như sau:



Hình 2.12. Thiết kế tổ chức chương trình quản lý điểm.

Một số lưu ý khi sử dụng đối tượng

- Thành phần lớp và thành phần thể hiện

Các thuộc tính, phương thức trong một lớp có thể là các thuộc tính, phương thức tĩnh (*static*) hoặc không tĩnh (*non-static*). Trong phần này, chúng ta thảo luận việc sử dụng từ khóa *static* để tạo các thuộc tính và phương thức thuộc về lớp – tức dùng chung cho mọi đối tượng của lớp (phân biệt với các thuộc tính và phương thức thuộc về đối tượng - không sử dụng từ khóa *static*).

Các thành phần được khai báo với từ khóa *static* còn được gọi là các *thành phần lớp* (*class variables*) hay các *thành phần tĩnh* (*static fields*). Các thành phần không được khai báo với từ khóa *static* được gọi là các *thành phần thể hiện* (*instance variables*). Khi tạo một thể hiện (tức một đối tượng) của một lớp, hệ thống sẽ phân bổ không gian bộ nhớ cho đối tượng đó và tất cả các thành phần thể hiện của nó.

+ Thuộc tính lớp và thuộc tính thể hiện

Một thuộc tính *static* chứa thông tin được chia sẻ bởi tất cả các đối tượng của lớp. Thuộc tính *static* đại diện cho lớp nên còn được gọi là thuộc tính lớp và có một vùng nhớ cố định. Bất kể đối tượng nào cũng có thể thay đổi giá trị của thuộc tính lớp. Nếu một đối tượng thay đổi thuộc tính lớp thì thuộc tính đó trong tất cả các đối tượng khác của lớp cũng thay đổi giá trị theo. Ta có thể thao tác với các thuộc tính lớp mà không cần tạo thể hiện của lớp.

Ví dụ 2.14:

Giả sử trong ứng dụng, ta tạo một số đối tượng User và gán mỗi đối tượng một id, bắt đầu với id = 1 cho đối tượng đầu tiên. Số id này là duy nhất cho mỗi đối tượng và vì vậy là biến thể hiện. Tuy nhiên, ta cần có một trường lưu số đối tượng User đã tạo ra để biết số id cần gán cho đối tượng tiếp theo và trường này liên quan tới toàn bộ lớp User.

```
public class User{  
    private int id;  
    private String pass;  
    private String name;  
    private Date birthday;  
    public static int numberofUser = 0;  
}
```

Ta có thể thao tác với các biến bằng tên lớp hoặc tham chiếu đối tượng như sau:

```
User.numberofUser  
user.numberofUser
```

- Phương thức lớp và phương thức thể hiện

Java cũng hỗ trợ phương thức tĩnh (phương thức với khai báo có từ khóa *static*). Phương thức được khai báo với từ khóa *static* là phương thức lớp. Phương thức không được khai báo với từ khóa *static* là phương thức thể hiện. Để gọi phương thức tĩnh, không cần tạo đối tượng mà chỉ cần dùng tên lớp. Tuy nhiên, vẫn có thể gọi phương thức tĩnh với một tham chiếu đối tượng.

Phương thức tĩnh thường được sử dụng để truy cập các trường tĩnh. Ví dụ, ta có thể thêm phương thức tĩnh vào lớp User để truy nhập trường tĩnh `numberOfUser`

```
public static int getNumberOfUser() {  
    return numberOfUser;  
}
```

- + Các phương thức静态 hiện có thể truy nhập trực tiếp các biến static hiện và phương thức static hiện.
- + Các phương thức static hiện có thể truy nhập trực tiếp các biến lớp và phương thức lớp.
- + Các phương thức lớp có thể truy nhập trực tiếp các biến lớp và phương thức lớp.

+ Các phương thức lớp không thể truy nhập trực tiếp biến static hiện hay phương thức static hiện mà phải sử dụng một tham chiếu đối tượng. Vì vậy, phương thức lớp không thể sử dụng từ khóa `this` vì không có static hiện nào cho `this` tham chiếu.

- Tham chiếu đối tượng

Trong Java, lớp là một kiểu dữ liệu, tương tự các kiểu được xây dựng sẵn như `int` và `boolean`, vì vậy, một tên lớp có thể được sử dụng để đặc tả kiểu biến trong một câu lệnh khai báo, kiểu tham số hình thức hay kiểu trả về của phương thức. Ví dụ câu lệnh dưới đây định nghĩa một biến loại `SinhVien` với tên `sinhVien`:

```
SinhVien sinhVien;
```

Tuy nhiên, khai báo một biến không tạo ra một đối tượng. Trong Java, không có biến nào có thể nắm giữ một đối tượng. Một biến chỉ có thể nắm giữ một tham chiếu tới một đối tượng. Các đối tượng được tạo ra trong vùng *bộ nhớ heap* của bộ nhớ máy tính. Thay vì nắm giữ chính đối tượng, biến `sinhVien` nắm giữ thông tin cần thiết để tìm ra đối tượng trong bộ nhớ. Thông tin này gọi là tham chiếu hay con trỏ tới đối tượng. Thực tế, một tham chiếu tới một đối tượng là địa chỉ

vùng nhớ nơi lưu trữ đối tượng. Khi sử dụng một biến kiểu lớp, máy tính sử dụng tham chiếu trong biến để tìm đến đối tượng thực sự trong vùng nhớ heap.

```
sinhVien = new SinhVien();
```

Tạo lập đối tượng sử dụng toán tử *new*. Toán tử này tạo một đối tượng và trả về một tham chiếu tới đối tượng đó. Câu lệnh trên tạo một đối tượng mới là thể hiện của lớp SinhVien, và lưu tham chiếu tới đối tượng vừa tạo vào biến sinhVien. Như vậy, giá trị của biến sinhVien là tham chiếu tới đối tượng, không phải là đối tượng.

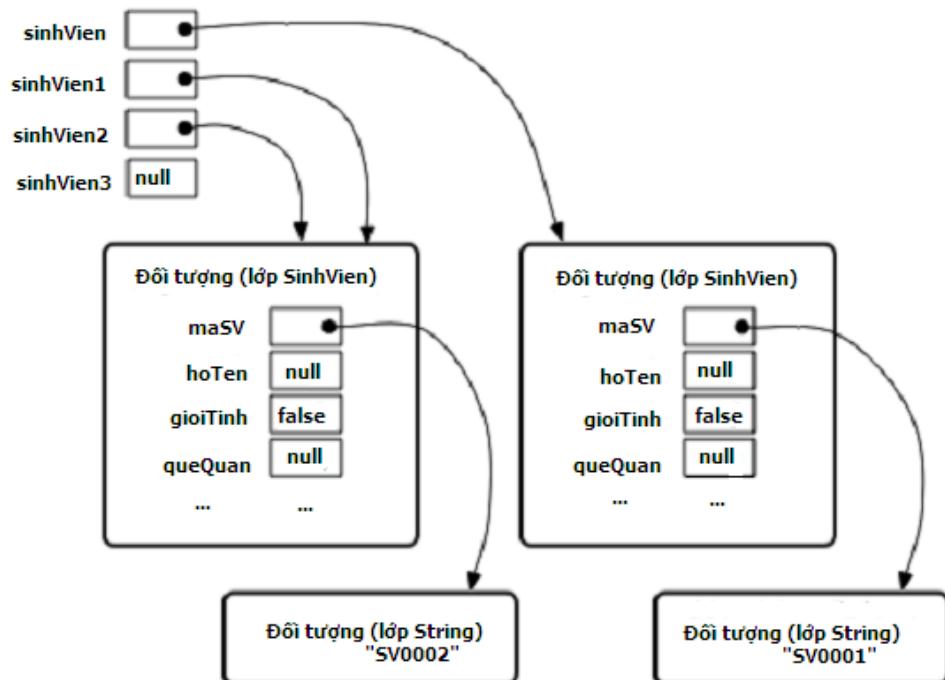
Ví dụ: `sinhVien.maSV`, `sinhVien.hoTen`, `sinhVien.ngaySinh`, `sinhVien.gioiTinh`, `sinhVien.queQuan...` tham chiếu tới các biến thể hiện là `maSV`, `hoTen`, `ngaySinh`, `gioiTinh`, `queQuan` của đối tượng loại SinhVien mà biến `sinhVien` tham chiếu tới.

```
SinhVien sinhVien, sinhViet1, sinhVien2, sinhVien3;
sinhVien = new SinhVien();
sinhVien1 = new SinhVien();
sinhVien2 = sinhVien1;
sinhVien3 = null;
sinhVien.maSV = "SV0001";
sinhVien2.maSV = "SV0002";
System.out.prinn(sinhVien.maSV);
System.out.println(sinhVien1.maSV);
System.out.println(sinhVien2.maSV);
```

Kết quả

```
SV001
SV002
SV002
```

+ Khi một biến đối tượng được gán cho biến khác, chỉ có tham chiếu được sao chép. Đối tượng được trả tới không được sao chép. Với câu lệnh “`sinhVien2 = sinhVien1;`” không đối tượng mới nào được tạo ra mà `sinhVien2` được thiết lập để liên hệ tới chính đối tượng `sinhVien1` trả tới.



Hình 2.13. Minh họa tham chiếu đối tượng.

- + Câu lệnh “`if sinhVien1 == sinhVien2`” kiểm tra liệu sinhVien1 và sinhVien2 có trỏ tới cùng đối tượng hay không.
- + Câu lệnh “`if (sinhVien1.maSV == sinhVien2.maSV)`” kiểm tra các biến thể hiện trong các đối tượng mà sinhVien1 và sinhVien2 trỏ tới có giá trị giống nhau hay không.
- Truyền tham số trong lời gọi phương thức đối tượng

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

- + Số các biến của danh sách tham biến hình thức phải bằng số các tham biến của danh sách tham biến hiện thời.

+ Kiểu của các tham biến hiện thời phải tương thích với kiểu của tham biến hình thức tương ứng.

Kiểu của tham biến hình thức	Giá trị được truyền
Kiểu nguyên thủy	Giá trị kiểu nguyên thủy
Kiểu tham chiếu	Giá trị tham chiếu

- 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 tham biến hiện thời cũng giống như quy tắc chuyển đổi kiểu.

Ví dụ 2.15:

```
static void khongThayDoi(int z) {
    z = 10;
}
```

Thực hiện đoạn lệnh

```
//truyen gia tri kieu nguyen thuy
int x = 5;
System.out.println("Truoc goi ham: x = " + x);
khongThayDoi(x);
System.out.println("Sau goi ham: x = " + x);
```

Kết quả:

```
Truoc goi ham: x = 5
Sau goi ham: x = 5
```

Như vậy, giá trị của x không bị thay đổi sau lời gọi phương thức.

- Truyền các giá trị kiểu tham chiếu

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 2.16:

```
static void thayDoi(SinhVien sv){  
    sv.maSV = "SV0000";  
}
```

Thực hiện đoạn lệnh

```
//truyen gia tri kieu tham chieu  
System.out.println("Mã sinh viên trước gọi hàm: " + sv2.maSV);  
thayDoi(sv2);  
System.out.println("Mã sinh viên sau gọi hàm: " + sv2.maSV);
```

Kết quả:

```
Mã sinh viên trước gọi hàm: SV0001  
Mã sinh viên sau gọi hàm: SV0000
```

Như vậy, giá trị của sinhVien không bị thay đổi sau lời gọi phương thức nhưng giá trị của biến thể hiện sinhVien.maSV thì bị thay đổi.

c) *Dữ liệu kiểu hằng, kiểu tĩnh*

Các phần trước đã hướng dẫn cách viết các thuộc tính và các phương thức trong một lớp. Trong phần này sẽ giới thiệu sâu hơn về các thuộc tính và phương thức khi chúng được khai báo là hằng hoặc tĩnh.

- *Dữ liệu kiểu hằng*: Bao gồm các loại sau:

- + Dữ liệu kiểu hằng từ lúc biên dịch chương trình (*compile-time constant*), giá trị dữ liệu không thể thay đổi sau đó.
- + Dữ liệu được khởi tạo trong thời gian chạy chương trình (*run-time constant*) và không thay đổi giá trị sau đó.

Trong trường hợp dữ liệu là hằng từ lúc biên dịch, giá trị hằng được gán ngay trong dòng khai báo hằng. Trình biên dịch giữ nguyên giá trị hằng này trong bất kì phép toán nào sau dòng lệnh khai báo hằng có sử dụng hằng đó. Trong ngôn ngữ Java, dữ liệu hằng kiểu như trên sử dụng từ khóa *final* để khai báo. Ví dụ `final int i1 = 9;` khai báo một hằng kiểu *int*, có tên là *i1*, có giá trị là 9. Giá trị 9 này sẽ không thay đổi trong suốt quá trình tồn tại của *i1*. Vì thế, giả sử viết `i1 = x;` thì chương trình sẽ có lỗi.

Khi làm việc với dữ liệu kiểu hằng, cần chú ý phân biệt với một số loại tương tự với nó như sau:

- + Thành viên hằng– *final member*, thành viên tĩnh– *static member* và hằng thành viên tĩnh– *final static member*.
- + Hằng tham số final parameter.
- + Hằng hàm/phương thức– *final method* và hàm/phương thức tĩnh– *static method*.
- + Lớp hằng final class.

- *Thuộc tính khai báo là hằng – final*: Khai báo thêm cho thuộc tính từ khóa *final*. Khi đó, thuộc tính đó sẽ có giá trị được giữ nguyên trong suốt thời gian sống của đối tượng chủ.

Ví dụ 2.17: Một lớp học cố định chỉ có 40 sinh viên. Cài đặt lớp *Lop* có thuộc tính *sosv* là hằng có giá trị 40

```
public Lop {  
    final int sosv = 40;  
    Lop () {}  
    // Code.....  
}
```

Giả sử ta có khai báo đối tượng *Lop*: `Lop lop1 = new Lop();` *sosv* là hằng số được tạo trong thời gian biên dịch chương trình. Khi biên dịch, gấp dòng khai báo `Lop lop1 = new Lop()` trình biên dịch sẽ tạo ra 1 biến có tên là *lop1*, có kiểu là *Lop*, *lop1* có thuộc tính *sosv* là hằng số được cấp phát một vùng nhớ bằng kích

cõ của kiểu *int*, gán giá trị cho vùng này là 40 và giá trị này sẽ không thay đổi trong suốt quá trình tồn tại của *lop1*.

Giả sử ta có một khai báo khác: *Lop lop2 = new Lop()* thì *lop2* này cũng có thuộc tính *sosv* là hằng số nguyên có giá trị hằng là 40. *lop1*, *lop2* là 2 biến khác nhau, *lop1.sosv* và *lop2.sosv* là 2 hằng khác nhau cùng có giá trị hằng là 40. Các lớp con của *Lop* cũng không thể thay đổi được giá trị hằng này.

Ví dụ dưới đây minh họa sự khác nhau giữa 2 loại hằng dữ liệu *compile-time* và *run-time*.

Ví dụ 2.18: Mỗi lớp được cấp ngẫu nhiên một số máy tính. Sử dụng *computer* là hằng số trong thời gian chạy do *Math.random* khởi tạo giá trị.

```
import java.lang.Math;
public class Lop{
    final int sosv = 40;
    final int computer =(int)(Math.random()*20);
    //public Lop() {}
    public void print(){
        System.out.println(sosv+" sinh vien co "+computer+" may
tinh");
    }
    public static void main(String[] args){
        Lop lop1 = new Lop();
        Lop lop2 = new Lop();
        Lop lop3 = new Lop();
        lop1.print();
        lop2.print();
        lop3.print();}
}
```

```

package Vihicle;
import java.lang.Math;

public class MyClass{
    final int number = 40;
    final int computer =(int) (Math.random()*20);
    //public MyClass(){}
    public void print(){
        System.out.println(number+" sinh vien co "+computer+" may tinh");
    }
    public static void main(String[] args){
        MyClass lop1 = new MyClass();
        MyClass lop2 = new MyClass();
        lop1.print();
        lop2.print();
    }
}

```

Output - Videl (run)

```

run:
40 sinh vien co 1may tinh
40 sinh vien co 4may tinh

```

Hình 2.14. Kết quả chạy chương trình.

- Thuộc tính khai báo là tĩnh – *static*

Một phương thức *x* của một lớp *c* được khai báo là *static* thì sẽ có một bản dùng chung duy nhất cho tất cả thể hiện của lớp *c*, bất kể lớp *c* được khai báo bao nhiêu thể hiện.

Ví dụ 2.19: Cần quản lý tổng số lớp hiện có. Sử dụng một biến đếm để mỗi khi có 1 lớp được sinh ra giá trị biến đó được tăng lên. Như vậy, lớp *Lop* khai báo thêm 1 thuộc tính *count* là *static*

```

import java.lang.Math;
public class Lop{
    final int sosv = 40;
    final int computer =(int) (Math.random()*20);
    static int count=0;
    //public Lop(){}
    public void print(){
        System.out.println("Lop thu "+(++count)+" hien nay
gom"+sosv+" sinh vien co "+computer+" may tinh");
    }
    public static void main(String[] args){
        Lop lop1 = new Lop();
        Lop lop2 = new Lop();
        Lop lop3 = new Lop();
        lop1.print();
        lop2.print();
        lop3.print();
    }
}

```

```

package Vihicle;
import java.lang.Math;

public class MyClass{
    final int number = 40;
    final int computer =(int) (Math.random()*20);
    static int count=0;
    //public MyClass(){}
    public void print(){
        System.out.println("Lop thu "+(++count)+" hiem nay gom"+number
                           +" sinh vien co "+computer+" may tinh");
    }
    public static void main(String[] args){
        MyClass lop1 = new MyClass();
        MyClass lop2 = new MyClass();
        MyClass lop3 = new MyClass();
        lop1.print();
        lop2.print();
        lop3.print();
    }
}

```

Hình 2.15. Biến count được dùng chung cho mọi thể hiện của Lop.

Trong chương trình trên, *lop1*, *lop2* và *lop3* cùng dùng chung thuộc tính *count* nên mỗi lần gọi phương thức *print* dù 3 lời gọi từ 3 lớp khác nhau, phương thức *(++count)* trong khai báo *print()* lại tăng giá trị của *count* lên 1 đơn vị. Ban đầu khởi tạo là 0, sau lệnh *lop1.print()* tăng lên 1, sau lệnh *lop2.print()* tăng lên 2 và sau lệnh *lop3.print()* tăng lên 3. Chương trình này cũng thể hiện sự hoạt động khác nhau của thuộc tính tĩnh và thuộc tính hằng. Vậy nếu 1 thuộc tính vừa là hằng vừa là tĩnh thì khi đó dữ liệu sẽ được khai báo là *final static*.

- *Phương thức hằng*: Có 2 lý do để cài đặt một phương thức là hằng:

+ Ngăn cho lớp thừa kế thay đổi nội dung của phương thức. Khi phương thức cha là hằng, lớp con không thể vượt quyền (*overriding*) phương thức của cha.

+ Lý do thứ 2 thuộc về phần biên dịch, khi một lời gọi phương thức hằng xảy ra, trình biên dịch sẽ bỏ qua một số bước dịch thông thường như: đẩy tham số vào bộ nhớ *stack*, nhảy tới phần mã nguồn của phương thức và thực thi nó, lấy các tham số khỏi *stack*, xử lý dữ liệu trả về v.v..

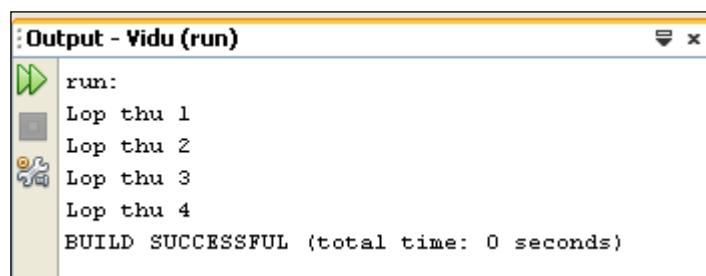
Một phương thức được khai báo là *private*, thì nó cũng là một phương thức hằng. Nó không truy nhập được bởi lớp con, không bị ghi đè... Phương thức *private* sẽ được đề cập thêm ở phần đa hình.

- *Phương thức tĩnh*: Một phương thức tĩnh có thể được gọi một cách độc lập với mọi thẻ hiện của lớp.

- + Phương thức tĩnh không được truyền biến *this* làm tham số ẩn.
- + Không thể sửa đổi các phương thức dữ liệu từ trong phương thức tĩnh trừ khi phương thức dữ liệu đó cũng là tĩnh.
- + Có thể gọi phương thức tĩnh mà không cần tạo thẻ hiện nào của lớp - gọi thẳng bằng tên lớp.

Ví dụ 2.20: Quay trở lại với ví dụ về *Lop*, nếu như có khai báo:

```
public class Lop{  
    static private int count=0;  
    static public void print(){  
        System.out.println("Lop thu "+(++count));  
    }  
    public static void main(String[] args){  
        Lop lop1 = new Lop();  
        Lop lop2 = new Lop();  
        Lop lop3 = new Lop();  
        lop1.print();  
        lop2.print();  
        lop3.print();  
        Lop.print();  
    }  
}
```



Hình 2.16. Phương thức tĩnh có thể gọi trực tiếp từ tên lớp.

count là biến *static* nên nó độc lập với *lop1*, *lop2*, *lop3*. Mỗi lần *lop1*, *lop2*, *lop3* và *Lop* gọi phương thức *print()* nó đều làm tăng giá trị của *count* lên 1 đơn

vị. Thêm vào đó phương thức *print* làm việc với *count* là *static* nên *print* cũng phải là *static*.

- Lớp **hằng**

Khai báo một lớp là **hằng**, thêm từ khóa *final* vào trước từ khóa *class*, nhằm mục đích không cho lớp ấy có lớp con. Trong một số bài toán với mục đích an toàn và bảo mật, người thiết kế có thể thiết kế lớp đối tượng nào đó là **hằng**.

Câu hỏi và bài tập cuối chương

Câu 1. Điền vào chỗ trống:

- a. Một ngôi nhà đã xây xong từ một bản thiết kế giống như một..... của một lớp (class).
- b. Định nghĩa một class bắt đầu bằng từ khóa....., lớp phải có tên file.....như tên lớp, với phần định dạng là .java.
- c. Đứng trước tên lớp là từ khóa.....
- d. Từ khóa..... để tạo ra một đối tượng của một lớp. Từ khóa này nằm bên trái tên lớp trong dòng khai báo.
- e. Mỗi một tham số truyền vào cho một phương thức phải có..... và....
- f. Từ khóa public là.....
- g. Để khai báo một đối tượng của một lớp đã được định nghĩa trong một package nào đó ta phải dùng câu lệnh.....

Câu 2. Hãy tạo lớp GradeBook

- a) Trong lớp viết phương thức displayMessage() hiển thị dòng chữ "Welcome to GradeBook!". Trong hàm main hiển thị dòng chữ này.
- b) Hãy sửa chương trình trên bằng cách thêm tên của khóa học vào phương thức displayMessage. Dòng thông báo in ra là: "Chào mừng bạn đến với khoa hoc" + tên của khóa học.
- c) Xây dựng lại lớp GradeBook có các thuộc tính và phương thức sau:
 - courseName là tên của khóa học, kiểu String, quyền private
 - setCourseName(): truyền tên cho courseName, kiểu public
 - getCourseName(): lấy ra tên của khóa học
 - displayMessage(): hiển thị ra thông báo: Chào mừng tới khóa học....

Hàm main: hãy sử dụng lớp GradeBook trên, nhập tên khóa học, hiển thị dòng thông báo. Chú ý hàm khởi tạo do các bạn tự viết cho phù hợp.

Câu 3.

- Tạo lớp Employee gồm có 3 thuộc tính: tên (kiểu String), họ (kiểu String) và lương hàng tháng (double). Lớp phải có một constructor khởi tạo 3 thuộc tính trên. Và các phương thức cập nhật và truy vấn (set và get) cho từng thuộc tính này. Nếu lương tháng là số không dương, thì gán nó là 0.0.

- Viết một chương trình demo đặt tên là EmployeeTest minh họa các khả năng có thể có của lớp Employee vừa cài đặt. Tạo ra 2 đối tượng của lớp này và hiển thị lương một năm của họ. Sau đó cho mỗi Employee lương tăng thêm 10% và hiển thị lại lương họ kiểm được trong 1 năm.

Câu 4. Xây dựng lớp Vecto để thực hiện các phép toán trên Vecto, bao gồm:

- Hàm tạo Vecto(int n) để khởi tạo một Vecto gồm n thành phần (vecto trong không gian Rn)

- Phương thức “cong” để thực hiện phép cộng hai Vecto

- Phương thức “hieu” để thực hiện phép hiệu hai vecto

- Phương thức “tichVoHuong” để thực hiện phép tích vô hướng của hai vecto

- Phương thức “tich” để thực hiện phép nhân vecto với một số

- Phương thức “doDai” để lấy độ dài của một vecto

- Phương thức “chuanHoa” để thực hiện việc chuẩn hoá vecto

- Phương thức “nhap” để nhập các toạ độ cho vecto

- Phương thức “print” để in các toạ độ của vecto ra màn hình

- Phương thức “vuongGoc” để kiểm tra xem hai Vecto có vuông góc với nhau hay không?

Câu 5. Xây dựng lớp SoPhuc để thực hiện các phép toán trên số phức, bao gồm:

- Hàm tạo SoPhuc(float pt, float pa) để khởi tạo phần thực bằng pt và phần ảo bằng pa
- Hàm tạo SoPhuc() để khởi tạo phần thực bằng 0 và phần ảo bằng 0
- Phương thức “cong” để thực hiện phép cộng hai số phức
- Phương thức “hieu” để thực hiện phép trừ hai số phức
- Phương thức “chia” để thực hiện phép chia hai số phức
- Phương thức “nghichDao” để tính nghịch đảo của một số phức
- Phương thức “soSanhBang”, “lonHon”, “nhoHon” so sánh hai số phức
- Phương thức “nhap” dùng để nhập số phức từ bàn phím
- Phương thức “in” dùng để in số phức ra màn hình

Câu 6. Xây dựng lớp PhanSo để thực hiện các phép toán trên phân số, bao gồm:

- Hàm tạo PhanSo() để khởi tạo phân số có tử số bằng 0 và mẫu số bằng 1.
- Hàm tạo PhanSo(int ts, int ms) để khởi tạo phân số có tử số bằng ts và mẫu số bằng ms.
 - Phương thức “cong”, “tru”, “nhan”, “chia” để thực hiện việc cộng, trừ, nhân, chia hai phân số.
 - Phương thức “nghichDao”, “doiDau”, “toiGian” để thực hiện việc nghịch đảo, đổi dấu và tối giản một phân số.
 - Phương thức “soSanhBang”, “lonHon”, “nhoHon” để thực hiện việc so sánh hai phân số.
 - Phương thức “nhap” dùng để nhập phân số từ bàn phím
 - Phương thức “in” dùng để in phân số ra màn hình

Chương 3. THÙA KẾ VÀ ĐA HÌNH

I. THÙA KẾ

1. Giới thiệu về thừa kế

a) *Vấn đề sử dụng lại trong lập trình*

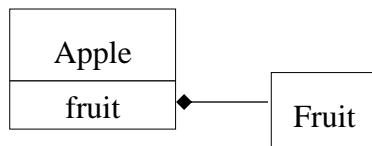
Cùng với sự phát triển của phần mềm, lập trình tạo ra các phần mềm cũng tăng dần về số lượng dòng lệnh, loại phần mềm, loại ngôn ngữ lập trình. Việc phải lập trình lại mọi thứ từ đầu không phải là một sự lựa chọn thông minh, thay vào đó, người lập trình có xu hướng sử dụng lại các đoạn chương trình đã viết để phục vụ cho các mục đích tiếp theo. Vì thế vấn đề sử dụng lại trong lập trình trở thành phổ biến.

Có nhiều cách để có thể sử dụng lại mã nguồn. Một cách làm khá thủ công mà hầu như lập trình viên nào cũng đã từng làm là sao chép lại mã nguồn bằng thao tác copy – paste. Cách thực hiện này khá đơn giản, tuy nhiên, chỉ nên làm với các chương trình hoặc các đoạn chương trình nhỏ, có sự tương đồng lớn về mã nguồn. Trong các chương trình lớn, việc copy-paste nhiều lần một đoạn chương trình nào đó sẽ không còn là cách làm hay nếu sau đó ta phát hiện ra đoạn chương trình đã sao chép không phù hợp với mong muốn. Lúc này ta sẽ phải dò lai tất cả các đoạn chương trình đã copy để thực hiện chỉnh sửa. Khi phát sinh lỗi từ các đoạn chương trình đã copy-paste cũng vậy, sẽ rất mất thời gian để thực hiện dò lỗi và chỉnh sửa từ tất cả các đoạn chương trình đã copy-paste này. Như vậy, nhược điểm thao tác copy – paste đoạn chương trình cần dùng lại là dễ phát sinh lỗi, khó tìm lại lỗi, và vẫn phải chỉnh sửa lại đoạn mã nguồn đã sao chép cho phù hợp với bài toán mới. Để khắc phục nhược điểm này, ngôn ngữ lập trình Java đã hỗ trợ phương pháp sử dụng lại các đoạn chương trình khoa học hơn so với phương pháp thủ công nêu trên. Đó là việc tạo lớp mới dựa vào những cung cấp có sẵn từ các lớp đã được biên dịch. Việc tạo lớp này dựa trên các mối quan hệ như sau:

- *Quan hệ “chứa”* (“*has – a*”) hay *quan hệ tích hợp thành phần* (*Composition*): Quan hệ này chỉ ra rằng có 2 lớp A và B, trong đó lớp B chỉ sử dụng một phần mĩ chương trình của lớp A (phương thức, thuộc tính) mà không sửa đổi gì về nội dung của lớp A. Do vậy, lớp A được coi là *hợp phần* (*composition*) của lớp B khi trong khai báo B chứa A (hay nói cách khác A thuộc về B). Để hiểu rõ hơn về quan hệ này, ta xét ví dụ phân loại hoa quả. Giả sử ta có một lớp *Fruit* để mô tả các loại hoa quả. Vì táo cũng là một loại hoa quả, do đó táo có tính chất chung nhất của các loại quả. Khi đó, nếu ta tạo hai lớp *Fruit* và *Apple* để chỉ hoa quả nói chung và quả táo nói riêng thì quan hệ *has-a* của lớp *Apple* với lớp *Fruit* được thể hiện như Hình 3.1 với khai báo trong Java như sau:

Ví dụ 3.1: Mô hình quan hệ has-a: lớp Apple có 1 thuộc tính là 1 đối tượng tham chiếu tới lớp *Fruit*.

```
// lớp hoa quả
public class Fruit {
    // Code .....
}
// lớp Táo
public class Apple {
    // táo là 1 loại quả
    private Fruit fruit = new Fruit();
}
```



Hình 3.1. Mô hình quan hệ has-a: Apple có 1 thuộc tính là 1 đối tượng tham chiếu tới lớp *Fruit*.

Lớp *Fruit* được tạo trước lớp *Apple*. Trong lớp *Apple* khai báo biến *fruit* có kiểu *Fruit*. Câu lệnh `private Fruit fruit = new Fruit();` để tạo ra 1 đối tượng *fruit* trong lớp *Apple*.

Ưu điểm của lập trình theo kiểu hợp thành phần là tính đơn giản, rõ ràng, dễ lập trình, dễ sử dụng.

Ví dụ sau đây thể hiện mối quan hệ này với một bài toán phức tạp hơn đó là biểu diễn mối quan hệ Người – Nhân viên – Quản lý.

Ví dụ 3.2: Lập trình tạo 3 lớp Người – Nhân viên – Quản lý

```
public class Nguoi {
    private String ten; // tên
    private Date ngaysinh; // ngày sinh
    public String layTen() {
        // lấy ra tên người
        return ten;
    }
}
public class Nhanvien {
    //Khai báo 1 người - Quan hệ has-as
    private Nguoi nguoi;
    private double luong; // lương
    public String layTen() {
        // lấy ra tên nhân viên
        return nguoi.layTen()
    }
}
public class Quanly {
    private Nhanvien ql; // thông tin của giám đốc
    private Nhanvien troly; // thông tin của cấp dưới
    // điền thông tin trợ lý
    public lapQuanly(Nhanvien nv) /* code... */
}
```

Ở ví dụ trên ta thấy xuất hiện một số nhược điểm khi sử dụng lập trình kiểu hợp thành. Thứ nhất, lớp *Nguoi* viết `String layTen()` để trả về tên của người, lớp *Nhanvien* cũng viết `String layTen()` trả về tên của nhân viên đó. Như vậy, ta đã viết 2 phương thức `layTen()` giống nhau, cùng thực hiện một thao tác trả lại tên, tuy nhiên 2 phương thức này không thể thay thế được cho nhau. Đây chính là sự thiếu tính linh hoạt, mềm dẻo khi lập trình kiểu hợp thành. Ngoài ra, khi sử dụng lớp quản lý *Quanly*, giả sử ta có 2 người quản lý:

```
Quanly a = new Quanly ();
Quanly b = new Quanly ();
b.lapQuanly(a); // gây ra lỗi
```

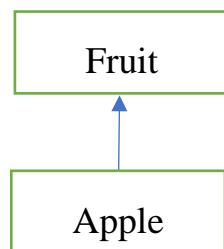
Với khai báo như trên, có *a* là nhà quản lý, *b* cũng là nhà quản lý, tuy nhiên nếu trong trường hợp cần biểu diễn người *a* quản lý người *b* thì với chương trình ở ví dụ trên là không thể. Do vậy, kiểu quan hệ thứ hai được đề xuất.

- Quan hệ “là” (“is – a”) giữa các đối tượng và kiểu lập trình thừa kế (Inheritance)

Lập trình thừa kế là cách lập trình tạo ra một lớp mới như là một kiểu hay một dạng của lớp cũ. Với cách này, lớp mới có thể thừa hưởng bản chất hay định dạng của lớp cũ mà không phải sao chép lại bất kỳ dòng mã nguồn nào của lớp cũ. Hay nói cách khác, lớp mới đã *thừa kế* mọi thuộc tính và phương thức của lớp đã có.

Ví dụ 3.3: Ví dụ Táo (Apple) cũng là một loại quả (Fruit)

```
// lớp quả
public class Fruit {
    // code ....
}
// lớp Táo
public class Apple extends Fruit {
    // code ....
}
```



Hình 3.2. Quan hệ *is-a*: Táo là 1 loại quả. Apple cũng là Fruit.

Trong ví dụ trên lớp *Fruit* được tạo trước lớp *Apple*, lớp *Apple* thừa kế từ lớp *Fruit*. Mọi thuộc tính và phương thức trong lớp *Fruit* thì lớp *Apple* đều có.

Ngoài việc lớp mới được thừa kế từ lớp cũ, nó còn được mở rộng cho phù hợp với mục đích sử dụng của lớp đó như: thêm vào các thuộc tính mới, thêm hay hiệu chỉnh các phương thức. Việc nhóm các đối tượng có cùng tập thuộc tính/phương thức thường là nhóm một số (chứ không phải tất cả) thuộc tính/phương thức giống nhau của các đối tượng. Ví dụ, nhóm tất cả xe chạy bằng động cơ thành một nhóm, rồi phân thành các nhóm nhỏ hơn tùy theo loại xe (xe ca, xe tải, xe máy ...). Mỗi nhóm con là một lớp các đối tượng tương tự, nhưng giữa tất cả

các nhóm con lại có chung một số đặc điểm. Khi đó quan hệ giữa các nhóm con với nhóm lớn được gọi là quan hệ “là”. Ví dụ: *một cái xe ca “là” xe động cơ; một cái xe tải “là” xe động cơ; một cái xe máy “là” xe động cơ...* Các đối tượng được nhóm lại thành một lớp thì có cùng tập thuộc tính và phương thức. Do đó, mọi đối tượng xe động cơ có cùng tập thuộc tính và phương thức, cũng như vậy, mọi đối tượng xe tải có cùng tập thuộc tính và phương thức.

Mỗi liên kết giữa các lớp trong quan hệ “là” xuất phát từ thực tế các lớp con cũng có mọi thuộc tính/phương thức của lớp cha, và có thêm các thuộc tính/phương thức khác. Như vậy, có thể thấy *thừa kế là sự cho phép 1 lớp chia sẻ các thuộc tính và phương thức của mình trong 1 hay nhiều lớp khác*.

b) Các thành phần trong quan hệ thừa kế

Trong quan hệ thừa kế, cần quan tâm đến các thành phần sau:

- Lớp cha - *superclass* (hoặc lớp cơ sở - *base class*): Lớp tổng quát hơn trong quan hệ “là”; Các đối tượng thuộc lớp cha có cùng tập thuộc tính và phương thức.

- Lớp con - *subclass* (hoặc lớp dẫn xuất – *derived class*): Lớp cụ thể hơn trong quan hệ “là”; Các đối tượng thuộc lớp con có cùng tập thuộc tính và phương thức (do thừa kế từ lớp cha), kèm thêm tập thuộc tính và phương thức của riêng lớp con.Ta nói rằng lớp con “thừa kế từ” lớp cha, hoặc lớp con “được dẫn xuất từ” lớp cha. Ví dụ : “xe tải thừa kế từ xe có động cơ”, “xe ca được dẫn xuất từ xe có động cơ”...

c) Phân tích bài toán theo mối quan hệ thừa kế

Để phân tích một bài toán theo mối quan hệ thừa kế, cần phải tìm ra các đối tượng xuất hiện trong bài toán. Ví dụ dưới đây minh họa việc phân tích để tìm ra các đối tượng này.

Ví dụ 3.4: Giả sử có 2 danh sách như sau:

Danh sách cán bộ

Họ và tên	Ngày sinh	Giới tính	Quê quán	Cấp hàm	Chức vụ
Nguyễn Văn A	30/10/1963	Nam	Hà Nội	Trung tá	Trưởng Khoa
Nguyễn Thị B	22/11/1976	Nữ	Ninh Bình	Đại úy	
....

Danh sách sinh viên

Họ và tên	Ngày sinh	Giới tính	Quê quán	Lớp	Khoa
Trần Văn A	20/05/1995	Nam	Thái Bình	B16D47	CN&ANTT
Lê Thị B	22/11/1997	Nữ	Hải Phòng	B1D50	ANĐT
....

Trong danh sách trên có 2 loại đối tượng là cán bộ và sinh viên, ta sẽ tạo 2 lớp để quản lý 2 loại đối tượng này.

CÁN BỘ	SINH VIÊN
Họ và tên Ngày sinh Giới tính Quê quán Cấp hàm Chức vụ	Họ và tên Ngày sinh Giới tính Quê quán Lớp Khoa
<i>Nhập tên</i> <i>Nhập ngày sinh</i> <i>Nhập giới tính</i> <i>Nhập quê quán</i> <i>Nhập cấp hàm</i> <i>Nhập chức vụ</i> <i>In thông tin cán bộ</i>	<i>Nhập tên</i> <i>Nhập ngày sinh</i> <i>Nhập giới tính</i> <i>Nhập quê quán</i> <i>Nhập lớp</i> <i>Nhập khoa</i> <i>In thông tin sinh viên</i>

Quan sát các lớp trên, ta thấy có những phương thức và thuộc tính chung. Nhóm những thuộc tính và phương thức đó lại và định nghĩa chúng trong 1 lớp gọi là lớp **Người** như sau:

NGƯỜI	
Họ và tên	<i>Nhập tên</i>
Ngày sinh	<i>Nhập ngày sinh</i>
Giới tính	<i>Nhập giới tính</i>
Quê quán	<i>Nhập quê quán</i>
	<i>In thông tin</i>

Hai lớp CÁN BỘ, SINH VIÊN có tất cả những phương thức và thuộc tính của lớp NGƯỜI. Ngoài ra 2 lớp này còn có thêm các thuộc tính và phương thức riêng của chúng. Ví dụ: CÁN BỘ có thêm thuộc tính *Cấp hàm*, *Chức vụ* và phương thức *Nhập cấp hàm*, *Nhập chức vụ*. Như vậy, thay vì định nghĩa 2 lớp CÁN BỘ, SINH VIÊN, ta chỉ cần định nghĩa lớp NGƯỜI với các phương thức và thuộc tính đã chỉ ra ở trên, rồi dùng nó để định nghĩa 2 lớp trên. Khi định nghĩa 2 lớp này, ta chỉ ra nó dựa trên lớp NGƯỜI và khai báo, định nghĩa thêm vào các thuộc tính và phương thức riêng cần thiết cho từng lớp. Để thuận tiện cho việc biểu diễn mối quan hệ giữa 3 lớp trên, nếu dùng ngôn ngữ mô tả thường dài dòng, thay vào đó, người thiết kế bài toán thường sử dụng sơ đồ để hỗ trợ lập trình cho bài toán. Một loại sơ đồ được sử dụng phổ biến là sơ đồ quan hệ đối tượng ORD (được trình bày ở phần tiếp theo).

2. Biểu diễn quan hệ thừa kế

a) Sử dụng sơ đồ quan hệ đối tượng ORD

Khi thiết kế bài toán dùng sơ đồ quan hệ đối tượng cần lưu ý:

- Mỗi lớp được biểu diễn thành một bảng hình chữ nhật như sau:

TÊN LỚP/ Tên Lớp
- Liệt kê thuộc tính
+ Liệt kê các phương thức chính

Hình 3.3. Biểu diễn 1 lớp.

Tên lớp, thuộc tính, phương thức có thể được viết ngắn gọn, thường là tiếng Anh đơn giản hoặc tiếng Việt không dấu. Ví dụ: họ và tên thành *hoten* hoặc *name*, Nhập tên thành `layTen()` hoặc `setname()`, với tên dài hoặc khó phân biệt có thể dùng dấu gạch nối `_` để nối các từ. Ví dụ: *SINHVIEN* hoặc *SINH_VIEN*.

Ví dụ 3.5: Lớp NGUOI có thể được mô tả theo 2 cách như sau:

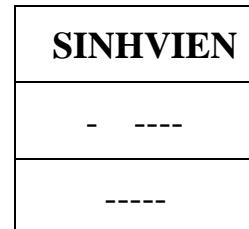
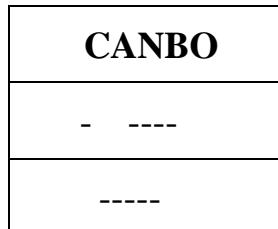
NGUOI	PERSON
- hoten	- name
- ngaysinh	- date
- gioitinh	- gender
- quequan	- hometown
+ nhapTen()	+ getname()
+ nhapNgaysinh()	+ getdate()
+ nhapGioitinh()	+ getgender()
+ nhapQuequan()	+ gethometown()
+ inThongtin()	+ info()

Xây dựng các lớp con CÁN BỘ, SINH VIÊN cũng tương tự như vậy.

- Việc mô tả các quan hệ thừa kế giữa các lớp trong ORD mục đích là để chỉ rõ sự khác biệt giữa các lớp tham gia quan hệ đó: Một lớp con khác lớp cha của nó ở chỗ nào? Các lớp con khác nhau ở chỗ nào?

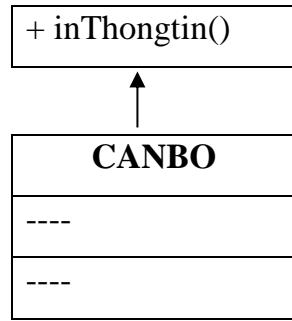
Ví dụ 3.6: Xét mối quan hệ Người – Cán bộ - Sinh viên, từng lớp được biểu diễn như sau:

NGUOI
- hoten
- ngaysinh
- gioitinh
- quequan
+ nhapTen()
+ nhapNgaysinh()
+ nhapGioitinh()
+ nhapQuequan()
+ inThongtin()

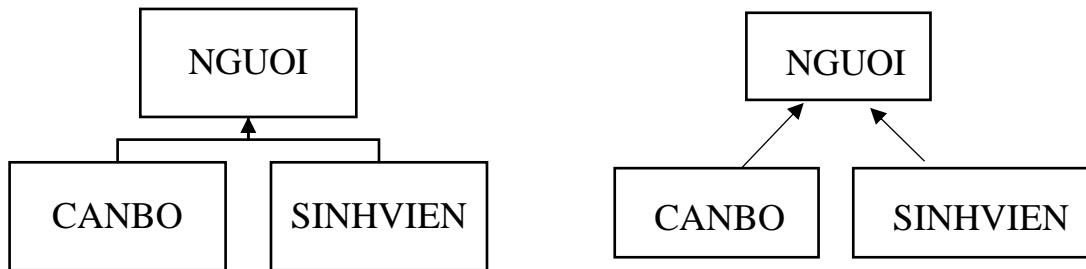


Biểu diễn một quan hệ thừa kế giữa hai lớp bằng một mũi tên trỏ từ lớp con đến lớp cha:

NGUOI
- hoten
- ngaysinh
- gioitinh
- quequan
+ nhapTen()
+ nhapNgaysinh()
+ nhapGioitinh()
+ nhapQuequan()



Có thể biểu diễn quan hệ với nhiều lớp con theo một trong hai kiểu sau:



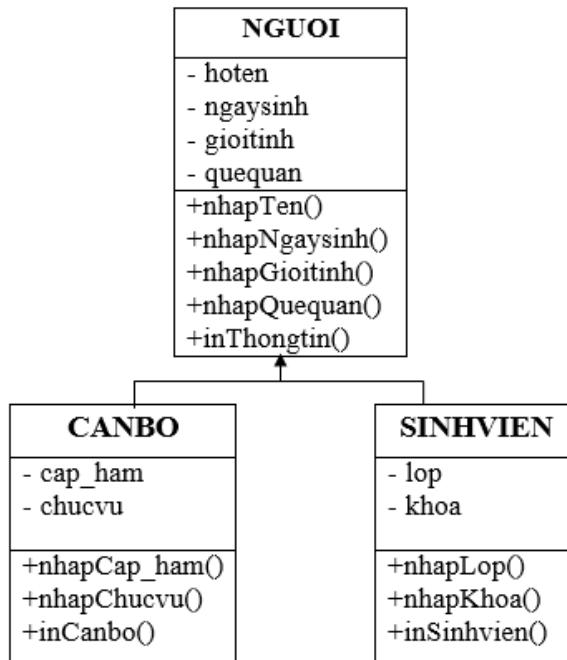
Mô tả cho các lớp con như sau:

CANBO	SINHVIEN
<ul style="list-style-type: none"> - cap_ham - chucvu 	<ul style="list-style-type: none"> - lop - khoa
<ul style="list-style-type: none"> + nhapCap_ham() + nhapChucvu() + inCanbo() 	<ul style="list-style-type: none"> + nhapLop() + nhapKhoa() + inSinhvien()

Biểu diễn sự khác biệt giữa các lớp:

- Nếu không có gì khác nhau thì không cần lập lớp con.
- Khi biểu diễn các thuộc tính và phương thức của các lớp con, chỉ cần liệt kê các thuộc tính/phương thức mà lớp cha không có:
 - Đơn giản hóa sơ đồ, không lặp lại các thuộc tính/phương thức được thừa kế (có thể tìm thấy chúng bằng cách “lần theo mũi tên”).
 - Nhấn mạnh các điểm khác biệt, cho phép dễ dàng nhận ra lý do cho việc lập lớp con.

Với ví dụ trên, sơ đồ ORD, hay còn gọi là cây thừa kế có được là:



Hình 3.4. Cây thừa kế Người - Cán bộ - Sinh viên.

Nhìn vào cây trên, có thể phát biểu như sau:

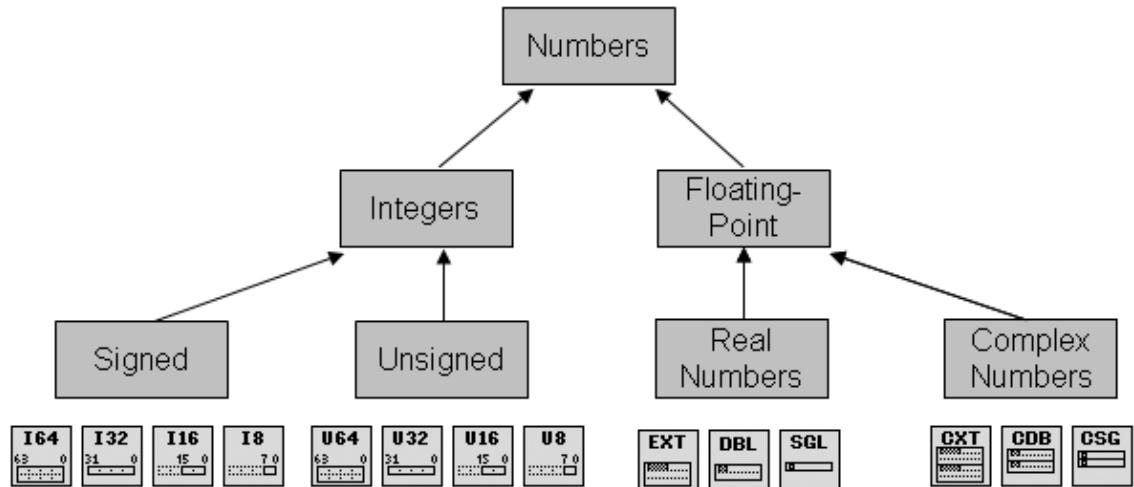
- Mỗi cán bộ đều có các thuộc tính: hoten, ngaysinh, gioitinh, quequan và phương thức nhapTen(), nhapNgaysinh(), nhapGioitinh(), nhapQuequan(), inThongtin(), kèm theo thuộc tính cap_ham, chucvu và phương thức nhapCap_ham(), nhapChucvu(), inCanbo() của riêng lớp đó.

- Phát biểu tương tự với mỗi sinh viên.

Tóm lại, với một bài toán cụ thể, nếu thiết kế theo hướng đối tượng, khi phân tích các lớp để thấy được sự thừa kế giữa các lớp đó, ta phải lần lượt chỉ ra được:

- (1) Mỗi lớp có thuộc tính và phương thức gì phục vụ cho bài toán.
- (2) Có đặc điểm gì chung giữa các lớp.
- (3) Có đặc điểm gì đặc trưng riêng của 1 lớp.
- (4) Lập cây ORD.

Nếu số lượng lớp quá nhiều, sơ đồ ORD sẽ lược bớt phần liệt kê các thuộc tính và phương thức, chỉ giữ lại tên lớp. Ví dụ một số sơ đồ ORD dùng khi thiết kế cây thừa kế với nhiều lớp mô tả các dữ liệu kiểu số như dưới đây:



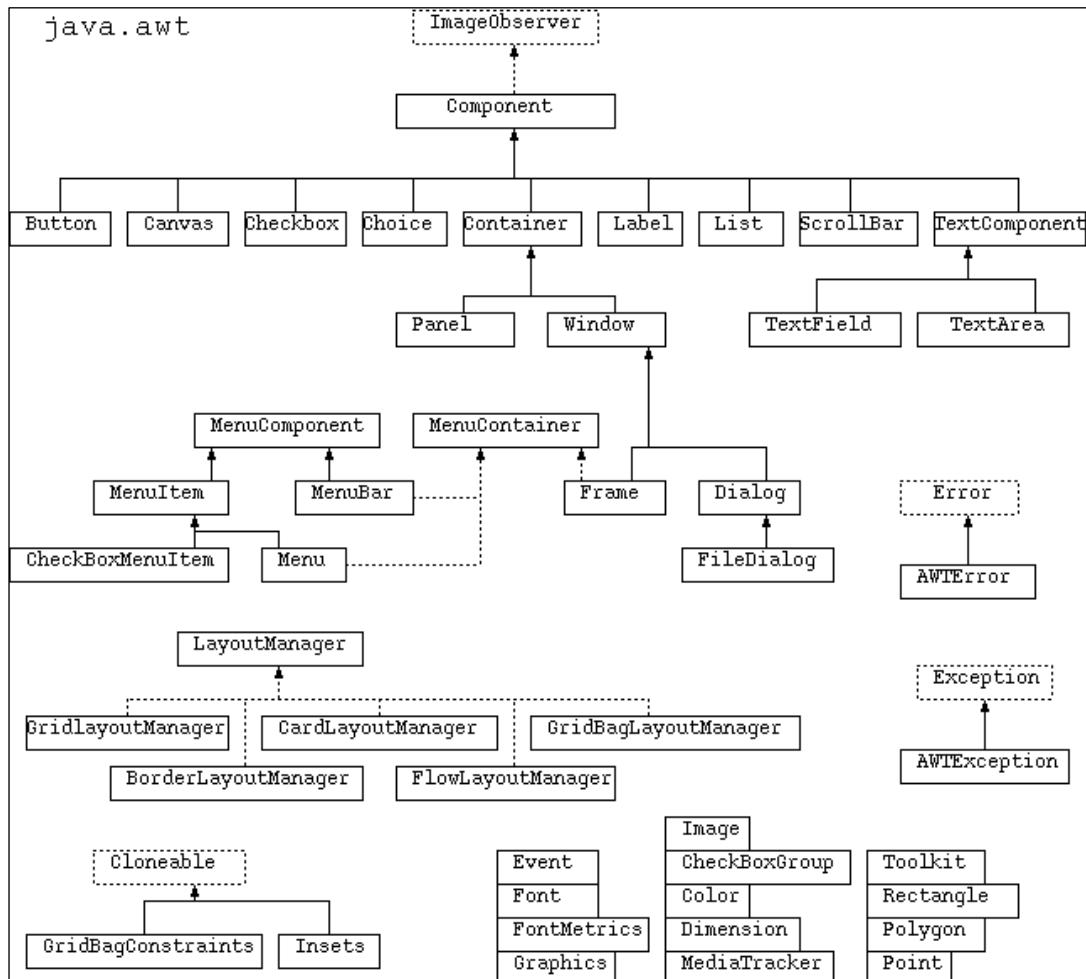
Hình 3.5. Cây thừa kế dữ liệu kiểu số trong Java.

b) Cây thừa kế

Ở phần trên chúng ta đã làm quen với việc sử dụng mô hình ORD để biểu diễn các mối quan hệ thừa kế. Các quan hệ thừa kế luôn được biểu diễn với các lớp con đặt dưới lớp cha để nhấn mạnh bản chất phả hệ của quan hệ. Ta cũng có thể có nhiều tầng thừa kế, tại mỗi tầng, các lớp con tiếp tục thừa kế từ lớp cha. Ví dụ: Một giáo viên là một cán bộ và cũng là một người. Nghĩa là các lớp con được thừa kế các thuộc tính và phương thức của mọi lớp cơ sở bên trên nó. Một giáo viên vì vậy sẽ có thuộc tính và phương thức của cán bộ, của người kèm theo các thuộc tính, phương thức của riêng đôi tượng giáo viên (ví dụ chuyên môn là công nghệ phần mềm, chức danh là giảng viên, trình độ tiến sĩ...). Cách thiết kế nhiều tầng như vậy gọi là cây thừa kế hoặc cây phả hệ. Trong phần này, chủ yếu giới thiệu cây phả hệ. Cách thiết kế cây sẽ được đề cập sâu hơn trong phần thừa kế và đa hình.

Hình 3.5 minh họa cây thừa kế các lớp dữ liệu kiểu số. Việc sử dụng cây thừa kế giúp người sử dụng có cái nhìn tổng quát về một gói dữ liệu, tổng quát về dự án mình đang phát triển hoặc về bài toán mình đang cần làm. Hình 3.6 giúp

người sử dụng nhìn tổng quan được gói AWT của Java gồm có những lớp nào để sử dụng.

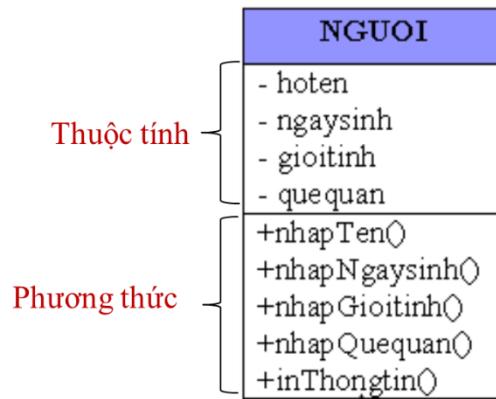


Hình 3.6. Cây thừa kế của gói AWT của Java.

Dụng xong cây thừa kế chúng ta có thể nhìn vào đó để lập trình. Lưu ý trong ngôn ngữ Java một lớp con chỉ có một lớp cha.

3. Cài đặt thừa kế

Trở lại bài toán in danh sách cán bộ và danh sách sinh viên trong ví dụ trên. Sau khi phân tích, lấy ra được một số thuộc tính và hành vi chung, ta đặt nó vào lớp cha – lớp Người (Hình 3.7).



Hình 3.7. Biểu diễn lớp cha.

Để phục vụ cho lập trình, có thể cài đặt thêm các phương thức mới cần thiết, ví dụ như cài đặt thêm các phương thức lấy ra các thuộc tính của một người: layTen(), layNgaysinh(), layGioitinh(), layQuequan() .

Ta có một cài đặt của lớp Người được thực hiện như sau:

Ví dụ 3.7: Cài đặt lớp Người

```

public class Nguoi{
    protected String hoten;
    protected String ngaysinh;
    protected String gioitinh;
    protected String quequan;

    // phuong thuc khai tao
    Nguoi(){
        System.out.println("Khai tao khong tham so lop Nguoi!");
    }
    Nguoi(String _hoten, String _ngaysinh, String _gioitinh,
String _quequan){
        this.hoten = _hoten;
        this.ngaysinh = _ngaysinh;
        this.gioitinh = _gioitinh;
        this.quequan = _quequan;
        System.out.println("Khai tao co tham so cho lop Nguoi!");
    }

    // cac phuong thuc nhap
    public void nhapTen(String _hoten){
        this.hoten = _hoten;
    }
    public void nhapNgaysinh(String _ngaysinh){
        this.ngaysinh = _ngaysinh;
    }
    public void nhapGioitinh(String _gioitinh){
        this.gioitinh = _gioitinh;
    }
    public void nhapQuequan(String _quequan){

```

```

        this.quequan    = _quequan;
    }

    // cac phuong thuc lay thong tin
    public String layTen(){
        return this.hoten;
    }
    public String layNgaysinh(){
        return this.ngaysinh;
    }
    public String layGioitinh(){
        return this.gioitinh;
    }
    public String layQuequan(){
        return this.quequan;
    }

    // in thong tin
    public void inThongtin(){
        System.out.print(this.layTen()+
            " "+this.layNgaysinh()+
            " "+this.layGioitinh()+" "+this.layQuequan());
    }
}

```

Hai lớp con sẽ sử dụng lại những khai báo và cài đặt của lớp *Nguoi* bằng cách thừa kế lại từ lớp này. Như vậy chỉ cần khai báo lớp *Nguoi* một lần mà dùng được cả cho khai báo lớp *Canbo* và khai báo lớp *Sinhvien*.

a) Cài đặt các lớp con

Để báo cho trình biên dịch rằng “lớp B thừa kế từ lớp A” hoặc “lớp B được dẫn xuất từ lớp A” phải thêm vào sau lớp B từ khóa *extends* với cú pháp như sau:

- Cú pháp:

```

public class A {
    .....// Định nghĩa lớp cha
}
public class B extends A{
    .....// Định nghĩa lớp con
}

```

Trong đó, A là lớp cha và B là lớp con được thừa kế từ lớp cha; *extends* là từ khóa thừa kế. Lớp A và lớp B có thể là lớp do người dùng tự định nghĩa hoặc là lớp đã được định nghĩa sẵn trong thư viện của Java. Nếu không có từ khóa *extends*,

ví dụ như định nghĩa lớp A, mặc định A được thừa kế từ lớp *Object* trong thư viện của Java. Lớp *Object* là lớp cha của mọi lớp được định nghĩa trong Java.

Xét bài toán với cây thừa kế đề cập ở phần trước. Chúng ta bắt đầu cài đặt cây thừa kế này trong Java. Đây là một cài đặt minh họa nên chúng ta sẽ không đi sâu phân tích xem cây thừa kế này đã đầy đủ các thuộc tính và phương thức của các lớp đối tượng hay chưa. Thực tế, việc thiết kế đó cũng tùy thuộc vào yêu cầu của từng bài toán cụ thể. Ta bắt đầu từ lớp cha - *Nguoi*. Lớp này định nghĩa người nói chung. Có các thuộc tính: `hoten` để lưu thông tin họ và tên một người, `ngaysinh` lưu ngày sinh, `gioitinh` lưu giới tính, `quequan` lưu quê quán và phương thức `nhapTen()` để nhập họ và tên cho người đó, `nhapNgaysinh()`, `nhapGioitinh()`, `nhapQuequan()` lần lượt để nhập ngày sinh, giới tính, quê quán của người đó. Và `inThongtin()` để in ra thông tin của một người.

Trong cài đặt trên, có hai phương thức: `Nguoi()` và `Nguoi(String _hoten, String _ngaysinh, String _gioitinh, String _quequan)` là hai phương thức khởi tạo. Trong đó, `Nguoi(){}` sẽ khởi tạo 1 đối tượng *Nguoi* không có thuộc tính (là đối tượng rỗng); `Nguoi(String _hoten, String _ngaysinh, String _gioitinh, String _quequan)` sẽ khởi tạo một người có tên, ngày sinh, giới tính và quê quán. Phương thức `public void inThongtin()` in ra thông tin của một người.

Hai lớp con *Canbo* và *Sinhvien* được cài đặt với cú pháp như sau:

```
public class Canbo extends Nguoi{.....}  
public class Sinhvien extends Nguoi{.....}
```

Khi đã thông báo cho trình biên dịch biết lớp *Canbo* và lớp *Sinhvien* được thừa kế từ lớp *Nguoi*, lúc này 2 lớp sẽ có quyền sử dụng lại những gì được thừa kế từ lớp *Nguoi*.

Java chỉ cho phép một lớp con được dẫn xuất từ một lớp cha, tuy nhiên lại cho phép một lớp cha có nhiều lớp con, cũng như cho phép một lớp con là lớp cha của lớp khác. Như vậy, ta có các khai báo đúng như sau:

```

public class Giaovien extends Canbo{.....}
public class Cblop extends Sinhvien{.....}
public class Btruong extends Cbolog{.....}
public class SVchuaTN extends Sinhvien{.....}
public class CuuSV extends Sinhvien{.....}

```

Việc cài đặt các lớp con và sử dụng những gì thừa kế từ lớp cha được thực hiện như sau:

- *Định nghĩa lớp con*: Mô tả một lớp con cũng giống như biểu diễn nó trong ORD, ta chỉ tập trung vào những điểm khác với lớp cha. Điều này mang đến những lợi ích sau:

- + Đơn giản hóa khai báo lớp;
- + Hỗ trợ nguyên lý đóng gói của hướng đối tượng;
- + Hỗ trợ tái sử dụng mã nguồn (sử dụng lại định nghĩa của các phương thức và thuộc tính);
- + Che giấu thông tin ;
- + Tiết kiệm bộ nhớ ;

Như vậy, lớp quản lý đối tượng Cán bộ cần được mô tả các thông tin:

CÁN BỘ	
Họ và tên	<i>Nhập tên</i>
Ngày sinh	<i>Nhập ngày sinh</i>
Giới tính	<i>Nhập giới tính</i>
Quê quán	<i>Nhập quê quán</i>
Cấp hàm	<i>Nhập cấp hàm</i>
Chức vụ	<i>Nhập chức vụ</i>
	<i>In thông tin cán bộ</i>

Hình 3.8. Biểu diễn lớp con.

Lưu ý:

- + Lớp dẫn xuất thừa kế mọi thuộc tính và phương thức của lớp cơ sở nhưng không thừa kế phương thức khởi tạo. Lớp *Canbo* được mô tả trong ORD chỉ giữ lại những thuộc tính/phương thức riêng của nó:

CANBO
- cap_ham
- chucvu
+ nhapCap_ham()
+ nhapChucvu()
+ inCanbo()

Những thuộc tính như họ và tên, ngày sinh... và phương thức như nhập tên, nhập ngày sinh... sẽ được sử dụng lại từ lớp *Nguo*i.

- + Có hai giải pháp gọi phương thức khởi tạo của lớp cơ sở:

- * Sử dụng phương thức khởi tạo mặc định.
- * Gọi phương thức khởi tạo của lớp cơ sở một cách tường minh.

Ví dụ 3.8: Định nghĩa lớp *CanBo* với phương thức khởi tạo:

```
public class CanBo extends Nguoi{  
    private String capHam;  
    private String chucVu;  
    public CanBo() {  
        System.out.println("Khởi tạo không tham số cho lớp  
CanBo!");  
    }  
    public CanBo(String hoTen, String ngaysinh, String gioiTinh,  
String queQuan, String capHam, String chucVu) {  
        this.hoTen = hoTen;  
        this.ngaysinh = ngaysinh;  
        this.gioiTinh = gioiTinh;  
        this.queQuan = queQuan;  
        this.capHam = capHam;  
        this.chucVu = chucVu;  
        System.out.println("Khởi tạo có tham số cho lớp CanBo!");  
    }  
    public String getCapHam() {  
        return capHam;  
    }  
}
```

```

        public String getChucVu() {
            return chucVu;
        }
        public void setCapHam(String capHam) {
            this.capHam = capHam;
        }
        public void setChucVu(String chucVu) {
            this.chucVu = chucVu;
        }
        public void inCanBo() {
            this.inThongTin();
            System.out.println(", " + capHam);
            System.out.println(", " + chucVu);
        }
    }
}

```

Trong đoạn mã khai báo thuộc tính của lớp *CanBo*, chúng ta không khai báo các thuộc tính `hoten`, `gioiTinh`, `queQuan`. Tuy nhiên quan sát hàm khởi tạo có tham số của lớp *CanBo*, ta thấy các đoạn lệnh:

+ Lớp *Canbo* đã sử dụng luôn thuộc tính đã khai báo ở lớp *Nguoi* làm thuộc tính của mình:

```

this.hoten = _hoten;
this.ngaysinh = _ngaysinh;
this.gioitinh = _gioitinh;
this.quequan = _quequan;

```

+ Lớp *Canbo* đã sử dụng luôn phương thức đã định nghĩa ở lớp *Nguoi* trong phương thức của mình:

```

public void inCanbo() {
    this.inThongtin();
    // ...
}

```

Điều này cho thấy lớp *CanBo* có các thuộc tính `hoten`, `gioiTinh`, `queQuan` và có phương thức `inThongTin()`, chúng tỏ các thuộc tính và phương thức này lớp *Canbo* thừa kế từ lớp *Nguoi*.

Giả sử ta khai báo 2 thẻ hiện của lớp *Nguoi* và 2 thẻ hiện của lớp *CanBo* trong hàm *main* như sau:

```

public class Demo {
    public static void main(String []args) {

```

```

        Nguoi    ng1      =      new      Nguoi("Nguyễn Văn
A", "15/10/1970", "Nam", "Hà Nội");
        Nguoi ng2 = new Nguoi();
        CanBo cb1 = new CanBo("Nguyễn Văn A", "15/10/1970", "Nam",
"Hà Nội", "Trung tá", "Trưởng khoa");
        CanBo cb2 = new CanBo();
    }
}

```

Kết quả in ra như sau:

```

Khoi tao co tham so cho lop Nguoi!
Khoi tao khong tham so cho lop Nguoi!
Khoi tao khong tham so cho lop Nguoi!
Khởi tạo có tham số cho lớp CanBo!
Khoi tao khong tham so cho lop Nguoi!
Khởi tạo không tham số cho lớp CanBo!

```

Như vậy có thể thấy:

- + Dòng lệnh khai báo `ng1` đã in ra dòng chữ đầu tiên.
 - + Dòng lệnh khai báo `ng2` đã in ra dòng chữ thứ hai.
 - + Dòng lệnh khai báo `cb1` đã in ra 2 dòng chữ thứ 3 và thứ 4
 - + Dòng lệnh khai báo `cb2` đã in ra 2 dòng chữ thứ 5 và thứ 6
 - + Dòng chữ thứ 2, thứ 3 và thứ 5 đều gọi khởi tạo không tham số `Nguoi()`
- từ các phương thức khởi tạo của của `cb1`, `cb2`

Như vậy khi gọi phương thức khởi tạo của lớp con, ta đã gọi ngầm định phương thức khởi tạo của lớp cha. Khi tạo một đối tượng của lớp con, đối tượng này chứa trong nó một *đối tượng con* của lớp cha. Đối tượng con này được tạo giống như câu lệnh `Nguoi subobject = new Nguoi()`. Ta gọi trường hợp này là *lớp cha đã khởi tạo bên ngoài lớp cha* (vì lớp cha đã khởi tạo thể hiện có tính chất của lớp cha trong lớp con). Vì phương thức khởi tạo của lớp cha xảy ra trước khi lớp con có thể truy nhập được vào nó nên lớp con không thừa kế được gì từ phương thức khởi tạo của lớp cha. Trường hợp này biểu diễn tình huống gọi là phương thức khởi tạo mặc định của lớp cơ sở.

Lưu ý: Phương thức khởi tạo của lớp con không thừa kế được phương thức khởi tạo của lớp cha. Ví dụ sau sẽ giải thích điều đó.

Ví dụ 3.9: Giả sử bỏ qua phương thức khởi tạo *Canbo()*

```
public class Canbo extends Nguoi{
    // code
    /* Canbo(){System.out.println("Khai khong tham so cho cac
can bo!");} */
    Canbo(String _hoten, String _ngaysinh, String _gioitinh,
String _quequan, String _cap_ham, String _chucvu){
        // code
    }
    // code
}
```

Lúc này, chương trình sẽ báo lỗi ở khai báo `Canbo cb1 = new Canbo();`. Như vậy, rõ ràng lớp *Canbo* không thể sử dụng được phương thức khởi tạo không tham số của lớp cha. Phương thức khởi tạo của lớp cha chỉ có thể được gọi từ phương thức khởi tạo của lớp con và lớp con không thừa kế được phương thức khởi tạo từ lớp cha.

Ví dụ 3.10: Với phương thức khởi tạo thứ 2 của lớp *Canbo*

```
Canbo(String _hoten, String _ngaysinh, String _gioitinh, String
_quequan, String _cap_ham, String _chucvu){
    this.hoten = _hoten;
    this.ngaysinh = _ngaysinh;
    this.gioitinh = _gioitinh;
    this.quequan = _quequan;
    this.cap_ham = _cap_ham;
    this.chucvu = _chucvu;
    System.out.println("Khai tao co tham so cho cac can bo!");
}
```

Trong ví dụ trên lập trình được vì thuộc tính lớp *Nguoi* đặt quyền truy nhập là *protected*, lớp *CanBo* có thể thừa kế và truy cập trực tiếp được các thuộc tính này. Tuy nhiên, nếu đặt quyền truy cập *private* cho các thuộc tính này của lớp *Nguoi* để đảm bảo tuyệt đối tính đóng gói và che giấu thông tin, khi đó lớp *CanBo* sẽ không thể truy cập trực tiếp được các thuộc tính *private* này. Điều này đồng nghĩa với việc ta không thể viết hàm khởi tạo có tham số của lớp *CanBo* như đã viết ở trên. Khi đó, lớp con phải sử dụng từ khóa *super* gọi tham chiếu hàm khởi tạo của lớp cha

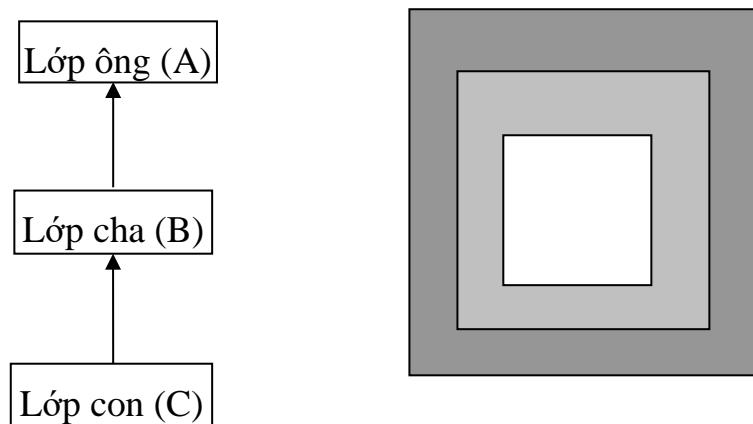
một cách tường minh. Lưu ý: *super* phải viết ở dòng đầu tiên trong thân phương thức khởi tạo.

Ví dụ 3.11: Ví dụ sử dụng *super*

```
public CanBo(String hoTen, String ngaysinh, String gioiTinh,  
String queQuan, String capHam, String chucVu) {  
    super(hoTen, ngaysinh, gioiTinh, queQuan);  
    this.capHam = capHam;  
    this.chucVu = chucVu;  
    System.out.println("Khởi tạo có tham số cho lớp CanBo!");  
}
```

Trong đó, *extends* là từ khóa thừa kế, *super* là từ khóa tham chiếu tới đối tượng của lớp cha. Với lớp *Sinhvien*, ta viết tương tự.

Tóm lại, khi tạo thể hiện của một lớp dẫn xuất, các phương thức khởi tạo được gọi bắt đầu từ lớp cơ sở nhất và chuyển dần về phía lớp dẫn xuất (Hình 3.9).



Hình 3.9. Trình tự gọi constructor khi có thừa kế.

Ngoài phương thức khởi tạo, lớp con có thêm các phương thức của riêng chúng, với các phương thức này khai báo và cài đặt như các phương thức của lớp thông thường.

Chương trình sau minh họa một cài đặt đầy đủ của lớp *Sinhvien*

Ví dụ 3.12:

```
public class Sinhvien extends Nguoi{
```

```

        private String lop;
        private String khoa;

        Sinhvien(){
            System.out.println("Khoi tao khong tham so cho cac sinh
vien!");
        }
        Sinhvien(String _hoten, String _ngaysinh, String _gioitinh,
String _quequan, String _lop, String _khoa){
            super(_hoten, _ngaysinh, _gioitinh, _quequan);
            this.lop = _lop;
            this.khoa = _khoa;
            System.out.println("Khoi tao co tham so cho cac sinh
vien!");
        }

        public void nhapLop(String _lop){
            this.lop = _lop;
        }
        public void nhapKhoa(String _khoa){
            this.khoa = _khoa;
        }

        public String layLop(){
            return this.lop;
        }
        public String layKhoa(){
            return this.khoa;
        }

        public void inSinhvien(){
            this.inThongtin();
            System.out.println(""+this.layLop()+" "+this.layKhoa());
        }
    }
}

```

Vậy với thiết kế thừa kế, lớp *CanBo* và lớp *SinhVien* đã sử dụng lại được thuộc tính/phương thức của lớp cha *Nguoi*, nhưng không thừa kế được phương thức khởi tạo.

Sau khi cài đặt xong lớp này, hàm *main* sử dụng các lớp đã cài đặt để tạo danh sách ở trên (bỏ qua các cài đặt về tạo bảng dữ liệu).

Ví dụ 3.13:

```

public static void main(String []args){
    // 2 the hien lop Canbo
    Canbo cb1 = new Canbo("Nguyen Van A", "30/10/1963", "Nam", "Ha
Noi", "Trung ta", "Truong Khoa");
    Canbo cb2 = new Canbo();
    cb2.nhapTen("Nguyen Thi B");
    cb2.nhapNgaysinh("22/11/1976");
    cb2.nhapGioitinh("Nu");
}

```

```

cb2.nhapQuequan("Ninh Binh");
cb2.nhapCap_ham("Dai uy");
cb2.nhapChucvu("Khong");
// in thong tin
System.out.println("Danh sach can bo la:");
cb1.inCanBo();
cb2.inCanBo();
// 2 the hien lop Sinhvien
Sinhvien sv1 = new Sinhvien("Tran Van A",
"20/05/1990", "Nam", "Thai Binh", "B16D47", "CN&ANTT");
Sinhvien sv2 = new Sinhvien();
sv2.nhapTen("Le Thi B");
sv2.nhapNgaysinh("22/11/1991");
sv2.nhapGioitinh("Nu");
sv2.nhapQuequan("Hai Phong");
sv2.nhapLop("B1D50");
sv2.nhapKhoa("ANĐT");
// in thong tin
System.out.println("Danh sach sinh vien la:");
sv1.inSinhvien();
sv2.inSinhvien();

```

```

Nguyen Van A 11/12/1990 Nam Ha Tay Trung uy To truong
Nguyen Thi B 11/12/1990 Nu Ninh Binh thieu uy Giao vien
Tran Van A 11/12/1990 Nam Ha Tay B16D47 CN&ANTT
Le Thi B 11/12/1990 Nu Ninh Binh B1D50 ANĐT
BUILD SUCCESSFUL (total time: 0 seconds)

```

Hình 3.10. In danh sách cán bộ và danh sách sinh viên.

Khi thiết kế thừa kế, quyền truy cập cũng góp phần quyết định đến thuộc tính và phương thức nào của lớp cha mà lớp con có thể truy cập được. Để thấy rõ vai trò của quyền truy cập trong quan hệ thừa kế, chúng ta phân tích từng quyền truy cập: *public*, *private* và *protected*.

+ ***public***: Với quyền này, mọi đối tượng khác có thể truy cập tới thành phần được khai báo là *public*. Vậy trong lớp cha, thuộc tính hay phương thức nào được khai báo là *public* thì không chỉ lớp con mà mọi lớp khác đều truy cập được.

+ ***private***: Thuộc tính hay phương thức có kiểu *private* chỉ được truy cập ở trong lớp đó. Như ví dụ trên, thuộc tính của lớp *CanBo* thì chỉ lớp *CanBo* có thể truy cập được. Như vậy, với các thuộc tính ở lớp cha mà ta không muốn cho lớp con truy cập trực tiếp thì nên khai báo là *private*. Các lớp con chỉ có thể truy cập tới các thuộc tính này thông qua 2 loại phương thức: Phương thức Truy vấn, có

dạng tên dạng *get...* hoặc *lay...* để lấy ra dữ liệu của thuộc tính; Phương thức Cập nhật, có dạng *set...* hoặc *nhap...* để gán dữ liệu mới cho thuộc tính như đã đề cập ở Chương 2.

+ ***protected***: Trong hợp nếu ta xây dựng lớp cha và muốn "cấp" cho lớp con quyền truy nhập tới một số thuộc tính/phương thức mà vẫn đảm bảo tính đóng gói, ta sử dụng từ khoá *protected*. Từ khóa này quy định quyền truy nhập tới thuộc tính/phương thức của lớp cha cho các lớp con và chỉ ra các lớp con của lớp đó. Đối với mọi đối tượng khác của Java, các thuộc tính/phương thức *protected* được coi như *private* (chúng đều không thể truy nhập được). Mặc dù theo tiêu chí đóng gói thì nên để mọi thứ là *private*, nhưng khi tạo các cây thừa kế, ta thường hay cấp quyền truy nhập *protected*.

Quyền truy cập các thuộc tính của lớp *Nguoi* để là *protected* trên lớp *CanBo* mới có thể khai báo phương thức khởi tạo hợp lệ như sau:

```
public CanBo(String hoTen, String gioiTinh, String queQuan,
String capHam, String chucVu) {
    this.hoTen = hoTen;
    this.gioiTinh = gioiTinh;
    this.queQuan = queQuan;
    this.capHam = capHam;
    this.chucVu = chucVu;
    System.out.println("Khởi tạo có tham số cho lớp CanBo!");
}
```

Tương tự như vậy, nếu lớp *Nguoi* khai báo phương thức *setHoTen()* là *protected* thì chỉ có lớp con *CanBo*, *SinhVien* và các lớp con của lớp *CanBo*, *SinhVien* mới có thể truy cập được, còn các lớp độc lập thì không thể.

Bảng 3.1 Tổng hợp các mức kiểm soát truy cập:

Mức truy cập	Trong cùng lớp	Lớp con	Bên ngoài
public	Có	Có	Có
protected	Có	Có	
private	Có		

Bảng 3.1. Bảng tổng hợp các quyền truy cập giữa các lớp.

b) Hiện tượng ẩn danh

Java làm xuất hiện hiện tượng ẩn danh liên quan đến cài đặt các phương thức trùng tên ở lớp cha và lớp con. Nếu như lớp cha có một phương thức nào đó được "nạp chồng" (*overload*) nhiều lần, việc định nghĩa lại tên phương thức đó trong lớp con sẽ không làm "ẩn" đi bất kì phương thức nào của lớp cha. Vì vậy phần nạp chồng sẽ làm việc mà không phải quan tâm xem phương thức này đã được định nghĩa ở lớp cha hay chưa. Nội dung về nạp chồng sẽ tiếp tục được đề cập trong các phần sau. Để làm rõ hơn hiện tượng ẩn danh, ta xét ví dụ sau:

Ví dụ 3.14: Quan sát các phương thức *doh* dưới đây

```
public class Homer {
    char doh(char c) {
        System.out.println("char");
        return 'd';
    }
    float doh(float f) {
        System.out.println("float");
        return 1.0f;
    }
}
public class Milhouse {}
public class Bart extends Homer {
    void doh(Milhouse m) {}
}
public class Hide {
    public static void main(String args[]) {
        Bart b = new Bart();
        b.doh(1); // doh(float) used
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
}
```

Kết quả chạy chương trình

```

class Hide {
    public static void main(String args[]) {
        Bart b = new Bart();
        b.doh(1); // doh(float) used
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
}

```

Output - Vdu (run)

- run:
- float
- char
- float

Hình 3.11. Kết quả chạy chương trình với ẩn danh.

Trong ví dụ trên, *Bart* là lớp con của lớp *Homer*. *b* là một thể hiện của lớp *Bart*. Câu lệnh *b.doh(1)* sử dụng phương thức `float doh (float f)` của *Homer*; *b.doh('x')* sử dụng `char doh(char c)` của *Homer*; *b.doh(1.0f)* cũng sử dụng `float doh(float f)`; còn *b.doh(new Milhouse())* sử dụng `void doh(Milhouse m)` của *Bart*. Như vậy, phương thức cùng tên của lớp cha với lớp con đều có vai trò bình đẳng, phương thức ở lớp con không che mất phương thức ở lớp cha. Trong phần tiếp theo khi nói về vượt quyền và nạp chồng (*overriding* và *overloading*) sẽ giải thích kỹ hơn vấn đề phương thức trùng tên làm việc như thế nào.

Tóm lại, chúng ta có thể định nghĩa lại các phương thức của lớp cơ sở sao cho:

- + Đối tượng của lớp dẫn xuất sẽ hoạt động với phương thức mới phù hợp với nó.
- + Có thể tái sử dụng phương thức cùng tên của lớp cơ sở bằng từ khóa *super*.

Ví dụ 3.15: Xây dựng lớp *Giaovien* thừa kế lớp *Nguoi*

```

/*Nguoi.java*/
public class Nguoi {
    // code
    public Nguoi() {}
    // lay ra ten
}

```

```

        public String layTen() {
            return this.name;
        }
    }
/*Giaovien.java*/
public class Giaovien extends Nguoi{
    // code
    public String layTen() {
        System.out.println("Ten giao vien!");
        return super.layTen();
    }
    public static void main(String args[]) {
        Giaovien frd = new Giaovien();
        frd.layTen();
    }
}

```

Giả sử lớp *Giaovien* nằm khác gói với lớp *Nguoi*, *Giaovien* thừa kế được lớp *Nguoi* thì cần lập trình:

- + Một hàm khởi tạo của *Nguoi* để là *public*. Trong ví dụ trên đã khai báo `public Nguoi () {}.`
- + Phương thức *layTen()* được định nghĩa lại ở lớp *Giaovien* phải cùng kiểu với *layTen()* của *Nguoi*. Trong ví dụ trên cùng trả về kiểu *String*.
- + Quyền của *layTen()* ở *Nguoi* là *public* thì quyền của *layTen()* ở *Giaovien* cũng phải là *public*.

Ngoài ra, trong ví dụ phương thức `public String layTen()` của lớp *Giaovien* còn sử dụng lại phương thức *layTen()* của lớp *Nguoi* thông qua câu lệnh `return super.layTen();`

Tóm lại, nếu ta muốn định nghĩa lại phương thức ở lớp con thì:

- + Phải có quyền truy cập không chặt hơn phương thức được định nghĩa lại.
- + Phải có kiểu giá trị trả lại như nhau.

c) Sự chuyển kiểu đối tượng với upcasting và downcasting

- Chuyển kiểu lên (*upcast*)

Xem xét cây thừa kế *Nguoi* - *CanBo* - *SinhVien*, các phát biểu sau là đúng: “*Mọi đối tượng CanBo đều là Nguoi. Mọi đối tượng SinhVien đều là Nguoi*”.

Nhưng phát biểu ngược lại: “*Mọi đối tượng Nguoi đều là CanBo. Mọi đối tượng Nguoi đều là SinhVien*” thì không đúng. Bởi vì thuộc tính của các lớp con bao gồm các thuộc tính và phương thức được thừa kế từ lớp cha cùng một số thuộc tính và phương thức của riêng nó. Như vậy, với một thể hiện của lớp *CanBo* hay lớp *SinhVien* đều có quyền truy nhập các thuộc tính và phương thức không có mức truy nhập riêng tư của lớp *Nguoi*. Hay nói cách khác, ta có thể đổi xử với một thể hiện của lớp *CanBo*, lớp *SinhVien* như với một thể hiện của lớp *Nguoi*. Hiện tượng các thể hiện của lớp con được đổi xử như một thể hiện của lớp cha được gọi là hiện tượng chuyển kiểu lên (*upcast*).

Lưu ý:

- + *Upcast* thường được dùng trong trường hợp đổi tượng của lớp con làm đổi số đầu vào cho các phương thức khai báo tham số có kiểu là lớp cha. *Upcast* thường gặp tại các lời gọi phương thức, khi trong định nghĩa phương thức một biến tham chiếu đến lớp cha được yêu cầu, nhưng biến tham chiếu đến lớp con cũng được chấp nhận.
- + *Upcast* không cần đổi kiểu tường minh.
- + Nếu ta dùng câu lệnh gán 1 đối tượng của lớp con cho đối tượng của lớp cha, trình biên dịch vẫn coi đối tượng của lớp cha là của lớp cha, không thể coi là đối tượng của lớp con. Vì thế câu lệnh `Nguoi ng2 = cb1;` hoàn toàn hợp lệ, nhưng lệnh `ng2.setCapHam();` bị báo lỗi.
- + Với câu lệnh gán trên, `cb1` không hề bị suy biến thành một đối tượng của lớp *Nguoi*, nó vẫn là đối tượng thuộc lớp *CanBo*. Bản chất thì vẫn vậy, chỉ có cách nhìn nhận là khác đi, vì thế câu lệnh sau vẫn hợp lệ: `cb1.setCapHam();`

Ví dụ 3.16: Thêm một ví dụ về *upcast* trong truyền tham số

```
import java.util.*;  
  
class Instrument {  
    public void play() {  
    }  
}
```

```

        static void tune(Instrument i) {
            i.play();
        }
    }
    class Wind extends Instrument {
        public static void main(String args[])
        {
            Wind flute = new Wind();
            Instrument.tune(flute); // Upcasting
        }
    }

```

- Chuyển kiểu xuống (downcast)

Upcast là quá trình tương tác với một thê hiện của lớp dẫn xuất như thê nó là thê hiện của lớp cơ sở. *Downcast* (chuyển kiểu xuống) là quy trình ngược lại: tương tác với với một thê hiện của lớp cơ sở như thê nó là thê hiện của lớp dẫn xuất.

Lưu ý:

- + *Downcast* là quy trình rắc rối hơn và có nhiều điểm không an toàn.
- + *Downcast* luôn đòi hỏi đổi kiểu tường minh (*explicit type cast*).

Để giải thích rõ hơn việc *downcast* không an toàn, ta phân tích đoạn lệnh sau:

```

CanBo cb1 = new CanBo();
Nguoi ng1 = cb1; //upcast
CanBo cb2 = (CanBo)ng1; //downcast
SinhVien sv = (SinhVien)ng1; //downcast

```

Đoạn lệnh trên sẽ báo lỗi ở dòng lệnh thứ 4 với lỗi chuyển kiểu. Do *ng1* là biến tham chiếu đến một đối tượng của lớp *CanBo* nhưng lại bị ép kiểu tham chiếu đến một đối tượng của lớp *SinhVien*. Đoạn lệnh thứ 3 không bị báo lỗi, vì *ng1* đúng là biến tham chiếu đến một đối tượng của lớp *CanBo*.

Như vậy sử dụng *Upcasting* thì an toàn còn *Downcasting* thì không an toàn. Do đó nên thận trọng và hạn chế dùng *downcasting*.

Lưu ý: Để khắc phục hiện tượng chương trình dễ bị đỗ vỡ do *downcast* gây ra, ta sử dụng toán tử *instanceof* để kiểm tra một thê hiện thuộc lớp nào:

Ví dụ 3.17: Cách sử dụng *instanceof*

```

public doSomething(CanBo e) {
    if (e instanceof Nhanvien) {...}
    else if (e instanceof SinhVien) {...}
    else {...}
}

```

II. ĐA HÌNH

1. Các loại đa hình

Có 3 loại đa hình:

- *Overloading*: đa hình nạp chồng.
- *Parametric*: đa hình tham số.
- *Overriding*: đa hình vượt quyền/ghi đè.

Tuy nhiên không phải lúc nào cũng có thể phân biệt được 3 loại trên một cách rõ ràng.

+ *Overloading* là loại đa hình cho phép các hàm có thể trùng tên nhau, thực hiện các chức năng giống nhau trong các lớp độc lập với nhau (có thể các lớp này không có quan hệ thừa kế). Ví dụ: Lớp *Nguoi* có phương thức *inThongtin()* in ra thông tin của 1 người và lớp *Lop* cũng có phương thức *inThongtin()* để in thông tin của một lớp học. Từ đó, đa hình *Overloading* cho phép chúng ta có thể định nghĩa một toán tử nào đó có phương thức cụ thể như thế nào là phụ thuộc vào các toán hạng áp dụng cho nó. Ví dụ với phương thức của thao tác cộng hai giá trị $a+b$ sẽ có các cách hoạt động khác nhau tùy thuộc vào a và b là số (thì tính phép cộng số học) hay là xâu (thì thực hiện nối xâu). Minh họa cụ thể nhất các em có thể thấy qua câu lệnh `System.out.println("In ra 1 so:" + 3)` thực hiện phép nối xâu *In ra 1 so 3* nhưng `System.out.println(2+3)` in ra kết quả $2+3 = 5$

+ *Parametric*: Là loại đa hình cho phép các hàm cùng tên, nhưng có danh sách tham số truyền vào khác nhau (về tên, kiểu, thứ tự tham số). Khi gọi tên một hàm và truyền giá trị của các đối số cho hàm đó, đa hình *parametric* sẽ làm cho trình biên dịch có thể chọn được phương thức phù hợp. Vì thế, chúng ta có thể

định nghĩa phương thức `cong()` với cùng một tên đó nhưng có thể tính tổng các giá trị như sau:

- * `int cong(int, int)`: tính tổng của 2 số nguyên kiểu int.
- * `float cong(float, float)`: tính tổng của 2 số thực kiểu float.
- * `char cong(char, char)`: tính tổng của 2 ký tự, phép tính này sẽ cho kết quả tùy thuộc vào người lập trình.

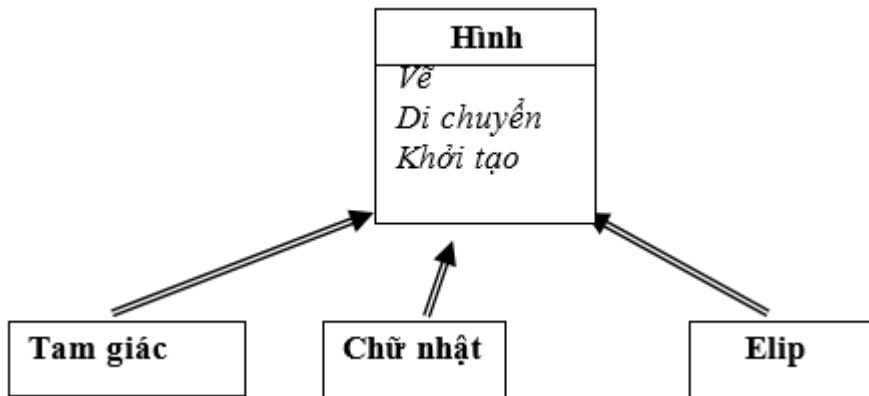
Như vậy, lúc này một chữ kí (*signature*) của hàm xác định duy nhất của hàm, sẽ là tên và kiểu được xác định bởi giá trị của các đối số khi truyền vào cho hàm. Chữ kí của hàm sẽ xác định xem trong một loạt các hàm trùng tên, hàm nào sẽ được sử dụng.

+ *Overriding*: Đây là khả năng định nghĩa lại một phương thức của lớp cha tại các lớp con, vì thế người dùng có thể gọi phương thức của một đối tượng mà không cần biết kiểu của nó là gì. Loại đa hình này cũng tạo ra sự khác biệt giữa các lớp trong một họ các đối tượng (cha khác con, anh em khác nhau) dù chúng có giao diện chung (đó là cùng cha, nên có thuộc tính và phương thức chung).

Xét bài toán cờ với các lớp đối tượng *King* (Tướng), *Queen* (Hậu), *Bishop* (Tượng), *Knight* (Mã), *Rook* (Xe), *Pawn* (Tốt) đều được thừa kế từ đối tượng *Piece* (Quân cờ). Các đối tượng này cùng có phương thức *movement* (dịch chuyển) nhưng có cách di chuyển hoàn toàn khác nhau tùy thuộc vào đối tượng nào được gọi. Vì thế nếu chương trình có thực hiện một phương thức *Piece.movement* sẽ hoạt động tùy xem quân cờ đó là quân nào.

Để tìm hiểu kĩ hơn về đa hình, chúng ta sẽ phân tích ví dụ dưới đây.

Ví dụ 3.18: Cho một cây thừa kế như sau:



Hình 3.12. Cây thừa kế hình học.

Các hình đều có phương thức *Vẽ*, *Di Chuyển* và *Khởi tạo*. Nhưng với các lớp con *Tam giác*, *Chữ nhật*, *Elip* sẽ có cách vẽ, di chuyển và khởi tạo khác nhau. Ví dụ lớp *Tam giác* cần phải vẽ 3 đường thẳng gập nhau tại 3 đỉnh, hình *Elip* lại vẽ 1 đường cong khép kín. Như vậy, có thể thấy đa hình đầy mạnh việc đóng gói dữ liệu: người dùng chỉ biết phương thức vẽ của lớp *Hình*, còn phương thức này thực thi ra sao trong các trường hợp khác nhau thì người dùng không được biết.

2. Liên kết phương thức

a) *Hoạt động của overloading và overriding*

Hai loại đa hình thể hiện rõ nhất mối quan hệ thừa kế: đó là *Overloading* (*nạp chồng*) và *Overriding* (*vượt quyền hoặc đè*).

- *Nạp chồng (Overloading)*: Đã biết hai hoặc nhiều hàm hoặc phương thức trùng tên với các chữ ký khác nhau được gọi là *Overloading-nạp chồng*. Trong lập trình hướng đối tượng, loại đa hình này được gọi là *phương thức chồng - method Overloading*: hai phương thức trùng tên nhưng danh sách tham số khác nhau.

Ví dụ 3.19: Trong lớp *Nguoi* phương thức khởi tạo là 2 phương thức trùng tên

```
public class Nguoi{  
    protected static int x = 0;  
    protected String hoten;
```

```

protected String ngaysinh;
protected String gioitinh;
protected String quequan;
// phuong thuc khai tao
Nguoi(){System.out.println("Khai tao khong tham so cho lop
Nguoi!");
}
Nguoi(String _hoten, String _ngaysinh, String _gioitinh,
String _quequan){
    //Nguoi.x++;
    this.hoten = _hoten;
    this.ngaysinh = _ngaysinh;
    this.gioitinh = _gioitinh;
    this.quequan = _quequan;

    System.out.println("Khai tao co tham so cho lop Nguoi!");
}
// code tiếp
}

```

Nhưng khả năng của đa hình nạp chồng phương thức còn phát triển hơn ở chỗ ta có thể đứng trong định nghĩa của một phương thức này, gọi tới phương thức nạp chồng với phương thức đó.

Ví dụ 3.20: Giả sử lớp *Nguoi* viết 2 phương thức *tang()*

```

public class Nguoi{
    protected static int x = 0;
    // code
    int tang(int amount) {
        x = x + amount;
        return count;
    }
    int tang() {
        return tang(1);
    }
}

```

Trong định nghĩa phương thức *tang()* có một lời gọi hàm tới phương thức *tang(int amount)*. Trình biên dịch vẫn hiểu rằng đây là hai phương thức khác nhau, chứ không phải gọi đệ quy.

Ứng dụng của nạp chồng phương thức là để cung cấp nhiều thông tin gộp lại từ nhiều nguồn khác nhau.

Ví dụ 3.21: Viết 2 phương thức *getInfo()* thực hiện 3 thông điệp in ra màn hình khác nhau

```

void getInfo(String name, String date, String address ) {
    // in tên, ngày sinh, quê quán
    System.out.println("Ho ten:"+name+" Ngay sinh:"+date+" Dia
chi:"+address);
}
void getInfo(String class, String faculty) {
    // cung cấp lớp học, ngành học
    System.out.println("Lop:"+class+" Khoa:"+faculty);
}
void getInfo( Nguoi a, School s, String message) {
    System.out.println(message + ": ");
    printInfo(a.getTen(), a.getDate(), a.getAddress());
    printInfo(s.getClassName(), s.getFaculty());
}

```

Hơn thế nữa, trong một lớp, phương thức khởi tạo cũng có thể nạp chồng.

Ví dụ 3.22: Hai phương thức khởi tạo của lớp *Nguoi*

```

public class Nguoi{
    protected static int x = 0;
    protected String hoten;
    protected String ngaysinh;
    protected String gioitinh;
    protected String quequan;
    // phuong thuc khai tao
    Nguoi(){
        this("", "", "", "");
        System.out.println("Khai tao khong tham so cho lop Nguoi!");
        Nguoi.x++;
    }
    Nguoi(String _hoten, String _ngaysinh, String _gioitinh,
String _quequan){
        Nguoi.x++;
        this.hoten = _hoten;
        this.ngaysinh = _ngaysinh;
        this.gioitinh = _gioitinh;
        this.quequan = _quequan;
        System.out.println("Khai tao co tham so cho lop Nguoi!");
    }
}

```

Chú ý rằng phương thức khởi tạo thứ 2 sử dụng tới từ khóa *this*. Từ khóa này là một tham số ẩn của phương thức mang ý nghĩa rằng đối tượng đang được khởi tạo. Phương thức *Nguoi()* sẽ gọi đến một phương thức khởi tạo khác của cùng lớp *Nguoi*. Như ví dụ trên *this("", "", "", "")* ; sẽ gọi đến phương thức khởi tạo *Nguoi(String _hoten, String _ngaysinh, String _gioitinh, String _quequan)*. Lưu ý rằng lời gọi tương tự như *this("", "", "", "")* ; luôn phải nằm ở câu lệnh đầu tiên của phương thức khởi tạo.

Lợi ích của nạp chồng phương thức khởi tạo đó là cung cấp các giá trị mặc định cho các tham số bị mất. Như ví dụ trên `Nguoi()` không có các tham số mặc định sẽ gán các thuộc tính của người đó là các xâu rỗng "". Làm như vậy, một người được khởi tạo luôn có giá trị.

Lợi ích của việc nạp chồng các phương thức còn ở chúng ta có thể làm "cùng một việc" với nhiều kiểu dữ liệu khác nhau.

Ví dụ 3.23: `inThongtin()` đã nạp chồng tại lớp `Student` và `Professor`. Cùng một thông điệp `inThongtin()` đã được “hiểu” theo 2 cách khác nhau

```
class Student extends Nguoi {
    void inThongtin(){
        inThongtinnguoi();
        inHocvan();
    }
}
class Professor extends Nguoi() {
    void inThongtin() {
        inThongtinnguoi();
        inLinhvucnghienuu();
    }
}
```

Các phương thức `print` và `println` của Java được nạp chồng rất nhiều lần. Phương thức `print` được nạp chồng 7 lần với 7 kiểu dữ liệu đầu vào khác nhau: `Object`, `char`, `char[]`, `int`, `float` v.v.. Vì thế chúng ta hoàn toàn có thể dùng `print` để in kí tự hoặc số nguyên, số thực được.

Tuy nhiên có một vài chú ý trong Java khi nạp chồng:

+ Gán hợp lệ: Nếu thực hiện phép gán `a=b` thì kiểu của `b` phải không nhỏ hơn kiểu của `a` (nhỏ theo nghĩa vùng nhớ được cấp phát và miền giá trị). Nếu kiểu của `b` lớn hơn kiểu của `a` thì ta phải ép kiểu tường minh: `a = (kiểu của a) b`.

Ví dụ 3.24: Ép kiểu tường minh, gán hợp lệ

```
class Test {
    public static void main(String args[]) {
        double d;
        int i;
        d = 5; // hợp lệ
```

```

        i = 3.5;           // không hợp lệ
        i = (int) 3.5;     // hợp lệ
    }
}

```

+ Gọi phương thức hợp lệ:

Ví dụ 3.25: Gọi phương thức hợp lệ

```

class Test {
    public static void main(String args[]) {
        in(5); // hợp lệ
    }
    static void in(double d) {

        System.out.println(d);
    }
}

```

Kết quả in ra là: 5.0. Gọi phương thức `in(5)` hợp lệ vì nó tương đương với phép gán `double d = 5` ở trên. Tuy nhiên lời gọi sau sẽ không hợp lệ

Ví dụ 3.26: Gọi phương thức không hợp lệ

```

class Test {
    public static void main(String args[]) {
        in(5.0); // không hợp lệ
    }
    static void in(int i) {
        System.out.println(i);
    }
}

```

`in(int)` không thể áp dụng được cho `in(double)`. Câu lệnh trên tương đương với `int i = 5.0`. Vì thế, những phương thức phổ biến thường nên nạp chòng. Như trường hợp trên, ta có thể cài đặt thêm một phương thức `in` khác nhận `double` làm tham số đầu vào.

Ví dụ 3.27: Dùng đa hình để xử lý nhiều kiểu dữ liệu tham số

```

class Test {
    public static void main(String args[]) {
        in(5);
        in(5.0);
    }
    static void in(double d) {
        System.out.println("double: " + d);
    }
}

```

```

    static void in(int i) {
        System.out.println("int: " + i);
    }
}

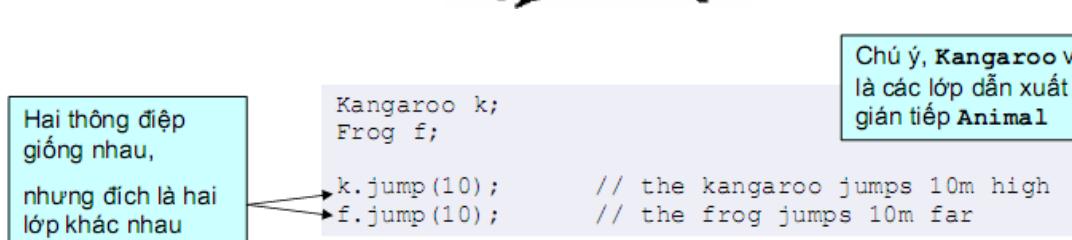
```

Tuy nhiên, đây chưa thực sự là trọng tâm của đa hình hướng đối tượng, vì đây thực sự là hai thông điệp in khác nhau. Ta cần một phương thức mà ứng với mỗi đối tượng ở các lớp khác nhau nó được hiểu theo các nghĩa khác nhau. Ta cần đến loại đa hình thứ hai: *Overriding*.

- Vượt quyền (*Overriding*)

Đa hình được cài đặt bởi một khái niệm tương tự Overloading nhưng hơi khác: *method overriding* với “*override*” có nghĩa “vượt quyền”.

+ *Method overriding*: nếu một phương thức của lớp cơ sở được định nghĩa lại tại lớp dẫn xuất thì định nghĩa tại lớp cơ sở có thể bị “che” bởi định nghĩa tại lớp dẫn xuất. Với *method overriding*, *tất cả thông điệp* (cả tên và tham số) là *hoàn toàn giống nhau* - điểm khác nhau là lớp đối tượng được nhận thông điệp. Ví dụ nhận được cùng một thông điệp *jump* “nhảy”, một con kangaroo và một con éch nhảy theo hai kiểu khác nhau: chúng cùng có phương thức “nhảy” nhưng các phương thức này có nội dung khác nhau.

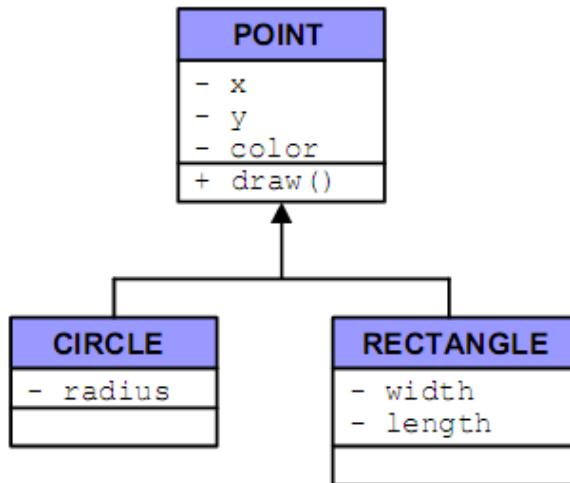


**Chú ý, Kangaroo và Frog đều
là các lớp dẫn xuất từ lớp cơ sở
gián tiếp Animal**

Hình 3.13. Hai đối tượng hiểu một thông điệp theo 2 cách riêng.

Để hiểu về đa hình vượt quyền, phân tích ví dụ sau.

Ví dụ 3.28: Xét phương thức *vẽ draw* của các lớp trong cây dưới đây:



Thông điệp *draw* gửi cho một thể hiện của mỗi lớp trên sẽ yêu cầu thể hiện đó tự vẽ chính nó.



Một thể hiện của *Point* phải vẽ một điểm, một thể hiện của *Circle* phải vẽ một đường tròn, và một thể hiện của *Rectangle* phải vẽ một hình chữ nhật.

Với đặc điểm đa hình của *method overriding*, ta sẽ có được điều trên, với phương thức *draw* được định nghĩa lại tại các lớp dẫn xuất.

```

public class Point {
    Point(int x, int y, String color){ //code
        x=x;
        y=y;
        color = color;
    }
    Point(){this(0,0,"");}
    public void draw(){ //code
        System.out.println("vẽ điểm");
    } // vẽ điểm
    private int x,y;
    String color;
}
public class Circle extends Point{
    Circle(int x, int y, String color, int bk){
        super(x,y,color);
        bk = bk;
    }
    Circle(){}
}

```

```

    public void draw(){
        System.out.println("Vẽ đường tròn");
    } // vẽ 1 đường tròn
    private int bk;
}

```

Kết quả khi tạo các thê hiện của các lớp khác nhau và gửi thông điệp *draw*, các phương thức thích hợp sẽ được gọi.

```

public static void main(String args[]) {
    Point p = new Point(0,0,"white");
    Circle c =new Circle(100,100,"blue",50);
    p.draw();
    // Vẽ 1 điểm trắng tại điểm (0,0)
    c.draw();
    // vẽ một hình tròn màu xanh, bán kính 50 tâm tại điểm
    (100,100)
}

```

Ta nói rằng: Phương thức *draw()* trong lớp *Dog* đã vượt quyền phương thức *draw()* của lớp *Animal*.

Tuy nhiên, với đa hình, chúng ta cần cẩn thận khi lập trình vì có thể có nhiều rắc rối nảy sinh liên quan tới vấn đề chuyển kiểu. Quay trở lại *upcasting* với cây Nguoi-Nhanvien-Quanly.

Ví dụ 3.29: Giả sử các lớp Nguoi, Nhanvien, Quanly cài đặt hành vi *talk* khác nhau như sau:

```

public class Nguoi {
    public void talk(){
        System.out.println("Toi la Nguoi!");
    }
}
public class Nhanvien extends Nguoi{
    public void talk(){
        System.out.println("Toi la Nhan vien!");
    }
}
public class Quanly extends Nhanvien{
    public void talk(){
        System.out.println("Toi la quan ly!");
    }
}

```

Rõ ràng với khả năng *upcasting* chúng ta có thể:

```

Quanly m = new Quanly();
Nhanvien e = m;// upcasting: an toan
Nguoi p = e; // upcasting: an toan
p.talk();

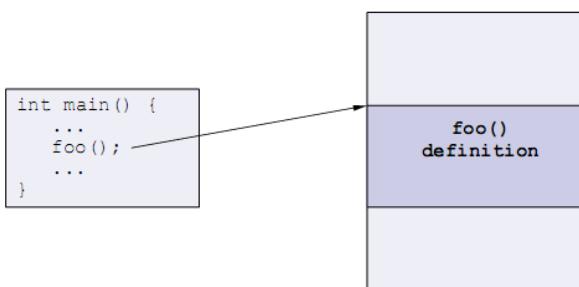
```

```
// p la Nguoi, nhung lai noi Toi la Nhan vien
e.talk();
// e la Nhan vien, nhung lai noi Toi la Quan ly
```

Phương thức `talk()` của `p` khi thực hiện lại ra kết quả khác với dòng chữ được in ra trong thân của phương thức `talk()` của `Nguoi` hay tương tự với `e.talk()` điều này thực sự có liên quan đến lời gọi phương thức.

b) Vai trò của liên kết phương thức đối với đa hình

Liên kết gọi hàm (Function call binding) là quy trình xác định khối mã hàm cần chạy khi một lời gọi hàm được thực hiện. Trong ngôn ngữ lập trình có cấu trúc (ví dụ C), việc này rất dễ dàng vì với mỗi tên hàm chỉ có một định nghĩa hàm.



Hình 3.14. Liên kết lời gọi hàm trong lập trình có cấu trúc

Hiện tượng đa hình hàm buộc trình biên dịch phải kiểm tra cả danh sách tham số cùng tên hàm khi xác định liên kết. Trong định nghĩa các lớp, vấn đề này phức tạp hơn. Nếu như lớp độc lập, không tham gia vào cây thừa kế nào thì quy trình hoạt động của *liên kết lời gọi phương thức* (*method call binding*) gần giống liên kết lời gọi hàm (*function call binding*) như đã đề cập ở trên, nghĩa là tìm phương thức phù hợp theo danh sách tham số, tên và kiểu trả về.

Hai hiện tượng liên kết như trên về cơ bản thuộc loại liên kết tĩnh (*static binding*). Sang phần tiếp theo ta sẽ tìm hiểu sự khác nhau giữa đa hình tĩnh dựa trên liên kết tĩnh và đa hình động dựa trên liên kết động.

- Đa hình tĩnh - *static polymorphism*

Static function call binding (hoặc *static binding* – liên kết tĩnh) là quy trình liên kết một lời gọi hàm với một định nghĩa hàm tại thời điểm biên dịch. Do đó

kiểu liên kết này còn được gọi là “*compile-time binding*” – liên kết khi biên dịch, hoặc “*early binding*” – liên kết sớm. *Đa hình tĩnh (dynamic polymorphism)* là loại đa hình được cài đặt bởi liên kết tĩnh.

Với liên kết tĩnh, quyết định “định nghĩa hàm nào được chạy” được đưa ra tại thời điểm biên dịch. Như vậy liên kết tĩnh có một số ưu điểm như sau:

- + Tốc độ nhanh.
- + Thích hợp cho các lời gọi hàm thông thường: mỗi lời gọi hàm chỉ xác định duy nhất một định nghĩa hàm, kể cả trường hợp hàm chồng.
- + Phù hợp với các lớp độc lập không thuộc cây thừa kế nào: Mỗi lời gọi phương thức từ một đối tượng của lớp hoặc một đối tượng được gán bởi một đối tượng khác cùng của lớp đó đều xác định duy nhất một phương thức.

Bên cạnh ưu điểm, liên kết tĩnh cũng có nhược điểm là địa chỉ đoạn mã cần chạy cho một lời gọi hàm cụ thể là không đổi trong suốt thời gian chương trình chạy.

Trong một số ngôn ngữ lập trình, như *C++*, với cách thiết kế các lớp đối tượng không thành một cây thừa kế ngay từ đầu, liên kết tĩnh là mặc định. Vì thế *đa hình tĩnh là mặc định*. Với ngôn ngữ lập trình *Java*, các lớp đối tượng tạo thành một cây thừa kế (gốc là *Object* đã học), *mặc định đa hình là đa hình động dựa trên liên kết động* mà chúng ta sẽ xem xét tiếp theo đây.

- *Đa hình động - dynamic polymorphism*

Dynamic function call binding (dynamic binding – liên kết động) là quy trình liên kết một lời gọi phương thức với một định nghĩa phương thức tại thời gian chạy. Nó còn gọi là “*run-time*” *binding* hoặc “*late binding*”. Khi chạy, chương trình sau khi đã xác định được phiên bản của phương thức nào phù hợp với đối tượng được gọi mới tiến hành chạy phương thức đó.

Đa hình động (dynamic polymorphism) là loại đa hình được cài đặt bởi liên kết động.

Lưu ý: Java cài đặt đa hình động mặc định dựa trên liên kết động. Do đó, trong trường hợp cần cài đặt đa hình tĩnh trong Java ta cần tới phương thức riêng được trình bày tiếp theo.

c) *Phương thức riêng*

Phương thức riêng là phương thức có quyền truy cập đặt là *private*.

Cú pháp:

```
private <type> <method name> (<parameters>) { ... }
```

Các cài đặt trong thân phương thức được thực hiện bình thường. Khi phương thức là *private* thì chỉ các phương thức trong cùng lớp với phương thức đó mới truy cập được. Vì thế nếu một phương thức ở lớp cha đặt là *private* thì lớp con không thể truy cập được tới phương thức này:

```
public class A {
    private void foo() { ... }
}
public class B extends A{
    ...
    A.foo(); // không hợp lệ
}
```

Trong ví dụ trên, lớp B không thể thừa kế được phương thức *private* từ lớp A, vì thế ghi đè phương thức *method overriding* sẽ không xảy ra.

3. Cài đặt đa hình

a) *Lớp trừu tượng - abstract class*

Lớp cha trong cây thừa kế khi không thực sự được dùng để tạo ra thể hiện thì nó được tạo ra nhằm mục đích cho tất cả các lớp được thừa kế từ lớp cha đó sử dụng chung mã nguồn.

Cú pháp khai báo một lớp trừu tượng sử dụng từ khóa *abstract*

```
abstract class Base{}
```

Khi khai báo một lớp là trừu tượng, việc tạo thể hiện cho lớp đó, hay cài đặt một phương thức nào đó cho lớp không có ý nghĩa, Java ngăn điều đó xảy ra ngay khi biên dịch để chặn người dùng.

Ví dụ 3.30: Đoạn lệnh viết lớp Shape mô tả các hình học nói chung dưới đây sẽ gây ra lỗi biên dịch

```
abstract class Shape {  
    protected int x, y;  
    Shape(int _x, int _y) {  
        x = _x;  
        y = _y;  
    }  
    // Trong hàm main ta viết  
    Shape s = new Shape(10, 10)// báo lỗi
```

Chúng ta chỉ có thể thừa kế lại `Shape(int _x, int _y)` ở lớp con của nó:

```
class Circle extends Shape {  
    int r;  
    public Circle(int _x, int _y, int _r) {  
        super(_x, _y);  
        r = _r;  
    }  
    // code  
}  
// Trong hàm main ta viết  
Circle c = new Circle(10,10,5); //hợp lệ
```

Các phương thức được cài đặt ở lớp trừu tượng chỉ phát huy tác dụng khi các thể hiện của lớp con gọi tới các phương thức này. Khi đó tính chất *mọi thể hiện của lớp con cũng được coi như thể hiện của lớp cha* càng được biểu hiện rõ ràng hơn.

b) Phương thức trừu tượng - abstract method

Phương thức được khai báo nguyên mẫu kiểu *prototype* tại lớp cơ sở nhưng được cài đặt thực tế tại lớp dẫn xuất gọi là *phương thức trừu tượng - abstract method*. Các lớp dẫn xuất khác nhau có cách cài đặt khác nhau. Một phương thức ở lớp cha được khai báo là phương thức trừu tượng thì bắt buộc phải được định nghĩa tại lớp dẫn xuất.

Ví dụ 3.31: Lớp *Shape* (Hình khối) và lớp *Circle* (Hình tròn) được cài đặt như sau:

```
abstract public class Shape{
    protected int x, y;
    public Shape() {}
    public Shape(int _x, int _y) {
        x = _x;
        y = _y;
    }
    public void moveTo(int x1, int y1) {
        erase();
        x = x1;
        y = y1;
        draw();
    }
    abstract public void erase();
    abstract public void draw();
}
public class Circle extends Shape {
    int r;
    public Circle(int _x, int _y, int _r) {
        super(_x, _y);
        r = _r;
        draw();
    }
    public void erase() {
        System.out.println("Than lop erase");
    }
    public void draw() {
        System.out.println("Than lop draw");
    }
    public static void main(String [] args){
        Circle c = new Circle (10,10, 100);
        c.erase();
        c.draw();
    }
}
```

```

class Circle extends Shape {
    int r;
    public Circle(int _x, int _y, int _r) {
        super(_x, _y);
        r = _r;
        draw();
    }
    public void erase() {
        System.out.println("Xoa o diem (" + x + "," + y + ")");
    }
    public void draw() {
        System.out.println("Tao o diem (" + x + "," + y + ")" +
            " duong tron ban kinh" + r);
    }
    public static void main(String args[]) {
        Circle c = new Circle(10,10,5);
    }
}

```

put - Vídu (run) Search Results

run:
Tao o diem (10,10) duong tron ban kinh5
BUILD SUCCESSFUL (total time: 0 seconds)

Hình 3.15. Kết quả chạy Circle.

Ta có thể cài thêm một lớp *Rectangle* (hình chữ nhật) cũng thừa kế từ *Shape* và sử dụng lại phương thức khởi tạo của *Shape*; cài đặt lại phương thức vẽ và xóa theo cách của riêng nó:

```

class Rectangle extends Shape {
    int width, height;
    public Rectangle(int _x, int _y, int _width, int _height) {
        super(_x, _y);
        width = _width;
        height = _height;
        draw();
    }
    public void erase() {
        System.out.println("Xoa o diem (" + x + "," + y + ")");
    }
    public void draw() {
        System.out.println("Tao o diem (" + x + "," + y + ")" +
            "hinh chu nhat chieu dai "+width+"chieu rong" +height);
    }
    // Trong main viết thêm
    public static void main(String args[]) {
        Circle c = new Circle(10,10,80);
        Rectangle r = new Rectangle(5,10,50,60);
        /* c.draw();
        r.draw();
        c.moveTo(15,20); */
    }
}

```

```

        r.moveTo(50, 90);
        c.erase();
        r.erase();*/
    }
}

public static void main(String args[]){
    Circle c = new Circle(10,10,5);
    Rectangle r = new Rectangle (20,10, 30,5);
}

```

8:39 INS

Output - Vdu (run) Search Results

```

run:
Tao o diem (10,10) duong tron ban kinh5
Tao o diem (20,10) hinh chu nhat chieu dai 30chieu rong5
BUILD SUCCESSFUL (total time: 0 seconds)

```

Hình 3.16. Cài thêm *Rectangle*.

III. MỎ RỘNG ĐA THÙA KẾ BẰNG GIAO DIỆN

1. Khái niệm và cú pháp khai báo giao diện

a) Khái niệm

Java không cho phép đa thừa kế ở điểm: một lớp con có nhiều lớp cha. Thay vào đó, Java phát triển khái niệm khác là *giao diện* - *interface*. *Interface* là mức trừu tượng cao hơn lớp trừu tượng. Có thể coi nó là lớp trừu tượng thuần túy. Nó cho phép trình biên dịch tạo ra một dạng giống như lớp: tên phương thức, danh sách tham số và kiểu trả về nhưng lại không có bất kì cài đặt phương thức nào. Một *interface* có thể chứa dữ liệu nhưng nó phải là *static* hoặc *final*. *Interface* không cho phép tạo thể hiện. Một *interface* bao gồm:

- Phương thức trừu tượng
- Hằng số (*static* hoặc *final*)
- Mặc định quyền truy cập là *public*

b) Cú pháp khai báo

- Cú pháp sử dụng cặp từ khóa **interface** và **implements**
- + Khai báo interface:

```
interface A{  
    ...  
}
```

+ Cài đặt interface:

```
class B implements A {  
    ...  
}
```

Lưu ý : lớp B phải định nghĩa các phương thức khai báo ở lớp A.

Ví dụ 3.32: Cài đặt giao diện *Action* cho *Circle*

```
interface Action {  
    void moveTo(int x, int y);  
    void erase();  
    void draw();  
}  
class Circle1 implements Action {  
    int x, y, r;  
    Circle1(int _x, int _y, int _r) { ... }  
    public void erase() {...}  
    public void draw() {...}  
    public void moveTo(int x1, int y1) {...}  
}
```

Trong một file .java, có thể định nghĩa nhiều giao diện hoặc có thể vừa định nghĩa giao diện vừa định nghĩa các lớp sử dụng nó.

2. Phát triển giao diện

a) Lớp trừu tượng cài đặt giao diện

Lớp trừu tượng cũng có thể cài đặt được giao diện. Khi đó, các phương thức của giao diện sẽ không bị yêu cầu phải cài đặt trong lớp trừu tượng. Có nghĩa là đoạn chương trình sau hoàn toàn hợp lệ.

Ví dụ 3.33: Lớp ảo *Shape* cài đặt giao diện *Action*

```
interface Action {  
    void moveTo(int x, int y);  
    void erase();  
    void draw();  
}  
abstract public class Shape implements Action{  
    protected int x, y;  
    public Shape() {}
```

```

public Shape(int _x, int _y) {
    x = _x;
    y = _y;
}
public void moveTo(int x1, int y1) {
    erase();
    x = x1;
    y = y1;
    draw();
}
}

```

Tuy nhiên lớp *Circle* thừa kế từ *Shape* vẫn bị đòi hỏi phải cài đặt lại *draw()*

và *erase()*:

```

public class Circle extends Shape {
    int r;
    public Circle(int _x, int _y, int _r) {
        super(_x, _y);
        r = _r;
        draw();
    }
    public void erase() {System.out.println("Than lop erase");}
    public void draw() {System.out.println("Than lop draw");}
    public static void main(String [] args){
        Circle c = new Circle (10,10, 100);
        c.erase();
        c.draw();
    }
}

```

Kết quả in ra vẫn không thay đổi, việc *Shape* cài đặt giao diện *Action* cũng giống như *Shape* khai báo các phương thức *draw()* , *erase()* , *moveTo()* trong thân của nó vậy. Lúc này, ta có thể khai báo lớp *Rectangle* không thừa kế từ lớp *Shape* mà vẫn có các phương thức *draw()* , *erase()* , *moveTo()* bằng cách cài đặt trực tiếp giao diện *Action*.

```

class Rectangle implements Action{
    int width, height, x,y;
    public Rectangle(int _x, int _y, int _width, int _height) {
        x= _x;
        y=_y;
        width = _width;
        height = _height;
        draw();
    }
    public void erase() {

```

```

        System.out.println("Xoa o diem (" + x + "," + y +
")");
    }

    public void draw() {
        System.out.println("Tao o diem (" + x + "," + y +
") " +
            " hinh chu nhat chieu dai "+width+"chieu rong" +height);
    }
    public void moveTo(int x, int y) {
        erase();
        x = x;
        y = y;
        draw();
    }
}

```

Chính tính linh động đó của *Interface* đã tạo cho nó khả năng tham gia vào đa thừa kế: *thay thế cho một lớp con thừa kế từ nhiều lớp cha*.

b) Cài đặt nhiều giao diện thay vì đa thừa kế

Java không cho phép đa thừa kế từ nhiều lớp cơ sở là để đảm bảo tính dễ hiểu và hạn chế xung đột. Tuy nhiên Java cho phép có thể cài đặt đồng thời nhiều giao diện. Hơn nữa, giao diện không có các cài đặt cho các phương thức, nên nó rất tiết kiệm bộ nhớ. Để cài đặt nhiều giao diện ta chỉ cần liệt kê tên các giao diện cách nhau bởi dấu phẩy. Cú pháp như sau:

- Khai báo interface:

```

//A.java
interface A{
    ...
}

// B.java
interface B{
    ...
}

// C.java
interface C{
    ...
}
.....

```

- Cài đặt interface:

```
class C implements A,B,C {  
    ...  
}
```

Lưu ý: Cài đặt bao nhiêu giao diện thì phải cài đặt bấy nhiêu phương thức của giao diện đó.

Một lớp vừa có thể cài đặt nhiều giao diện, vừa có thể thừa kế được từ một lớp khác. Như ví dụ dưới đây: lớp *Hero* mô tả các siêu nhân có thể đánh nhau, bơi lội hoặc là bay.

Ví dụ 3.34:

```
interface CanFight {  
    void fight();  
}  
interface CanSwim {  
    void swim();  
}  
interface CanFly {  
    void fly();  
}  
class ActionCharacter {  
    public void fight() {}  
}  
class Hero extends ActionCharacter  
    implements CanFight, CanSwim, CanFly {  
    public void swim() {}  
    public void fly() {}  
}  
public class Adventure {  
    static void t(CanFight x) { x.fight(); }  
    static void u(CanSwim x) { x.swim(); }  
    static void v(CanFly x) { x.fly(); }  
    static void w(ActionCharacter x) { x.fight(); }  
    public static void main(String args[]) {  
        Hero spiderman = new Hero();  
        t(spiderman); // Coi nhu la mot CanFight  
        u(spiderman); // Coi nhu la mot CanSwim  
        v(spiderman); // Coi nhu la mot CanFly  
        w(spiderman); // Coi nhu la mot ActionCharacter  
    }  
}
```

Ta thấy rằng giữa *interface* và lớp cài đặt nó cũng có khả năng *upcasting*. Như ví dụ trên khai báo biến *spiderman* là một thể hiện của lớp *Hero* nhưng vẫn dùng nó làm tham số đầu vào cho các phương thức *t*, *u*, *v*, *w*.

Lưu ý: Các phương thức có thể chồng chéo lên nhau gây ra hiện tượng nạp chồng (*overloading*) như ví dụ dưới đây.

Ví dụ 3.35: Trong cùng 1 file, viết đoạn chương trình sau:

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
    //public int f() { return 1; }
    public int f(int i) { return 1; }
}
class C2 implements I1, I2 {
    public void f() {} //C2 cai dat void f() cua I1
    public int f(int i) { return 1; }
}
class C3 extends C implements I2 {
    public int f(int i) { return 1; }
}
```

Trong đoạn chương trình trên, ở lớp C2, phương thức *int f(int i)* của I2, phương thức *void f()* của I1 được cài đặt và hai phương thức này đã nạp chồng lẫn nhau. Trong lớp C3, cài đặt phương thức *int f(int i)* của I2, nhưng nó lại vượt quyền (*overriding*) phương thức *int f(int i)* của C. Nếu bỏ khai báo phương thức *int f(int i)* trong C, thay vào đó cài đặt phương thức *int f()* như sau:

```
class C {
    public int f() { return 1; }
}
class C4 extends C implements I3 {
    //dinh nghia lai f cua I3
    public int f() { return 1; }
}
```

Khi đó, ở lớp C3 hiện tượng nạp chồng đã xảy ra, ở lớp C4: *public int f()* vượt quyền phương thức *int f()* của C. Như vậy đã có xung đột xảy ra, mặc dù không gây ra lỗi cho chương trình, nhưng nó làm chương trình mất tính rõ ràng.

Xét tiếp câu lệnh:

```
class C5 extends C implements I1 {} // báo lỗi  
interface I4 extends I1, I3 {} // báo lỗi
```

Ở đây C5 thừa kế từ C, mà C khai báo và cài đặt phương thức `int f()`, nhưng phương thức này không thể nạp chồng được với phương thức `void f()` trong I1. Đối với C5 `void f()` chưa được cài đặt, dù ta đã cài đặt một phương thức `void f()` trong C2. Điều đó cũng đúng vì lúc đó `void f()` được xem như là phương thức của C2. Tiếp tục thay dòng lệnh khai báo C5 thành *implements I3*

```
class C5 extends C implements I3 {}
```

Câu lệnh trên không báo lỗi, dù C5 chưa cài đặt phương thức `int f()` nhưng có lớp C đã cài đặt `int f()` rồi nên được chấp nhận.

Tóm lại, mọi việc sẽ trở nên rắc rối khi chúng ta không cẩn thận để cho các lớp nạp chồng, chồng chéo lên nhau hoặc đôi khi không biết phương thức nào cài đặt ở lớp nào. Chính vì thế đa thừa kế là phức tạp và cần cẩn thận khi dùng.

Tiếp tục với dòng lệnh: `interface I4 extends I1, I3 {}`. Khi viết dòng lệnh này, trình biên dịch sẽ báo lỗi: *I3 and I1 are incompatible, both define f(), but with unrelated return types*. Có nghĩa là I1 và I3 không tương thích, cả hai cùng định nghĩa `f()` nhưng với các kiểu trả về không liên quan đến nhau. Nếu bỏ đi 1 trong 2 Interface *I1* hoặc *I3*, câu lệnh lại hợp lệ; nếu thay *I1* bằng *I2*, câu lệnh cũng hợp lệ; nếu đổi tên phương thức *f* trong *I1* thành tên khác, ví dụ *fx*, câu lệnh cũng hợp lệ. Như vậy Java cho phép một giao diện thừa kế từ nhiều giao diện khác, nhưng không cho phép các giao diện khác đó có các phương thức trùng tên nhưng khác kiểu trả về.

c) Mở rộng lớp trừu tượng và giao diện

Java cung cấp các khả năng sau:

- Một giao diện có thể thừa kế từ một hay nhiều giao diện khác.

- Một lớp trừu tượng có thể thừa kế một lớp trừu tượng khác, đồng thời cũng có thể cài đặt nhiều giao diện khác.

Như vậy các câu lệnh sau hoàn toàn hợp lệ:

```
interface I1 {}  
interface I2 {}  
interface I3 extends I1, I2 {}  
abstract class A1 {}  
abstract class A2 extends A1 implements I1, I2 {}
```

Đây là cách để mở rộng lớp trừu tượng hoặc mở rộng giao diện.

Như vậy lớp trừu tượng có thể có phương thức và thuộc tính. Còn giao diện thì có thể đa thừa kế. Chính vì thế ta nói giao diện là một lớp trừu tượng thuận tủy nhưng ở mức cao hơn.

d) Các ứng dụng khác của giao diện

Thứ nhất, giao diện có thể được dùng để tạo ra một nhóm các hằng. Lúc này *interface* giống như kiểu liệt kê mà ta đã gặp trong Pascal hay C:

```
public interface Months {  
    int JANUARY = 1, FEBRUARY = 2, MARCH = 3,  
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,  
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,  
    NOVEMBER = 11, DECEMBER = 12;  
}
```

Một cách tự động, các hằng số trên đều có quyền *public* nên các lớp sau đó cài đặt *interface* này đều có thể truy nhập vào các hằng số trên. Các số trên là các hằng dữ liệu tĩnh được khởi tạo trong thời gian biên dịch.

Thứ hai, có thể dùng *interface* để khởi tạo các trường dữ liệu

```
public interface RandVals {  
    int rint = (int)(Math.random() * 10);  
    long rlong = (long)(Math.random() * 10);  
    float rfloat = (float)(Math.random() * 10);  
    double rdouble = Math.random() * 10;  
}
```

Các trường dữ liệu trên sẽ có kiểu *final static*, giống như các ví dụ chúng ta đã phân tích trong phần hằng dữ liệu tĩnh với giá trị của các dữ liệu được khởi tạo trong thời gian chạy. Như vậy chúng ta có thể sử dụng interface trên như sau:

```
public class TestRandVals {  
    public static void main(String args[]) {  
        System.out.println(RandVals.rint);  
        System.out.println(RandVals.rlong);  
        System.out.println(RandVals.rfloat);  
        System.out.println(RandVals.rdouble);  
    }  
}
```

Câu hỏi và bài tập cuối chương

Câu 1.

a. Xây dựng lớp trừu tượng Shape có các phương thức:

- double area(); trả về giá trị diện tích của hình.
- double volume(); trả về giá trị thể tích của hình.
- abstract String getName(); trả về tên của đối tượng hình học.
- abstract void toString(); in ra màn hình giá trị các thuộc tính của đối tượng dưới dạng chuỗi.

b. Xây dựng lớp Point kế thừa lớp Shape với 2 thuộc tính x, y và các phương thức khởi tạo:

- Point();
- Point(int t) cho biết hoành độ x = t;
- Point(int t, int m);
- Các phương thức get, set giá trị các thuộc tính.
- Phương thức toString, getName;

c. Xây dựng lớp Square kế thừa từ lớp Point, có các thuộc tính x, y và a (chiều dài cạnh). Xây dựng:

- 3 phương thức khởi tạo cho lớp Square.
- Các phương thức get, set giá trị thuộc tính.
- Phương thức tính chu vi hình vuông.

d. Xây dựng lớp Circle kế thừa lớp Point, có các thuộc tính: x, y, radius. Xây dựng các phương thức :

- Phương thức khởi tạo: Circle(); Circle(R, x, y);
- Các phương thức get, set giá trị thuộc tính.
- Phương thức area(), perimeter() tính diện tích, chu vi hình tròn.

e. Xây dựng lớp Cylinder kế thừa từ lớp Circle, có các thuộc tính x, y, r, h (chiều cao). Xây dựng các phương thức:

- Phương thức khởi tạo.
- Các phương thức get, set giá trị thuộc tính.
- Phương thức area(), volume() để tính diện tích và thể tích hình trụ.

f. Xây dựng lớp Test để kiểm tra tính kế thừa của các đối tượng hình học, kiểm tra các phương thức của các lớp đã tạo.

Câu 2.

a. Xây dựng lớp trừu tượng như sau:

```
abstract class Hinh{  
    abstract public float getArea();  
}
```

b. Xây dựng lớp HinhVuong, HinhTron, HinhTamGiac bằng cách kế thừa từ lớp Hinh và ghi đè phương thức getArea() để tính diện tích của các hình tương ứng.

Câu 3.

a. Xây dựng lớp DaySo để mô tả một dãy số, gồm các phương thức sau:

- Phương thức nhap để nhập dãy số từ bàn phím.
- Phương thức print để in dãy số ra màn hình.
- Hàm tạo DaySo(int n) dùng để khởi tạo một mảng gồm n phần tử.

b. Xây dựng giao diện Sort như sau:

```
interface Sort{  
    public void Sort();  
}
```

c. Xây dựng các lớp BubbleSort, SelectionSort, InsertionSort bằng cách kế thừa từ lớp DaySo và thực thi giao diện Sort để thực hiện việc sắp xếp.

Câu 4. Xây dựng lớp trừu tượng A, có phương thức trừu tượng inRa() và phương thức in() in các số từ 0 đến 9 ra màn hình.

- Xây dựng 2 lớp B, C kế thừa từ lớp A, ghi đè phương thức inRa();
- Với lớp B, phương thức inRa() in ra màn hình dòng chữ “Day la lop B”.
- Với lớp C, phương thức inRa() in ra màn hình dòng chữ “Day la lop C”.
- Phương thức main() của lớp B và lớp C gọi sử dụng 2 phương thức in() và inRa();

Câu 5.

a) Xây dựng lớp “DaySo” để mô tả một dãy số, gồm các phương thức sau:

- Phương thức “nhap” dùng để nhập dãy số từ bàn phím.
- Phương thức “print” dùng để in dãy số ra màn hình.
- Hàm tạo DaySo(int n) dùng để khởi tạo một mảng gồm n phần tử.

b) Xây dựng giao diện Sort như sau:

```
interface Sort{
    public void Sort();
}
```

c) Xây dựng các lớp “QuickSort”, “BubbleSort”, “SelectionSort”, “InsertSort” bằng cách kế thừa từ lớp DaySo và triển khai giao diện Sort để thực hiện việc sắp xếp: nổi bọt, chọn trực tiếp, chèn trực tiếp.

Câu 6.

a) Xây dựng lớp trừu tượng “Sort” như sau:

```
abstract class Sort extends DaySo{
    abstract public void sapXep();
}
```

trong đó lớp DaySo được xây dựng trong Câu 5.

b) Xây dựng các lớp “QuickSort”, “BubbleSort”, “SelectionSort”, “InsertionSort” bằng cách kế thừa từ lớp Sort để thực hiện việc sắp xếp: nổi bọt, chọn trực tiếp, chèn trực tiếp.

Chương 4. CÁC LUỒNG VÀO RA VÀ XỬ LÝ NGOẠI LỆ

I. CÁC LUỒNG VÀO/RA

Việc lưu trữ dữ liệu trong các biến chương trình, các mảng có tính chất tạm thời và dữ liệu sẽ mất đi khi biến ra khỏi tầm ảnh hưởng của nó hoặc khi chương trình kết thúc. Sử dụng tệp (*file*) giúp cho các chương trình có thể lưu trữ một lượng lớn dữ liệu, cũng như có thể lưu trữ dữ liệu trong một thời gian dài ngay cả khi chương trình kết thúc. Phần này sẽ trình bày cách các chương trình Java có thể tạo, đọc, ghi và xử lý các tệp tuần tự và các tệp truy cập ngẫu nhiên.

Xử lý tệp là một phần của công việc xử lý các luồng, giúp cho một chương trình có thể đọc, ghi dữ liệu trong bộ nhớ, trên tệp và trao đổi dữ liệu thông qua các kết nối trên mạng. Đây là một vấn đề hết sức cơ bản, quan trọng mà bất kỳ một ngôn ngữ lập trình nào cũng phải hỗ trợ những thư viện, hàm để xử lý một số thao tác cơ bản nhất đối với kiểu dữ liệu tệp.

1. Luồng và phân loại luồng

a) Khái niệm luồng

Tất cả những hoạt động nhập/xuất dữ liệu (nhập dữ liệu từ bàn phím, lấy dữ liệu từ mảng về, ghi dữ liệu ra đĩa, xuất dữ liệu ra màn hình, máy in..) đều được quy về một khái niệm gọi là luồng (*stream*). Luồng thường được hệ thống xuất nhập trong Java gắn kết với một thiết bị vật lý. Tất cả các luồng đều có chung một nguyên tắc hoạt động ngay cả khi chúng được gắn kết với các thiết bị vật lý khác nhau. Vì vậy cùng một lớp, phương thức xuất nhập có thể dùng chung cho các thiết bị vật lý khác nhau. Chẳng hạn cùng một phương thức có thể dùng để ghi dữ liệu ra *console*, đồng thời cũng có thể dùng để ghi dữ liệu xuống một tệp trên đĩa. Java hiện thực luồng bằng tập hợp các lớp phân cấp trong gói `java.io`. Về bản chất, luồng là một “dòng chảy” của dữ liệu đến từ một nguồn hoặc đi đến một đích. Nguồn và đích có thể là tệp, bộ nhớ, một tiến trình, hay thiết bị (bàn phím, màn hình..)

b) Phân loại luồng

Theo hướng luồng, có thể phân thành 2 loại:

- Luồng nhập: Gắn với các thiết bị nhập như bàn phím, máy scan, tệp...
- Luồng xuất: Gắn với các thiết bị xuất như màn hình, máy in, tệp...

Việc xử lý vào ra thông qua luồng giúp cho lập trình viên không phải quan tâm đến bản chất của thiết bị vào ra. Chương trình sử dụng *input stream* để đọc dữ liệu từ nguồn. Chương trình sử dụng *output stream* để ghi dữ liệu xuống đích. Luồng có thể được kết nối tới một tệp trên đĩa mềm, một tệp trên đĩa cứng, một kết nối mạng hoặc có thể tới bộ nhớ. Chương trình Java gửi và nhận dữ liệu thông qua các đối tượng thuộc một kiểu luồng dữ liệu nào đó.

Theo biểu diễn luồng, Java định nghĩa hai kiểu luồng: *byte* và *ký tự*.

- Luồng *byte* (hay luồng thao tác theo đơn vị *byte*) hỗ trợ việc xuất nhập dữ liệu trên *byte*, thường được dùng khi đọc ghi dữ liệu nhị phân.

- Luồng ký tự (hay luồng thao tác với ký tự) được thiết kế hỗ trợ việc xuất nhập dữ liệu kiểu ký tự (*Unicode*). Trong một vài trường hợp luồng ký tự sử dụng hiệu quả hơn luồng *byte*, nhưng ở mức hệ thống thì tất cả những xuất nhập đều phải qui về *byte*. Luồng ký tự hỗ trợ hiệu quả chỉ đối với việc quản lý, xử lý các ký tự.

Các luồng *byte* được định nghĩa dùng hai lớp phân cấp. Mức trên cùng là hai lớp trừu tượng *InputStream* và *OutputStream*. *InputStream* định nghĩa những đặc điểm chung cho những luồng nhập *byte*. *OutputStream* mô tả cách xử lý của các luồng xuất *byte*.

InputStream

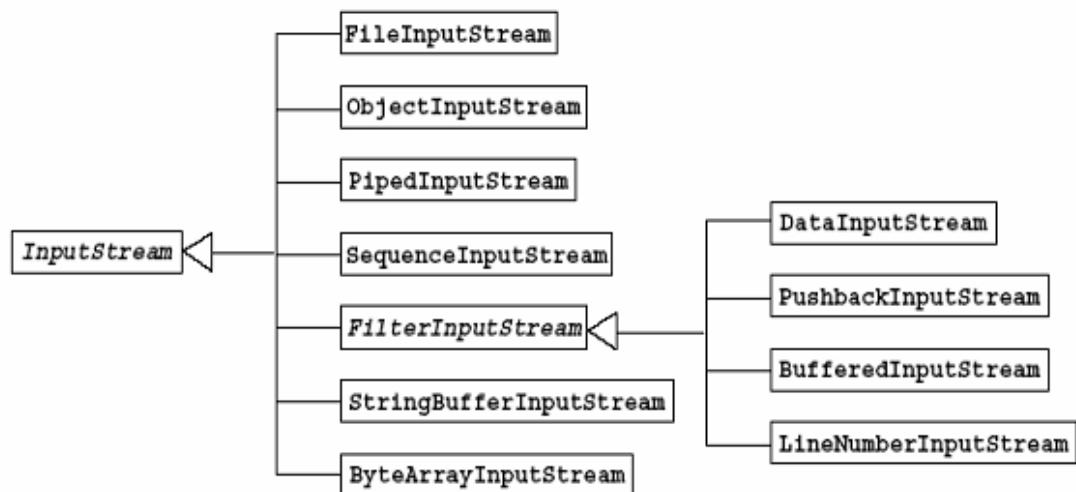
```
int read()
int read(byte buf[])
int read(byte buf[], int
offset, int length)
void close()
```

```

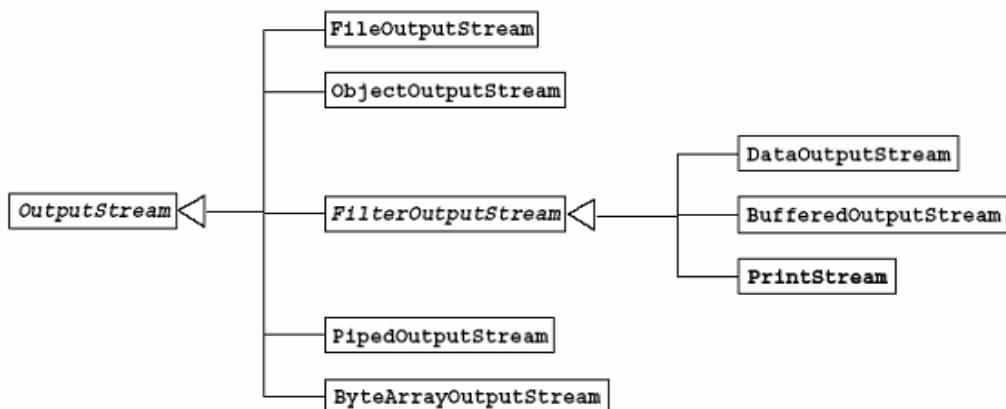
OutputStream
int write(int c)
int write(byte buf[])
int write(byte buf[], int
offset, int length)
void close()
void flush()

```

Các lớp con dẫn xuất từ hai lớp *InputStream* và *OutputStream* sẽ hỗ trợ chi tiết tương ứng với việc đọc ghi dữ liệu trên những thiết bị khác nhau.



Hình 4.1. Cây phân thừa kế các lớp điều khiển dòng nhập.



Hình 4.2. Cây thừa kế các lớp điều khiển dòng xuất.

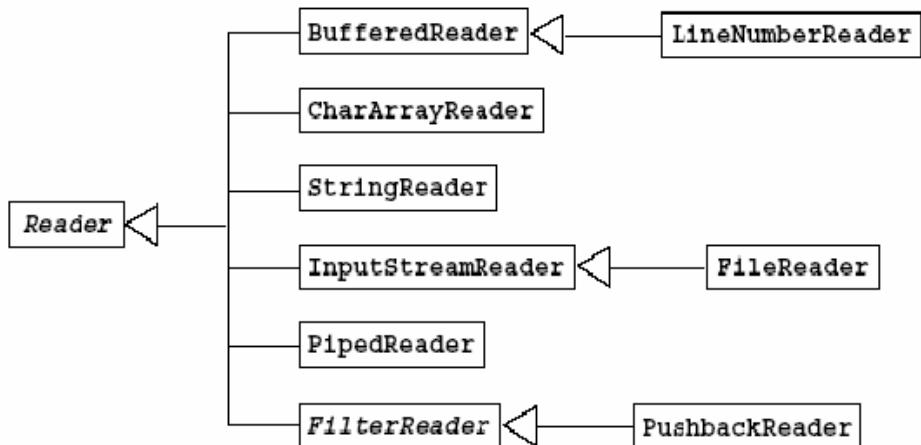
Các luồng ký tự được định nghĩa dùng hai lớp phân cấp. Mức trên cùng là hai lớp trừu tượng *Reader* và *Writer*. Lớp *Reader* dùng cho việc nhập dữ liệu của

luồng, lớp *Writer* dùng cho việc xuất dữ liệu của luồng. Những lớp dẫn xuất từ *Reader* và *Writer* thao tác trên các luồng ký tự *Unicode.Reader*

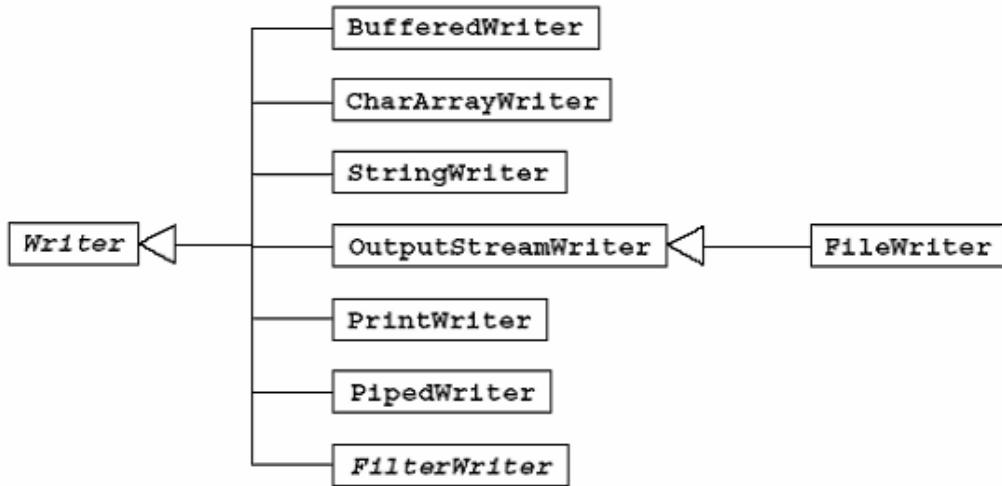
```
Reader
int read()
int read(char buf[])
int read(char buf[], int offset, int length)
void close()
```

```
Writer
int write(int c)
int write(char buf[])
int write(char buf[], int offset, int length)
void close()
void flush()
```

Các lớp con dẫn xuất từ hai lớp *Reader* và *Writer* sẽ hỗ trợ chi tiết tương ứng với việc đọc ghi dữ liệu trên những thiết bị khác nhau.



Hình 4.3. Cây thừa kế các lớp đọc dữ liệu luồng kí tự.



Hình 4.4. Cây thừa kế các lớp xuất dữ liệu luồng kí tự.

2. Thao tác với luồng

a) Nguyên tắc chung

Đối tượng nhập xuất: để nhập hoặc xuất dữ liệu, chúng ta phải tạo ra đối tượng vào hoặc ra. Đối tượng vào hoặc ra thuộc kiểu luồng tương ứng và phải được gắn với một nguồn dữ liệu hoặc một đích tiêu thụ dữ liệu.

Sử dụng bộ đệm: bộ đệm là một kỹ thuật để tăng tính hiệu quả của thao tác vào ra: đọc và ghi dữ liệu theo khói, giảm số lần thao tác với thiết bị. Thay vì ghi trực tiếp tới thiết bị thì chương trình ghi lên bộ đệm. Khi bộ đệm đầy thì dữ liệu được ghi ra thiết bị theo khói. Có thể ghi vào thời điểm bất kỳ bằng phương thức `flush()`. Thay vì đọc trực tiếp từ thiết bị thì chương trình đọc từ bộ đệm. Khi bộ đệm rỗng thì dữ liệu được đọc theo khói từ thiết bị

b) Thao tác với một số luồng trong Java API

- Thao tác với luồng chuẩn

Tất cả các chương trình viết bằng Java luôn tự động *import* gói `java.lang`. Gói này có định nghĩa lớp `System`, bao gồm một số đặc điểm của môi trường *runtime*, nó có ba biến luồng được định nghĩa trước là `in`, `out` và `err`, các biến này là các trường *fields* được khai báo *static* trong lớp `System`. Các biến luồng được mô tả dưới đây:

- + `System.out`: luồng xuất chuẩn, mặc định là *console*. `System.out` là một đối tượng kiểu *PrintStream*.
- + `System.in`: luồng nhập chuẩn, mặc định là bàn phím. `System.in` là một đối tượng kiểu *InputStream*.
- + `System.err`: luồng lỗi chuẩn, mặc định cũng là *console*. `System.out` cũng là một đối tượng kiểu *PrintStream* giống `System.out`.
- + Đọc dữ liệu với luồng nhập chuẩn (`System.in`)

Ví dụ 4.1: Sử dụng `System.in` đọc mảng byte

```
import java.io.*;

class ReadBytes{
    public static void main(String args[]) throws IOException{
        byte data[] = new byte[100];
        System.out.print("Enter some characters.");
        System.in.read(data);
        System.out.print("You entered: ");
        for(int i=0; i < data.length; i++)
            System.out.print((char) data[i]);
    }
}
```

Lưu ý:

`System.in` không sử dụng được trực tiếp để đọc ký tự. Để sử dụng `System.in` đọc ký tự/xâu, ta cần thực hiện như sau: 1- tạo đối tượng luồng ký tự (*InputStreamReader*), 2- tạo đối tượng luồng có bộ đệm (*BufferedReader*)

Ví dụ 4.2: Sử dụng `System.in` đọc xâu ký tự

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(reader);
String s;
try {
    s = in.readLine();
}
catch (IOException e) {...}
```

- + Xuất dữ liệu với luồng xuất chuẩn (`System.out`)

Ví dụ 4.3: Sử dụng `System.out` xuất ký tự

```

import java.io.*;
class WriteDemo{
    public static void main(String args[]){
        int b; b = 'X';
        System.out.write(b);
        System.out.write('\n');
    }
}

```

- Thao tác với tệp

+ Lớp File

Một trong các nguồn và đích dữ liệu thông thường là tệp. Lớp *File* cung cấp các chức năng cơ bản để thao tác với tệp. Lớp *File* nằm trong gói `java.io`, dùng để tạo tệp, mở tệp, cung cấp các thông tin về tệp và thư mục.

Ví dụ 4.4:

Tạo đối tượng File

```

File myFile;
myFile = new File("data.txt");
myFile = new File("myDocs", "data.txt");

```

Thư mục (directory) cũng được coi như là một file, có phương thức `isDirectory()` để kiểm tra đối tượng là file hay thư mục.

Ví dụ 4.5:

```

File myDir = new File("myDocs");
File myFile = new File(myDir, "data.txt");

```

Các phương thức cơ bản của lớp File:

```

String getName()
String getPath()
String getAbsolutePath()
String getParent()
boolean renameTo(File newName)
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute()
long lastModified()
long length()

```

```
boolean delete()
boolean mkdir()
String[] list()
```

Như đã trình bày ở phần phân loại luồng, theo biểu diễn luồng, Java định nghĩa hai kiểu luồng (*byte* và *ký tự*). Thao tác với hai kiểu luồng này khác nhau.

- Thao tác tuân tự với luồng ký tự: tuân tự đọc/ghi từ/lên File tuân tự từ trên xuống dưới, từ trái sang phải, từng kí tự trong tệp.

+ Đọc dữ liệu: Cần 2 biến

* FileReader: đọc ký tự từ tệp

* BufferedReader: đọc từng kí tự trong tệp đưa vào bộ đệm

Ví dụ 4.6:

```
File file = new File("data.txt");
FileReader reader = new FileReader(file);
BufferedReader in = new BufferedReader(reader);
String s;
try {
    s = in.readLine();
}
catch (IOException e) {
}

class Abc {
public void read(BufferedReader in) {
String s;
try {
    s = in.readLine();
    //...
}
catch (IOException e) {
    //...
}
}

public void doSomething() {...}
//...
}

File file = new File("data.txt");
FileReader reader = new FileReader(file);
BufferedReader in = new BufferedReader(reader);
Abc abc = new Abc();
abc.read(in);
abc.doSomething();
```

+ Ghi dữ liệu: Cần 2 biến

* FileWriter: ghi ký tự ra tệp

* PrintWriter: ghi theo định dạng

Ví dụ 4.7:

```
File file = new File("data.out");
FileWriter writer = new FileWriter(file);
PrintWriter out = new PrintWriter(writer);
String cb1 = "Nguyễn Văn A ngày sinh 15/10/1990 giới tính nam
cấp hàm trung úy";
try {
    out.println(s);
    out.close();
}
catch (IOException e) {
}

class Abc {
    ...
    public void write(PrintStream out) {
        ...
        try {
            out.println(s);
            out.close();
        }
        catch (IOException e) {...}
    }
    class Abc {
        ...
        public String write() {
            String buf;
            buf += ...
            return buf;
        }
    }
}
```

- Thao tác tuần tự với luồng byte: đọc/ghi từ/lên file theo từng byte dữ liệu một.

+ Đọc dữ liệu

* FileInputStream: đọc dữ liệu từ tệp.

* DataInputStream: đọc dữ liệu kiểu nguyên thủy readBoolean, readByte, readChar, readShort, readInt, readLong, readFloat, readDouble.

* ObjectInputStream: đọc đối tượng.

Ví dụ 4.8:

```
import java.io.*;
public class TestDataInputStream {
    public static void main(String args[]) {
        try {
            FileInputStream fin = new FileInputStream(args[0]);
            DataInputStream din = new DataInputStream(fin);
            while (true) {
                System.out.println(din.readInt());
            }
        } catch (EOFException e) {...}
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

+ Ghi dữ liệu

* FileOutputStream: ghi dữ liệu ra tệp.

* DataOutputStream: ghi các dữ liệu nguyên thủy writeBoolean, writeByte, writeChar, writeShort, writeInt, writeLong, writeFloat, writeDouble.

* ObjectOutputStream: ghi đối tượng.

Ví dụ 4.9:

```
import java.io.*;
public class TestDataOutputStream {
    public static void main(String args[]) {
        int a[] = {2, 3, 5, 7, 11};
        try {
            FileOutputStream         fout         = new
FileOutputStream(args[0]);
            DataOutputStream dout = new DataOutputStream(fout);
            for (int i=0; i<a.length; i++)
                dout.writeInt(a[i]);
            dout.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

+ Thao tác truy cập ngẫu nhiên trên tệp

Bên cạnh việc xử lý xuất nhập trên tệp theo kiểu tuần tự, Java cung cấp lớp *RandomAccessFile* không dẫn xuất từ *InputStream* hay *OutputStream* mà hiện thực các *interface DataInput*, *DataOutput* (có định nghĩa các phương thức *I/O* cơ bản). *RandomAccessFile* hỗ trợ vấn đề định vị con trỏ tệp bên trong một tệp dùng phương thức *seek(long newPos)*.

RandomAccessFile là một lớp độc lập (kế thừa trực tiếp từ *Object*), đảm nhận việc đọc và ghi dữ liệu ngẫu nhiên. Lớp này cài đặt các giao diện *DataInput* và *DataOutput*. Sử dụng *RandomAccessFile* đòi hỏi kích thước bản ghi cố định.

Ví dụ 4.10:

```
// Lớp WriteRandomFile.java
import java.io.*;
public class WriteRandomFile {
    public static void main(String args[]) {
        int a[] = { 2, 3, 5, 7, 11, 13 };
        try {
            File fout = new File(args[0]);
            RandomAccessFile out;
            out = new RandomAccessFile(fout, "rw");
            for (int i=0; i<a.length; i++)
                out.writeInt(a[i]);
            out.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
}
// Lớp ReadRandomFile.java
import java.io.*;
public class ReadRandomFile {
    public static void main(String args[]) {
        try {
            File fin = new File(args[0]);
            RandomAccessFile in = new RandomAccessFile(fin,
"r");
            int recordNum = (int) (in.length() / 4);
            for (int i=recordNum-1; i>=0; i--) {
                in.seek(i*4);
                System.out.println(in.readInt());
            }
        }catch (IOException e) {e.printStackTrace();}
    }
}
```

+ Lớp Scanner:

Là lớp mới hỗ trợ nhập dữ liệu, kế thừa trực tiếp từ *Object*. Một đối tượng *Scanner* được tạo ra sử dụng hàm khởi tạo với đối số là đối tượng vào (luồng chuẩn, tệp, xâu ký tự...)

Có các phương thức hỗ trợ nhập trực tiếp: *nextType*, *hasNextType*

Ví dụ 4.11:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
Scanner sc = new Scanner(str);
while (sc.hasNextInt()) {
    System.out.println(sc.nextInt());
}
class Abc {
    public void read(Scanner sc) {...}
    public void doSomething() {...}
    ...
}
}
```

Lưu ý:

Khi sử dụng *Scanner* để đọc dữ liệu vào trong các biến có các kiểu dữ liệu khác nhau, đôi khi sẽ bị trôi mất dòng nhập như ví dụ dưới đây

Ví dụ 4.12: Quay trở lại với cây thừa kế Người – Cán bộ – Sinh viên đã phân tích ở các chương trước. Giả sử ta khai báo thêm lớp Cán bộ thuộc tính *luong* để quản lý lương của một Cán bộ

```
public class CanBo extends Nguoi{
    private String capHam;
    private String chucVu;
    private double luong;
    public CanBo(){
        super("", "", "", "", "", "", 0);
    }
    public CanBo(String hoTen, String ngaySinh, String gioiTinh,
String queQuan, String capHam, String chucVu, double luong) {
        super(hoTen, ngaySinh, gioiTinh, queQuan);
        this.capHam = capHam;
        this.chucVu = chucVu;
        this.luong = luong;
    }
}
```

```

        public String getCapHam() {
            return capHam;
        }
        public String getChucVu() {
            return chucVu;
        }
        public void setCapHam(String capHam) {
            this.capHam = capHam;
        }
        public void setChucVu(String chucVu) {
            this.chucVu = chucVu;
        }
        public void setLuong(double l){this.luong = l;}
        public double getLuong(){return this.luong;}
        public void inThongtin(){
            super.inThongtin();
            System.out.printf("\t Cấp hàm:%s \t Chức vụ:%s \t
Lương:%.3f
\n",this.getCapHam(),this.getChucVu(),this.getLuong());
        }
    }
}

```

Trong lớp Demo chứa hàm *main* giả sử có các khai báo như sau:

```

public class Demo2 {
    public static void main(String []args){
        CanBo[] lscb;
        int n;
        Scanner sc = new Scanner(System.in);
        System.out.println("Nhập số cán bộ: ");
        n = sc.nextInt();
        lscb = new CanBo[n];
        // Khởi tạo từng phần tử Cán bộ trong danh sách
        for(int i =0; i<n; i++)
            lscb[i] = new CanBo();
    }
}

```

Tạo một mảng Cán bộ là *lscb*. Sau khi cho người dùng nhập vào số cán bộ cần nhập dữ liệu, lưu trong biến *n* qua câu lệnh *n = sc.nextInt()* ; Các câu lệnh tiếp theo để khởi tạo mảng *lscb* gồm có *n* phần tử và từng phần tử được khởi tạo là 1 Cán bộ không có tham số.

Đến đây chương trình chưa xảy ra vấn đề. Tuy nhiên, nếu viết tiếp chương trình để nhập từng phần tử cho từng Cán bộ

```

System.out.println("Nhập từng cán bộ: ");
for(int i =0; i<n; i++){
    String tmp = new String();
    System.out.println("Nhập họ tên: ");

```

```

        //tmp = sc.nextLine();
        lscb[i].nhapTen(sc.nextLine());
        System.out.println("Nhập ngày sinh: ");
        lscb[i].nhapNgaysinh(sc.nextLine());
        System.out.println("Nhập giới tính: ");
        lscb[i].nhapGioitinh(sc.nextLine());
        System.out.println("Nhập quê quán: ");
        lscb[i].nhapQuequan(sc.nextLine());
        System.out.println("Nhập cấp hàm: ");
        lscb[i].setCapHam(sc.nextLine());
        System.out.println("Nhập chức vụ: ");
        lscb[i].setChucVu(sc.nextLine());
        System.out.println("Nhập lương: ");
        lscb[i].setLuong(sc.nextDouble());
    }
}

```

Kết quả chạy chương trình:

```

run:
Nhập số cán bộ:
3
Nhập từng cán bộ:
Nhập họ tên:
Nhập ngày sinh:

```

Lỗi khi chạy chương trình đã xảy ra, người sử dụng không thể nhập họ tên. Lỗi này do khi chúng ta nhập n bằng `n = sc.nextInt()`; người dùng nhập vào số 3 và bấm Enter. Như vậy, 3 được đưa vào n còn kí hiệu Enter vẫn còn trong dòng nhập. Do đó, khi gặp tiếp câu lệnh `lscb[i].nhapTen(sc.nextLine());` thì `sc.nextLine()` đã đọc luôn kí hiệu Enter đang có trên dòng nhập để gán cho `lscb[i].nhapTen()`. Vì thế người dùng không thể nhập họ và tên vào từ bàn phím. Đây là hiện tượng *làm trôi dòng nhập* rất phổ biến khi chúng ta nhập nhiều kiểu dữ liệu số, xâu, số...cho các bài toán.

Để khắc phục hiện tượng này, có nhiều cách để xử lý kí tự Enter kia. Tuy nhiên, cách đơn giản và hiệu quả có thể kể đến dùng các lớp quản lý các dữ liệu kiểu số như `Integer`, `Double` và phương thức `valueOf(String)` để nhập dữ liệu. Cụ thể như sau:

```

public class Demo2 {
}

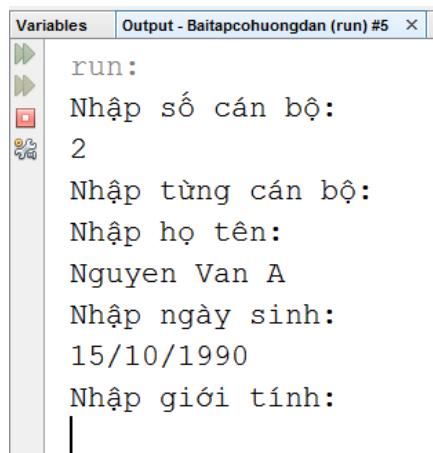
```

```

public static void main(String []args) {
    CanBo[] lscb;
    int n;
    Scanner sc = new Scanner(System.in);
    System.out.println("Nhập số cán bộ: ");
    n = Integer.valueOf(sc.nextLine());
    lscb = new CanBo[n];
    // Khởi tạo từng phần tử Cán bộ trong danh sách
    for(int i = 0; i < n; i++)
        lscb[i] = new CanBo();
    System.out.println("Nhập từng cán bộ: ");
    for(int i = 0; i < n; i++){
        String tmp = new String();
        System.out.println("Nhập họ tên: ");
        lscb[i].nhapTen(sc.nextLine());
        System.out.println("Nhập ngày sinh: ");
        lscb[i].nhapNgaysinh(sc.nextLine());
        System.out.println("Nhập giới tính: ");
        lscb[i].nhapGioitinh(sc.nextLine());
        System.out.println("Nhập quê quán: ");
        lscb[i].nhapQuequan(sc.nextLine());
        System.out.println("Nhập cấp hàm: ");
        lscb[i].setCapHam(sc.nextLine());
        System.out.println("Nhập chức vụ: ");
        lscb[i].setChucVu(sc.nextLine());
        System.out.println("Nhập lương: ");
        lscb[i].setLuong(Double.valueOf(sc.nextLine()));
    }
}

```

Kết quả chương trình cho phép nhập dữ liệu bình thường



3. Thao tác chuỗi hóa đối tượng

Một đối tượng có thể được lưu trong bộ nhớ tại nhiều vùng nhớ khác nhau. Các thuộc tính của đối tượng có thể không phải là kiểu nguyên thủy mà là kiểu tham chiếu đối tượng.

Đối tượng muốn ghi/đọc được phải thuộc lớp có cài đặt giao diện *Serializable* - đây là giao diện nhãm, không có phương thức. Một lớp khi thực thi giao diện *Serializable* thì các đối tượng thuộc lớp đó có thể được chuỗi hóa trạng thái lưu vào tệp (quá trình *Serialization*) và phục hồi lại trạng thái từ tệp (quá trình *Deserialization*)

Ví dụ 4.13:

```
// File Record.java
import java.io.Serializable;
class Record implements Serializable {
    private String name;
    private float score;
    public Record(String s, float sc) {
        name = s;
        score = sc;
    }
    public String toString() {
        return "Name: " + name + ", score: " + score;
    }
}

// File TestObjectOutputStream
import java.io.*;
public class TestObjectOutputStream {
    public static void main(String args[]) {
        Record r[] = { new Record("john", 5.0F),
                      new Record("mary", 5.5F),
                      new Record("bob", 4.5F) };
        try {
            FileOutputStream      fout      = new
FileOutputStream(args[0]);
            ObjectOutputStream     out       = new
ObjectOutputStream(fout);
            for (int i=0; i<r.length; i++)
                out.writeObject(r[i]);
            out.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
}
// TestObjectInputStream.java
import java.io.*;
public class TestObjectInputStream {
    public static void main(String args[]) {
        Record r;
        try {
            FileInputStream fin = new FileInputStream(args[0]);
            ObjectInputStream in = new ObjectInputStream(fin);
            while (true) {
```

```
        r = (Record) in.readObject();
        System.out.println(r);
    }
} catch (EOFException e) {
    System.out.println("No more records");
}
catch (ClassNotFoundException e) {
    System.out.println("Unable to create object");
}
catch (IOException e) {
    e.printStackTrace();
}
}
```

II. XỬ LÝ NGOẠI LỆ

1. Ngoại lệ

a) *Khái niệm*

Ngoại lệ (*exception*) là một sự kiện xảy ra trong quá trình thực thi chương trình (*runtime*), phá vỡ luồng bình thường của chương trình. Trong lập trình, ta có thể gặp nhiều tình huống ngoại lệ, ví dụ khi chương trình thực hiện phép chia một số nguyên dương cho 0 (ngoại lệ tính toán số học), đọc một giá trị không nguyên trong khi đang chờ đọc một giá trị kiểu *int* (ngoại lệ định dạng số), hoặc truy cập tới một phần tử không nằm trong mảng (ngoại lệ chỉ số nằm ngoài mảng), sử dụng tham chiếu đối tượng chưa được khởi gán tới đối tượng nào (*null object reference*)... Có thể xem ngoại lệ là một lỗi đặc biệt, xảy ra tại thời điểm chạy chương trình (*runtime*). Khi xảy ra một ngoại lệ, nếu không xử lý thì chương trình kết thúc ngay và trả lại quyền điều khiển cho hệ điều hành. Xét ví dụ sau:

Ví du 4.14:

```
public class Testtrycatch1{  
    public static void main(String args[]){  
        int data=50/0;//Tình huống ngoại lệ  
        System.out.println("Phan code con lai...");  
    }  
}
```

Chương trình sẽ cho kết quả sau:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
```

Trong ví dụ trên, phần còn lại của chương trình không được thực thi (trong ví dụ này là lệnh *in Phan code con lai...* không được thực hiện). Hay nói cách khác, khi xảy ra ngoại lệ của phép chia $50/0$ thì chương trình kết thúc ngay tại đây. Giả sử nếu có khoảng 100 dòng lệnh sau tình huống ngoại lệ của phép chia $50/0$, thì tất cả các dòng lệnh này sẽ không được thực thi. Để tránh việc chương trình bị phá vỡ, hay các lệnh sau tình huống ngoại lệ vẫn được thực thi, ta cần phải xử lý các ngoại lệ. Java cung cấp cơ chế hiệu quả để xử lý ngoại lệ, các cơ chế này sẽ được đề cập chi tiết ở phần sau.

b) *Phân loại*

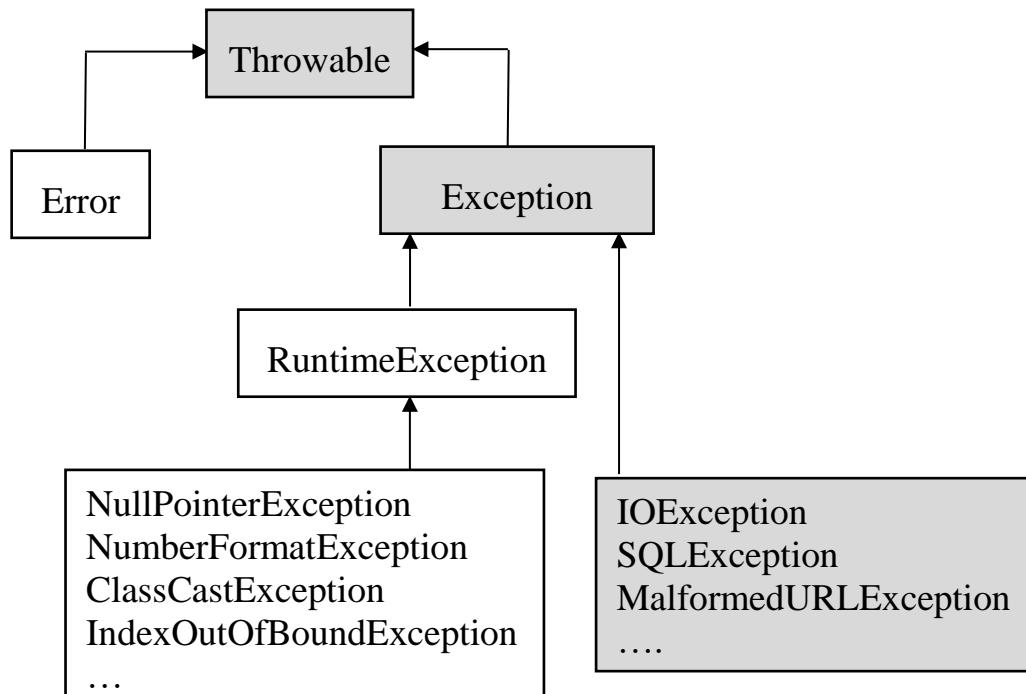
Trong ngôn ngữ Java phân biệt hai loại ngoại lệ là ngoại lệ cần kiểm tra (*Checked Exception*) và ngoại lệ không cần kiểm tra (*Unchecked Exception*).

- Ngoại lệ cần kiểm tra là các ngoại lệ xảy ra tại thời điểm biên dịch chương trình (Compile time). Những ngoại lệ này thường liên quan đến lỗi cú pháp (*syntax*) và bắt buộc chúng ta phải "*bắt*" (*catch*) nó.

- Ngoại lệ không cần kiểm tra là các ngoại lệ xảy ra tại thời điểm chạy chương trình. Những ngoại lệ này thường liên quan đến lỗi logic và không bắt buộc chúng ta phải "*bắt*" (*catch*) nó.

Như vậy, điểm khác biệt giữa các lớp ngoại lệ cần kiểm tra và ngoại lệ không cần kiểm tra chính là thời điểm xác định được ngoại lệ có thể xảy ra. Đối với ngoại lệ cần kiểm tra, việc kiểm tra được thực hiện ngay thời điểm biên dịch, và một số IDE sẽ giúp hiển thị lỗi cú pháp nếu ta gọi một phương thức ném (method *throw*) ra bất kỳ lớp ngoại lệ cần kiểm tra nào mà không được bắt ngoại lệ (*catch*). Một số lớp ngoại lệ cần kiểm tra tiêu biểu như: *IOException*, *InterruptedException*, *XMLParseException*. Đối với ngoại lệ không cần kiểm tra, việc xác định có ngoại lệ xảy ra hay không chỉ có thể thực hiện ở thời điểm thực thi chương trình, và các IDE sẽ không giúp chúng ta xác định được điều đó. Một số lớp ngoại lệ không

cần kiểm tra tiêu biểu là: *NullPointerException*, *IndexOutOfBoundsException*, *ClassCastException*. Ngoài ra, một dấu hiệu khác giúp ta phân biệt được hai loại ngoại lệ trên đó là: Tất cả các ngoại lệ cần kiểm tra được kế thừa từ lớp *Exception* ngoại trừ lớp *RuntimeException* (). *RuntimeException* là lớp cơ sở của tất cả các lớp ngoại lệ không cần kiểm tra (Hình 4.5).



Hình 4.5 Hệ thống phân cấp các lớp ngoại lệ trong Java.

Ví dụ 4.15: Ngoại lệ cần kiểm tra:

```

package vidu;

public class CheckedException {

    public static void main(String[] args) {
        System.out.println(ABC);
    }
}
  
```

Lúc này, ngay tại dòng lệnh `System.out.println(ABC);` sẽ bị lỗi. Nguyên nhân là vì lệnh trên mục đích để hiển thị chuỗi *ABC*, mà muốn hiển thị chuỗi thì bắt buộc chuỗi đó phải nằm trong cặp dấu " ". Trong câu lệnh này, ta không đặt chuỗi *ABC* trong cặp " ". Đây chính là trường hợp của ngoại lệ cần kiểm tra. Nếu

ta tiến hành chạy chương trình thì sẽ có thông báo lỗi hiển thị trong cửa sổ Console.

Ví dụ 4.16: Ngoại lệ không cần kiểm tra

```
package vidu;

public class UncheckedException {

    public static void main(String[] args) {
        int a = 5, b = 0;
        System.out.println(a/b);
    }
    // Ngoại lệ không cần kiểm tra
}
```

Lúc này, chương trình sẽ không báo lỗi gì trong đoạn chương trình trên nhưng khi biên dịch thì sẽ có thông báo lỗi "/ by zero" (lỗi chia cho 0) trong màn hình Console. Đây chính là ngoại lệ không cần kiểm tra.

c) *Ngoại lệ do người dùng tự định nghĩa (Custom Exception)*

Trong Java, ngoại lệ do người dùng tự định nghĩa hay còn gọi là loại lỗ tùy biến được sử dụng để tùy biến ngoại lệ theo yêu cầu của người dùng. Nhờ ngoại lệ này, người dùng có thể có kiểu và thông điệp ngoại lệ riêng cho mình. Thông thường, để tạo ra custom exception thuộc loại *checked* chúng ta kế thừa từ lớp *Exception*. Để tạo custom exception thuộc loại *unchecked* chúng ta kế thừa từ lớp *RuntimeException*. Trong các ứng dụng thực tế, thông thường custom exception được tạo là *checked exception*.

Ví dụ 4.17: Tạo ngoại lệ tùy biến thuộc loại checked exception:

```
class InvalidAgeException extends Exception {
    InvalidAgeException(String s) {
        super(s);
    }
}
```

và sử dụng ngoại lệ tùy biến:

```
class CustomExceptionExample {

    static void validate(int age) throws InvalidAgeException {
```

```

        if (age < 18) {
            throw new InvalidAgeException("not valid");
        } else {
            System.out.println("welcome to vote");
        }
    }

    public static void main(String args[]) {
        try {
            validate(13);
        } catch (Exception m) {
            System.out.println("Exception occurred: " + m);
        }

        System.out.println("rest of the code...");
    }
}

```

Kết quả thực thi chương trình trên như sau:

```

Exception occurred: com.gpcoder.throwex.InvalidAgeException: not
valid
rest of the code...

```

2. Xử lý ngoại lệ

Ngôn ngữ Java cung cấp cơ chế xử lý ngoại lệ rất hiệu quả. Việc xử lý này làm hạn chế tối đa trường hợp hệ thống bị hỏng (crash) hay hệ thống bị ngắt đột ngột.

Khi một ngoại lệ xảy ra, đối tượng (object) tương ứng với ngoại lệ đó được tạo ra. Đối tượng này sau đó được truyền cho phương thức là nơi mà ngoại lệ xảy ra. Đối tượng này chứa thông tin chi tiết về ngoại lệ. Thông tin này có thể được nhận và và được xử lý. Các môi trường runtime như *IllegalAccessException*, *EmptyStackException*... có thể tạo ra ngoại lệ. Chương trình đôi khi có thể tự tạo ra ngoại lệ. Lớp ‘*Throwable*’ được Java cung cấp là lớp trên cùng của lớp *Exception* (lớp đầu tiên trong cây thừa kế), lớp này là lớp cha của tất cả các ngoại lệ khác (Hình 4.5). Ngoài ra, trong Java người dùng có thể tự định nghĩa ngoại lệ, gọi là *Custom Exception*. *Custom Exception* trong Java được sử dụng để tùy biến ngoại lệ theo yêu cầu của người dùng. Với sự giúp đỡ của loại ngoại lệ này, người dùng có thể có kiểu và thông điệp ngoại lệ riêng cho mình.

Bảng 4.1 liệt kê một số ngoại lệ trong Java.

Ngoại lệ	Lớp cha của thứ tự phân cấp ngoại lệ
RuntimeException	Lớp cơ sở cho nhiều ngoại lệ java.lang
ArthmeticException	Lỗi về số học, ví dụ như ‘chia cho 0’.
IllegalAccessException	Lớp không thể truy cập.
IllegalArgumentException	Đối số không hợp lệ.
ArrayIndexOutOfBoundsException	Lỗi tràn mảng.
NullPointerException	Khi truy cập đối tượng null.
SecurityException	Cơ chế bảo mật không cho phép thực hiện.
ClassNotFoundException	Không thể nạp lớp yêu cầu.
NumberFormatException	Việc chuyển đổi từ chuỗi sang số thực không thành công.
AWTException	Ngoại lệ về AWT
IOException	Lớp cha của các lớp ngoại lệ I/O
FileNotFoundException	Không thể định vị tập tin
EOFException	Kết thúc một tập tin.
NoSuchMethodException	Phương thức yêu cầu không tồn tại.
InterruptedException	Khi một luồng bị ngắt.

Bảng 4.1. Danh sách một số ngoại lệ trong Java.

a) Mô hình xử lý ngoại lệ

Trong Java, mô hình xử lý ngoại lệ giám sát việc thực thi mã chương trình để phát hiện ngoại lệ. Mô hình xử lý ngoại lệ của Java được gọi là bắt và ném ngoại lệ (catch and throw). Trong mô hình này, khi một ngoại lệ xảy ra, ngoại lệ sẽ bị chặn và chương trình chuyển đến một khối xử lý ngoại lệ. Người lập trình

phải xử lý các ngoại lệ khác nhau có thể phát sinh trong chương trình. Các ngoại lệ phải được xử lý, hoặc thoát khỏi chương trình khi nó xảy ra.

Xử lý ngoại lệ trong Java được thực hiện theo mô hình hướng đối tượng, đó là:

- Tất cả các đối tượng ngoại lệ đều là thể hiện của một lớp kế thừa từ lớp *Throwable* hoặc các lớp con của nó (Hình 4.5).

- Các đối tượng ngoại lệ có nhiệm vụ chuyển thông tin về ngoại lệ (loại và trạng thái của chương trình) từ vị trí xảy ra ngoại lệ đến nơi quản lý/xử lý nó.

Ngôn ngữ Java cung cấp 5 từ khoá để xử lý các ngoại lệ, gồm: *try; catch; throw; throws; finally*, với các cách xử lý ngoại lệ sau:

- Sử dụng khối *try...catch* để xử lý.

- Sử dụng nhiều khối *catch* (*multipath*) để bắt nhiều ngoại lệ.

- Sử dụng khối *try...catch...finally*.

- Sử dụng *try with resource*.

- Sử dụng *Nested try* (lồng một khối *try* trong một *try* khác).

- Sử dụng từ khóa *throw* và *throws*.

b) Xử lý ngoại lệ sử dụng khối *try...catch*

Sử dụng khối *try...catch* để phân tách đoạn chương trình thông thường và phần xử lý ngoại lệ, với cú pháp như sau:

```
try {  
    // Khối lệnh có khả năng gây ra ngoại lệ;  
} catch (Exception e) {  
    //khối lệnh;  
}
```

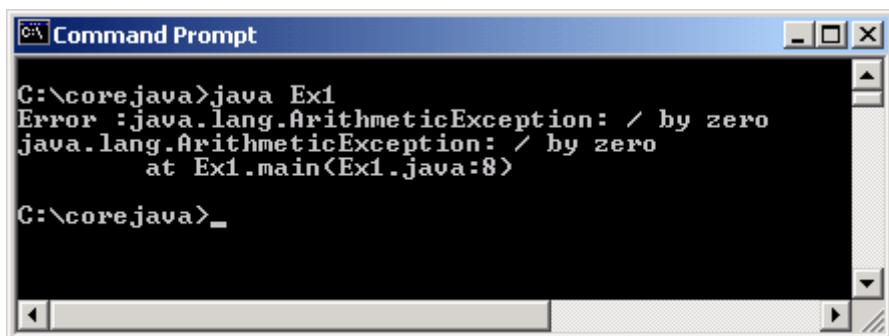
Trong cú pháp trên, các câu lệnh trong khối *try* gây ra ngoại lệ còn khối *catch* để bắt (xử lý) ngoại lệ. Ở đây, *e* là đối tượng của lớp *Exception*. Khi ngoại lệ

không biết thuộc kiểu nào, chúng ta có thể sử dụng lớp *Exception* để bắt ngoại lệ đó.

Ví dụ 4.18: Xử lý ngoại lệ với khối try...catch

```
try {
    doFileProcessing(); // phương thức do người sử dụng định nghĩa
    displayResults();
} catch (Exception e) // thể hiện của ngoại lệ
{
    System.err.println("Error :" + e.toString());
    e.printStackTrace();
}
```

Trong ví dụ trên, ta sử dụng đối tượng *e* của lớp *Exception* để in các chi tiết về ngoại lệ. Các phương thức *toString* và *printStackTrace* được sử dụng để mô tả các ngoại lệ xảy ra. Hình sau chỉ ra kết xuất của phương thức *printStackTrace()*.



Hình 4.6. Kết xuất của phương thức *printStackTrace()*.

Với *catch(Exception e)*: khối *catch()* bắt các lỗi xảy ra trong khi thi hành phương thức *doFileProcessing* hay *display*. Nếu một lỗi xảy ra trong khi thi hành phương thức *doFileProcessing()*, khi đó phương thức *displayResults()* sẽ không bao giờ được gọi. Chương trình sẽ chuyển đến thực hiện khối *catch*.

Ví dụ 4.19: Xử lý ngoại lệ với khối try...catch cho chương trình ở Error! Reference source not found.

```
public class Testtrycatch2{
    public static void main(String args[]) {
        try{
            int data=50/0;
        }catch(ArithmetricException e) {
            System.out.println(e);
        }
    }
}
```

```

        }
        System.out.println("Phan code con lai...") ;
    }
}

```

Trong ví dụ trên, lệnh gây ra ngoại lệ `int data=50/0;` được đặt và khỏi `try` và ta sử dụng đối tượng `e` của lớp ngoại lệ `ArithmaticException` để in chi tiết về ngoại lệ (`System.out.println(e);`). Chương trình trên sau khi chạy sẽ cho kết quả như sau:

```

java.lang.ArithmaticException: / by zero
Phan code con lai...

```

Như vậy, lúc này phần còn lại của chương trình sau ngoại lệ đã được thực thi (Lệnh in `System.out.println("Phan code con lai...");` đã được thực hiện).

Ví dụ 4.20: Quay trở lại với Ví dụ 4.12 sử dụng mảng để nhập vào một danh sách cán bộ. Giả sử tạo ra một lỗi cụ thể vượt quá chỉ mục của mảng

```

public class Demo2{

    public static void main(String[] args) {
        // Viết tiếp code
        try {
            System.out.println("Danh sách cán bộ: ");
            for(int i =1; i<=n; i++) {
                lscb[i].inThongtin();
            }

        } catch (ArrayIndexOutOfBoundsException e2) {
            System.out.println("Lỗi! Vượt quá chỉ mục của mảng!
" + e2);
        }
    }
}

```

Đoạn code trên đã mắc một lỗi vượt quá chỉ số của mảng do câu lệnh `for` đã cho biến `i` chạy từ 1 đến `n` trong khi mảng trong Java chỉ được chạy từ 0 đến `n-1`. Do đó ngoại lệ `ArrayIndexOutOfBoundsException` đã được tung ra. Đoạn lệnh bắt ngoại lệ này đã xử lý in ra dòng chữ: Lỗi! Vượt quá chỉ mục của mảng!
`java.lang.ArrayIndexOutOfBoundsException: 2.`

c) Xử lý ngoại lệ sử dụng nhiều khôi catch

Cú pháp thực hiện tương tự như trên, tuy nhiên sau khôi *try* là nhiều khôi *catch*, các khôi *catch* có thể được sử dụng để xử lý các loại ngoại lệ khác nhau.

```
try {  
    // đoạn mã có khả năng gây ra ngoại lệ;  
} catch(Exception e1){  
    //khôi lệnh;  
} catch(Exception e2){  
    //khôi lệnh;  
} catch(Exception eN){  
    //khôi lệnh;  
}
```

Nếu một ngoại lệ xảy ra trong khôi *try*, ngoại lệ sẽ được ném vào trong khôi *catch* thứ nhất. Nếu kiểu dữ liệu của ngoại lệ bị ném ra khớp với *ngoại_lệ_1* thì nó sẽ được bắt tại đây, nếu không ngoại lệ sẽ được chuyển đến khôi thứ hai. Việc này cứ tiếp tục đến khi ngoại lệ bị bắt bởi một khôi *catch* hoặc ngoại lệ đi qua hết các khôi *catch*.

Trước Java 7, chúng ta có thể bắt nhiều ngoại lệ bằng cách sử dụng nhiều khôi lệnh *catch*, như ví dụ dưới đây:

Ví dụ 4.21: Với cây thừa kế Người – Cán bộ - Sinh viên, với chương trình dựng lên từ Ví dụ 4.12. Giả sử ta đưa đoạn chương trình có trong Ví dụ 4.12 vào khôi *try ... multicatch* như sau:

```
public class Demo2 {  
    public static void main(String []args) {  
        try{  
            ...// Code giống ví dụ trước  
        }catch      (ArrayIndexOutOfBoundsException  
NullPointerException e) {  
            System.out.println("Có lỗi xảy ra:" + e);  
        }  
    } }
```

2 Ngoại lệ đã được xử lý trong một đoạn lệnh bằng cách ghép 2 khôi *cacth* lại làm 1 bằng đoạn code *catch (ArrayIndexOutOfBoundsException | NullPointerException e)*

d) Xử lý ngoại lệ sử dụng khối try...catch...finally

Cú pháp đầy đủ của khối *try...catch* đề cập ở trên là khối *try...catch...finally*:

```
try {  
    // đoạn mã có khả năng gây ra ngoại lệ;  
} catch (Exception e1) {  
    //khối lệnh;  
} catch (Exception e2) {  
    //khối lệnh;  
} catch (Exception eN) {  
    //khối lệnh;  
}  
finally{  
    /* khối lệnh này luôn được thực hiện cho dù ngoại lệ có xảy ra  
    hay không.  
    */  
}
```

Trong cấu trúc trên, khối *finally* khai báo các câu lệnh trả về nguồn tài nguyên cho hệ thống và in những thông báo. Khối *finally* luôn được thực thi dù cho ngoại lệ có được xử lý hay không.

Ví dụ 4.22: Tiếp tục với ví dụ 4.20, thêm đoạn code sau *finally*

```
public class Demo2 {  
    public static void main(String []args) {  
        try{  
            ...// Code giống ví dụ trước  
        } catch (ArrayIndexOutOfBoundsException  
        |  
        NullPointerException e) {  
            System.out.println("Có lỗi xảy ra:" + e);  
        }  
        finally{  
            System.out.println("Chuong trinh da chay xong");  
        }  
    }  
}
```

Khi chạy chương trình trên cho ra kết quả:

```
Baitapcohuongdan (run) #5 x | Baitapcohuongdan (run) #6 x
run:
Nhập số cán bộ:
2
Nhập từng cán bộ:
Nhập họ tên:
a
Có lỗi xảy ra:java.lang.NullPointerException
Chương trình đã chạy xong
```

Khi sử dụng các tài nguyên để vào ra với tệp thường xuyên phải đóng các tài nguyên này bằng cách sử dụng khối *finally*. Trong ví dụ dưới đây, chương trình đọc dữ liệu từ 1 file bằng cách sử dụng *FileReader* và file này được đóng bằng cách sử dụng khối *finally*.

Ví dụ 4.23:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

    public static void main(String args[]) {
        FileReader fr = null;
        try {
            File file = new File("file.txt");
            fr = new FileReader(file); char [] a = new char[50];
            fr.read(a)//đọc nội dung mảng
            for(char c : a)
                System.out.print(c); //In ra từng kí tự
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fr.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Lưu ý:

- + Đối với khối *try* có thể có nhiều khối *catch* hoặc không có, nhưng khối *finally* thì chỉ có một.

+ Khối *finally* sẽ không được thực thi nếu chương trình bị thoát (bằng cách gọi *System.exit()*) hoặc xảy ra một lỗi không thể tránh khiến chương trình bị ngừng.

+ Tất cả các khối *catch* phải được sắp xếp từ cụ thể nhất đến chung nhất. Ví dụ việc bắt *ArithimeticExption* phải ở trước việc bắt *Exception*:

```
class TestMultipleCatchBlock1{
    public static void main(String args[]) {
        try{
            int a[] = new int[5];
            a[5] = 30/0;
        } catch (Exception e) {
            System.out.println("Task chung duoc hoan thanh");
        } catch (ArithimeticException e) {
            System.out.println("Task1 duoc hoan thanh");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Task2 duoc hoan thanh");
        }
        System.out.println("Phan code con lai...");
    }
}
```

e) Xử lý ngoại lệ sử dụng *try-with-resource*

Try-with-resources còn gọi là quản lý tài nguyên tự động (*automatic resource management*), là cơ chế xử lý ngoại lệ mới được giới thiệu trên Java 7, tự động đóng các tài nguyên được sử dụng trong khối *try ... catch*. Để sử dụng lệnh này, ta cần khai báo các tài nguyên (*resource*), là một đối tượng cần phải được đóng lại mỗi khi sử dụng nó. Ví dụ: khi ta mở một file, một kết nối CSDL hoặc một *socket*, sau khi sử dụng xong, ta cần phải đóng lại ... Trước Java 7, ta thường đóng tài nguyên ở bên trong *finally*. Tuy nhiên, từ phiên bản Java 7 trở đi, Java đã tự động quản lý công việc này giúp chúng ta. Ta chỉ cần bỏ tài nguyên vào bên trong *try*, không cần phải đóng tài nguyên ở bên trong *finally* nữa.

Cú pháp lệnh *try-with-resources*:

```
try(FileReader fr = new FileReader("file path")) {
    // Các câu lệnh sử dụng tài nguyên;
} catch () {
    // Các câu lệnh;
}
```

Ví dụ 4.24: Chương trình đọc dữ liệu trong 1 file sử dụng lệnh *try-with-resources*:

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

    public static void main(String args[]) {
        try(FileReader fr = new FileReader("E://file.txt")) {
            char [] a = new char[50];
            fr.read(a); //đọc nội dung mảng
            for(char c : a)
                System.out.print(c); // in lần lượt các kí tự
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Một số chú ý khi sử dụng lệnh *try-with-resources*:

- + Để sử dụng lớp với lệnh *try-with-resources*, nó được thực hiện với *AutoCloseable interface* và phương thức *close()* được gọi tự động trong lúc chạy chương trình.
- + Có thể khai báo nhiều lớp trong lệnh *try-with-resources*.
- + Trong khi khai báo nhiều lớp trong khôi *try* của lệnh *try-with-resources*, các lớp này được đóng theo thứ tự ngược lại.
- + Ngoại trừ việc khai báo các tài nguyên trong dấu ngoặc đơn, các nội dung khác đều giống trong khôi *try...catch*.
- + Tài nguyên khai báo tại khôi *try* được ngầm khai báo là cuối cùng.

Ví dụ 4.25:

```
package com.gpcoder.exception;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryResourceExample {
```

```

        public static void main(String[] args) throws IOException{
            BufferedReader br = null;
            try {
                br = new BufferedReader(new
FileReader("D:\\gpcoder.txt"));

                String line;
                while ((line = br.readLine()) != null) {
                    System.out.println(line);
                }

            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    if (br != null)
                        br.close();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
}

```

Trong ví dụ trên, ta sử dụng một thẻ hiện của *BufferedReader* để đọc dữ liệu từ tập tin. *BufferedReader* là một tài nguyên phải được đóng sau khi chương trình kết thúc với nó. Trước Java SE 7, ta có thể sử dụng một khối *finally* để đảm bảo một tài nguyên được đóng lại bất kể câu lệnh *try* hoàn thành bình thường hay có ngoại lệ xảy ra. Tuy nhiên, từ phiên bản Java 7 trở đi, chúng ta có thể viết lại như sau:

```

package com.gpcoder.exception;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryResourceJava7Example {
    public static void main(String[] args) throws IOException{
        try (BufferedReader br = new BufferedReader(new
FileReader("D:\\gpcoder.txt"))){
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```
}
```

Trong ví dụ này, tài nguyên đã khai báo trong câu lệnh *try-with-resources* là một *BufferedReader*. Khai báo xuất hiện trong ngoặc đơn ngay sau khi thử từ khoá *try*. Lớp *BufferedReader*, trong Java SE 7 trở lên, thực hiện giao diện *java.lang.AutoCloseable*. Bởi vì thể hiện của *BufferedReader* được khai báo trong câu lệnh *try-with-resource*, nó sẽ được đóng lại bất kể câu lệnh *try* hoàn thành bình thường hay có ngoại lệ xảy ra.

Ví dụ 4.26: Sử dụng nhiều tài nguyên bên trong một khối *try-with-resources*

```
package com.gpcoder.exception;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class TryMultiResourceJava7Example {

    public static void main(String[] args) throws
FileNotFoundException, IOException {
        try (Reader reader = new FileReader("D:\\gpcoder.txt");
             BufferedReader br = new BufferedReader(reader)) {

            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        }
    }
}
```

Trong ví dụ trên ta tạo hai tài nguyên nằm trong dấu ngoặc đơn sau từ *try* là *FileReader* và *BufferedReader*. Hai tài nguyên này sẽ được đóng theo thứ tự ngược lại thứ tự mà chúng được tạo ra (liệt kê bên trong dấu ngoặc đơn): đầu tiên *BufferedReader* sẽ được đóng, sau đó đến *FileReader*.

f) Xử lý ngoại lệ sử dụng Nested try

Việc đặt một khối *try* bên trong một khối *try* khác được gọi là *Nested try*. Các khối *try* lồng nhau trong Java giúp xử lý tình huống khi một phần của một

khối mã lệnh có thể gây ra một lỗi và toàn bộ khối còn lại có thể gây ra lỗi khác. Trong các tình huống đó, việc xử lý ngoại lệ phải được lồng vào nhau.

Cú pháp:

```
try {
    lenh 1;
    lenh 2;
    try {
        lenh 1;
        lenh 2;
    } catch(Exception e) {
    }
} catch(Exception e) {
}
```

Ví dụ 4.27:

```
class Excep6{
    public static void main(String args[]){
        try{
            try{
                System.out.println("Thuc hien phep chia");
                int b =39/0;
            }catch(ArithmeticException){
                System.out.println(e);
            }
            try{
                int a[]=new int[5];
                a[5]=4;
            }catch(ArrayIndexOutOfBoundsException e){
                System.out.println(e);
            }
            System.out.println("Lenh khac");
        }catch(Exception e){
            System.out.println("Da xu ly");
        }
        System.out.println("Luong chuan..");
    }
}
```

Khi chạy chương trình trên sẽ cho kết quả như sau:

```
Thuc hien phep chia
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
Lenh khac
Luong chuan..
```

g) Xử lý ngoại lệ sử dụng từ khóa throws và throw.

- Sử dụng từ khóa throws

Không nhất thiết phải xử lý ngoại lệ trong phương thức vì một số lý do sau: không đủ thông tin để xử lý, không đủ thẩm quyền, mã rắc rối do lẩn đoạn mã xử lý lỗi và đoạn mã thuật toán điều khiển. Trường hợp này phương thức nên khai báo ném ngoại lệ bằng từ khóa *throws* để nếu khi thực hiện phương thức có phát sinh lỗi thì một đối tượng ngoại lệ tương ứng với lỗi đó sẽ được tạo ra và “ném” lên mức trên (ví dụ hàm mức trên gọi thực thi phương thức). Từ khóa *throws* được khai báo ở cuối dấu ngoặc () trước khi bắt đầu một phương thức.

Ví dụ 4.28: Trong lớp Cán bộ, khi viết các phương thức xử lý lương của 1 Cán bộ có thể ném ra ngoại lệ về định dạng

```
public class CanBo extends Nguoi{  
    ...  
    private double luong;  
    ...  
    public void setLuong(double l) throws NumberFormatException  
{this.luong = l;}  
    public double getLuong() throws NumberFormatException  
{return this.luong;}  
    ...  
}
```

Ở ví dụ trên phương thức *setLuong()*, *getLuong()* đã khai báo ném ra một ngoại lệ *NumberFormatException* lên mức trên. Đoạn lệnh này sẽ không bị xử lý ngay khi biên dịch lớp Canbo mà sẽ (có thể) bị xử lý trong lớp *Demo2*, hàm *main*.

Ta cũng có thể khai báo tung ra nhiều ngoại lệ bằng cách khai báo các ngoại lệ sau từ khóa *throws* cách nhau bởi dấu phẩy đẩy lên mức trên.

```
public void withdraw(int n) throws RemoteException,  
    InsufficientFundsException {  
    //khởi lệnh;  
}
```

- Sử dụng từ khóa *throw*

Từ khóa *throw* dùng để ném ra một ngoại lệ cụ thể. Chúng ta có thể ném một trong hai ngoại lệ *checked* hoặc *unchecked* trong java bằng từ khóa *throw*. Nhưng từ khóa *throw* cũng có thể được sử dụng để ném ngoại lệ tùy chỉnh (ngoại lệ do người dùng tự định nghĩa).

Ví dụ 4.29: Giả sử chúng ta chỉ tuyển dụng các Cán bộ đủ 18 tuổi trở lên. Do đó, khi tạo 1 cán bộ, chúng ta có 1 phương thức `public int tinhTuoi()` để tính tuổi của Cán bộ đó. Sau đó tạo phương thức `kiemtratuoi()`. Nếu tuổi dưới 18, ta ném ra ngoại lệ *ArithmeticException* thông báo chưa đủ tuổi tuyển dụng, nếu không in ra một thông báo đủ tuổi tuyển dụng. Code tiếp sau.

```
...
public int tinhTuoi() {
    int tuoi=0;
    //code tính tuổi từ Ngày sinh
    return tuoi;
}
public void kiemtratuoi() {
    if (tinhTuoi() < 18)
        throw new ArithmeticException("Chưa đủ tuổi tuyển
dụng");
    else
        System.out.println("Đủ tuổi tuyển dụng");
}
```

Sau đó, khi chạy chương trình, giả sử tuổi của Cán bộ nhập vào nhỏ hơn 18 sẽ có ngoại lệ được ném ra:

Viết tiếp code :

```
...
for(int i =0; i<n; i++) {
    lscb[i].kiemtratuoi();
    lscb[i].inThongtin();
}

}      catch      (ArrayIndexOutOfBoundsException
NullPointerException | NumberFormatException e) {
    System.out.println("Có lỗi xảy ra:" + e);
}
```

Kết quả chạy chương trình trên:

```

Baitapcohuongdan (run) #5 x Baitapcohuongdan (run) #6 x
Nhập lương:
500
Danh sách cán bộ:
Chương trình đã chạy xong
Exception in thread "main" java.lang.ArithmetricException: Chưa đủ tuổi tuyển dụng
|   at Chuong4_io_exception.CanBo.kiemtratuoi (CanBo.java:50)
|   at Chuong4_io_exception.Demo2.main (Demo2.java:47)
C:\Users\MyPC\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 30 seconds)

```

Hoặc bổ sung thêm code:

```

} catch (ArrayIndexOutOfBoundsException | NullPointerException
| NumberFormatException |ArithmetricException e) {
    System.out.println("Có lỗi xảy ra:" + e);
}

```

```

Baitapcohuongdan (run) #5 x Baitapcohuongdan (run) #6 x
tu
Nhập chức vụ:
k
Nhập lương:
500
Danh sách cán bộ:
Có lỗi xảy ra:java.lang.ArithmetricException: Chưa đủ tuổi tuyển dụng
Chương trình đã chạy xong

```

Lưu ý: Sự khác nhau giữa *throw* và *throws* trong Java:

- Từ khóa *throw* trong Java được sử dụng để ném ra một ngoại lệ rõ ràng còn từ khóa *throws* được sử dụng để khai báo một ngoại lệ.
- Ngoại lệ *checked* không được truyền ra nếu chỉ sử dụng từ khóa *throw*, tuy nhiên nó lại được truyền ra ngay cả khi chỉ sử dụng từ khóa *throws*.
- Sau *throw* là một thẻ hiện nhưng sau *throws* là một hoặc nhiều lớp.
- *throw* được sử dụng trong phương thức còn *throws* được khai báo ngay sau dấu đóng ngoặc đơn của phương thức.
- Với *throw* có thể ném nhiều ngoại lệ. Tuy nhiên, với *throws* ta có thể khai báo nhiều ngoại lệ.

Câu hỏi và bài tập cuối chương

Câu 1. Hãy viết một chương trình bắt lỗi chia cho 0 trong phép chia a/b với a và b là 2 số dương nhập vào từ bàn phím.

Câu 2. Hãy nhập vào một tên (file hoặc thư mục). Hiện ra thông báo xem:

- Đó là file hay là thư mục

- Đó là đường dẫn tương đối hay tuyệt đối

Câu 3. Cho lớp AccountRecord bao gồm có các thông tin của một tài khoản người dùng như sau:

```
public class AccountRecord{
    private int account;
    private String firstName;
    private String lastName;
    private double balance;

    // no-argument constructor calls other constructor with
    default values
    public AccountRecord(){
        this( 0, "", "", 0.0 ); // call four-argument constructor
    } // end no-argument AccountRecord constructor

    // initialize a record
    public AccountRecord( int acct, String first, String last,
    double bal ){
        setAccount( acct );
        setFirstName( first );
        setLastName( last );
        setBalance( bal );
    } // end four-argument AccountRecord constructor

    // set account number
    public void setAccount( int acct ){
        account = acct;
    } // end method setAccount

    // get account number
    public int getAccount(){
        return account;
    } // end method getAccount

    // set first name
    public void setFirstName( String first ){
        firstName = first;
    } // end method setFirstName

    // get first name
```

```

public String getFirstName() {
    return firstName;
} // end method getFirstName

// set last name
public void setLastName( String last ) {
    lastName = last;
} // end method setLastName

// get last name
public String getLastname() {
    return lastName;
} // end method getLastname

// set balance
public void setBalance( double bal ) {
    balance = bal;
} // end method setBalance

// get balance
public double getBalance() {
    return balance;
} // end method getBalance
} // end class AccountRecord

```

Hãy viết chương trình xuất 1 danh sách các account ra file clients.txt

Câu 4. Sinh một dãy số nguyên ngẫu nhiên, ghi dãy số này ra tệp “dayso.dat”. Đọc lại dãy số này từ tệp để tìm tất cả các số nguyên tố trong dãy này và ghi ra tệp “daynn.dat”

Câu 5. Sinh ngẫu nhiên một dãy số, ghi dãy số này ra tệp tin, sau đó tách tệp này thành hai tệp, một tệp chứa toàn các phần tử chẵn, tệp còn lại chứa toàn các phần tử lẻ.

Câu 6. Viết chương trình cắt một tệp thành n tệp với n là một số nguyên dương cho trước.

Câu 7. Cho một tệp văn bản có kích thước nhỏ, viết chương trình kiểm tra xem trong tệp này có chứa xâu s (cho trước) hay không?

Câu 8. Viết chương trình gộp nhiều tệp tin thành một tệp tin.

Câu 9. Sinh ngẫu nhiên một ma trận

+ Ghi ma trận này ra tệp mt.dat theo quy tắc sau:

- Hàng đầu tiên của tệp ghi số hàng của ma trận
- Hàng thứ 2 của tệp ghi số cột của ma trận
- Các hàng tiếp theo mỗi hàng ghi một hàng tương của ma trận, các phần tử trên một hàng được cách nhau bởi dấu ‘;’
- + Đọc lại ma trận này từ tệp mt.dat

Câu 10. Sinh ngẫu nhiên một dãy số

- + Ghi dãy số này ra tệp ds.dat theo quy tắc sau:
- Hàng đầu tiên của tệp ghi số phần tử của dãy số
- Hàng thứ 2 của tệp lần lượt ghi các phần tử của dãy số, các phần tử này được phân cách nhau bởi dấu cách
- + Đọc lại dãy số này từ tệp ds.dat sau đó tính tổng các phần tử và ghi vào hàng thứ 3 của tệp ds.dat

Câu 11.

- + Cho một tệp văn bản, đếm xem tệp này có bao nhiêu dòng.
- + Tìm mọi tệp tin có phần mở rộng là txt trong một thư mục cho trước

Câu 12.

- + Sinh ngẫu nhiên một ma trận
- + Ghi ma trận này ra tệp mt.dat theo quy tắc sau:
- Hàng đầu tiên của tệp ghi số hàng của ma trận
- Hàng thứ 2 của tệp ghi số cột của ma trận
- Các hàng tiếp theo mỗi hàng ghi một hàng tương của ma trận
- + Đọc lại ma trận này từ tệp mt.dat

Câu 13.

- + Tìm các số nguyên tố nhỏ hơn một số n (cho trước)

- + Ghi các số nguyên tố này ra tệp theo định dạng như sau:
 - Hàng đầu tiên của tệp ghi số lượng các số nguyên tố tìm được
 - Hàng thứ 2 của tệp lần lượt ghi số nguyên tố tìm được, các số này cách nhau bởi một dấu cách

Câu 14.

- + Tìm các số amstrong nhỏ hơn một số n (cho trước)
- + Ghi các số amstrong này ra tệp theo định dạng như sau:
 - Hàng đầu tiên của tệp ghi số lượng các số tìm được
 - Hàng thứ 2 của tệp lần lượt ghi số amstrong tìm được, các số này cách nhau bởi một dấu cách

Câu 15.

- + Tìm các số đối xứng nhỏ hơn một số n (cho trước)
- + Ghi các số tìm được ra tệp theo định dạng như sau:
 - Hàng đầu tiên của tệp ghi số lượng các số tìm được
 - Hàng thứ 2 của tệp lần lượt ghi số tìm được, các số này cách nhau bởi một dấu cách

Câu 16.

- + Tìm các số Fibonaci nhỏ hơn một số n (cho trước)
- + Ghi các số tìm được ra tệp theo định dạng như sau:
 - Hàng đầu tiên của tệp ghi số n
 - Hàng thứ 2 của tệp lần lượt ghi số tìm được, các số này cách nhau bởi một dấu cách

Chương 5. LỚP CƠ SỞ VÀ LẬP TRÌNH TỔNG QUÁT

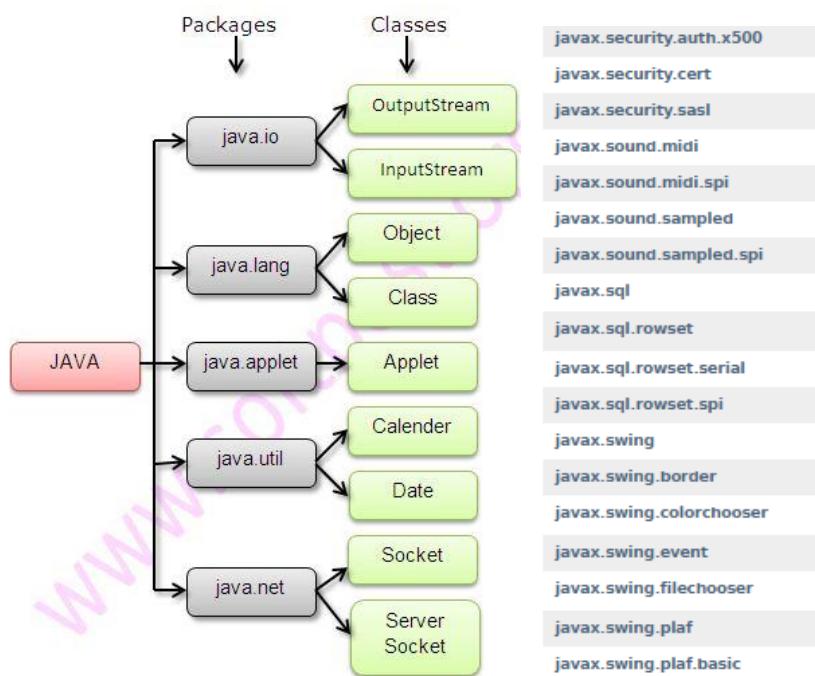
I. MỘT SỐ LỚP CƠ SỞ

1. Giới thiệu chung

Các kiểu dữ liệu của Java được chia ra làm 2 loại: dữ liệu nguyên thủy và dữ liệu tham chiếu. Các kiểu dữ liệu nguyên thủy như *int*, *float*, *char*... đã được giới thiệu trong chương 1 của giáo trình. Tất cả các dữ liệu không nguyên thủy là các dữ liệu tham chiếu, do các lớp tạo ra để xác định kiểu của đối tượng.

Thư viện của Java có rất nhiều gói thực hiện các chức năng khác nhau. Trong bản *Standard Edition 8* của Java cung cấp 2 gói chính là *java* gồm các lớp cốt lõi và *javax* gồm các lớp mở rộng. Trong 2 gói này lại có các gói con nhằm phục vụ các lớp bài toán khác nhau. Chẳng hạn *java.io* gồm các lớp phục vụ nhập, xuất dữ liệu (xem chương 4), *java.awt* chứa các lớp là các đối tượng đồ họa (chương 6), *java.lang* cung cấp các lớp để thiết kế ra được ngôn ngữ lập trình Java... Những lớp trong các gói này đều có thể được gọi là các lớp cơ sở.

Hình 5.1 dưới đây mô tả một phần các lớp cơ sở này.



Hình 5.1. Một số gói và lớp của java và javax.

Trong chương này, chúng ta sẽ tìm hiểu một số lớp cơ sở nằm trong `java.lang` và `java.util`. Những lớp được chọn lựa để giới thiệu là các lớp cần thiết để người dùng khai báo các thuộc tính cho lớp của mình.

Các lớp trong gói `java.lang` như `Object`, `String`, `Math`... là những lớp không cần phải khai báo chèn vào chương trình bằng câu lệnh *import* nhưng chương trình vẫn sử dụng được. Những lớp này trình biên dịch sẽ ngầm định tự đính kèm khi biên dịch và chạy chương trình. Trong đó `Object` là lớp gốc, cha của mọi lớp trong Java.

Ngoài ra, còn có một số lớp cần thiết gọi là các lớp tiện ích, khai báo trong `java.util` nhằm hỗ trợ người lập trình như xử lý nhập/xuất dữ liệu với *Scanner*, mô tả cấu trúc dữ liệu với các *collection* hay tổng quát hóa dữ liệu với *Generics*...

2. Lớp String

a) Giới thiệu lớp String

Trong các chương trước, khi làm quen với cây thừa kế Người – Cán bộ - Sinh viên, chúng ta đã làm quen với dữ liệu kiểu `String`. Phần này sẽ giới thiệu sâu hơn về lớp `String` này. Lớp `java.lang.String` cung cấp nhiều phương thức thao tác và xử lý với chuỗi. Với các phương thức này, chúng ta có thể thực hiện các hoạt động trên chuỗi như cắt chuỗi, nối chuỗi, chuyển đổi chuỗi, so sánh chuỗi, thay thế chuỗi...

Java String là một khái niệm mạnh mẽ bởi vì mọi thứ được đối xử như là một `String` nếu người dùng đưa ra bất cứ mẫu biểu nào trong các ứng dụng dựa trên Window, dựa trên Web hay di động. Phần dưới đây chúng tôi sẽ trình bày cách khai báo, tạo thể hiện và một số phương thức quan trọng của lớp `String`.

Lớp `String` cung cấp các *constructor* để khởi tạo các đối tượng `String` theo nhiều cách khác nhau. Có 4 cách thể hiện của các *constructor* này:

Ví dụ 5.1. Minh họa 4 cách khởi tạo khác nhau 1 xâu

```

public class StringConstructors{
    public static void main(String[] args){
        char[] charArray = {'b', 'i', 'r', 't', 'h', ' ', 'd',
        'a', 'y'};
        String s = new String("xin chao");
        String s1 = new String();
        String s2 = new String(s);
        String s3 = new String (charArray);
        String s4 = new String(charArray, 6,3)
        System.out.printf(
        "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
    }
}

```

Kết quả:

```

s1=
s2= xin chao
s3 = birth day
s4 = day

```

s được khởi tạo là một *String* bằng phương thức khởi tạo không tham số *String()*. Đối tượng *String* mới được sinh ra sẽ không chứa kí tự nào, còn gọi là xâu rỗng, tức là xâu được biểu diễn "", có độ dài bằng 0.

Tiếp theo, s2 khởi tạo một đối tượng *String* nhận tham số đầu vào là một xâu *String(String thamsodauvao)*. Đối tượng mới được sinh ra này sẽ có cùng dãy kí tự với xâu s là tham số đầu vào.

Tương tự, s3 sẽ được khởi tạo với tham số đầu vào là mảng *charArray String(char[] thamsodauvao)*. Do vậy, s3 cũng là một đối tượng chứa dãy kí tự là bản sao của mảng *charArray*.

Cuối cùng, s4 được khởi tạo là một đối tượng *String* nhận tham số đầu vào là một mảng kí tự và 2 số nguyên 6 và 3. Biến s4 sẽ chứa dãy kí tự là bản sao của mảng *charArray* nhưng chỉ copy bắt đầu từ vị trí *charArray[6]* và đến hết *charArray[8]* sang s4. Cho nên, *s4= "day"*. Đây là cách khởi tạo có dạng *String(char[] thamsodauvao, int vitribatdaucopy, int sokitucopy)* với *thamsodauvao* là mảng kí tự, *vitribatdaucopy* là vị trí bắt đầu sao chép kí tự từ mảng kí tự sang đối tượng *String* cần khởi tạo, *sokitucopy* là số kí tự được sao

chép sang. Nếu số vị trí bắt đầu + số kí tự vượt quá giới hạn chỉ số mảng thì một ngoại lệ *StringIndexOutOfBoundsException* được tung ra.

Ngoài phương thức khởi tạo, lớp *String* có một số phương thức hay sử dụng như sau:

- Phương thức *toUpperCase()* và *toLowerCase()*: Phương thức *toUpperCase()* để chuyển đổi chuỗi thành chữ hoa và phương thức *toLowerCase()* để chuyển đổi chuỗi thành chữ thường. Ví dụ:

```
String s="Vietnam";
System.out.println(s.toUpperCase());//Chuyen doi thanh VIETNAM
System.out.println(s.toLowerCase());//Chuyen doi thanh vietnam
System.out.println(s);//Vietnam(khong co thay doi nao)
```

- Phương thức *trim()*: loại bỏ các khoảng trắng ở trước và sau chuỗi (*leading* và *trailing*). Ví dụ:

```
String s=" Vietnam ";
System.out.println(s);
//in ra chuoi nhu ban dau    Vietnam     (van con khoang trang whitespace)
System.out.println(s.trim());
//in ra chuoi sau khi da cat cac khoang trong trang Vietnam
```

- Phương thức *startsWith()* và *endsWith()*:

```
String s="Vietnam";
System.out.println(s.startsWith("Vi"));
//in ra true vi bat dau xau s la xau Vi
System.out.println(s.endsWith("k"));
// in ra false vi ket thuc xau s la m khong phai k
```

- Phương thức *charAt()* : trả về ký tự tại chỉ mục đã cho. Ví dụ:

```
String s="Vietnam";
System.out.println(s.charAt(0));//tra ve V
System.out.println(s.charAt(3));//tra ve t
```

- Phương thức *length()* : trả về độ dài của chuỗi. Ví dụ:

```
String s="Vietnam";
System.out.println(s.length());//tra ve do dai la 7
```

- Phương thức `intern()` : Ban đầu, một vùng của các chuỗi là trống, được duy trì riêng cho lớp `String`. Khi phương thức `intern` được gọi, nếu vùng đã chứa một chuỗi bằng với đối tượng `String` như khi được xác định bởi phương thức `equals(object)`, thì chuỗi từ vùng đó được trả về. Nếu không thì, đối tượng `String` này được thêm vào vùng đó và một tham chiếu tới đối tượng `String` này được trả về. Ví dụ:

```
String s=new String("Vietnam");
String s2=s.intern();
System.out.println(s2); //tra ve Vietnam
```

- Phương thức `valueOf()` : trả về biểu diễn chuỗi của giá trị đã cho. Phương thức này biến đổi kiểu đã cho như `int`, `long`, `float`, `double`, `Boolean`, `char`, và mảng `char` thành chuỗi. Ví dụ:

```
int a=10;
String s=String.valueOf(a);
System.out.println(s+11); //In ra ket qua la 1011
```

- Phương thức `replace()` : thay thế tất cả sự xuất hiện của dãy ký tự đầu bởi dãy ký tự thứ hai. Ví dụ:

```
String s1="Java la mot ngon ngu lap trinh. Java la mot nen tang.
Java la mot hon dao.";
//thay the tat ca su xuat hien cua "Java" thanh "Nana"
String replaceString=s1.replace("Java", "Nana");
System.out.println(replaceString);
```

Chương trình trên sẽ cho kết quả:

```
NaNa la mot ngon ngu lap trinh. NaNa la mot nen tang. NaNa la
mot hon dao.
```

Bảng 5.1 tóm tắt các phương thức và miêu tả của phương thức `String` trong Java:

STT	Phương thức và miêu tả
1	char charAt(int index) Trả về ký tự tại chỉ mục đã cho
2	int compareTo(Object o)

	So sánh String này với đối tượng Object khác
3	int compareTo(String anotherString) So sánh hai chuỗi theo từ điển
4	int compareToIgnoreCase(String str) So sánh hai chuỗi theo từ điển, bỏ qua các sự khác nhau về kiểu chữ.
5	String concat(String str) Nối chuỗi đã cho vào phần cuối của chuỗi này
6	boolean contentEquals(StringBuffer sb) Trả về true nếu và chỉ nếu String này biểu diễn dãy liên tục các ký tự giống như StringBuffer đã cho.
7	static String copyValueOf(char[] data) Trả về một String mà biểu diễn dãy ký tự trong mảng đã cho
8	static String copyValueOf(char[] data, int offset, int count) Trả về một String mà biểu diễn dãy ký tự trong mảng đã cho.
9	boolean endsWith(String suffix) Kiểm tra nếu chuỗi này kết thúc với hậu tố đã cho
10	boolean equals(Object anObject) So sánh chuỗi này với Object đã cho
11	boolean equalsIgnoreCase(String anotherString) So sánh String với String khác, bỏ qua sự khác nhau về kiểu chữ
12	byte[] getBytes() Mã hóa String này thành dãy các byte liên tục bởi sử dụng bộ ký tự (charset) mặc định của platform, lưu giữ kết quả vào trong một mảng byte mới.
13	byte[] getBytes(String charsetName)

	Mã hóa String này thành dãy các byte liên tục bởi sử dụng bộ ký tự (charset) đã gắn tên, lưu giữ kết quả vào trong một mảng byte mới.
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Sao chép các ký tự từ chuỗi này vào trong mảng ký tự đích
15	int hashCode() Trả về một mã hóa băm (hash code) cho chuỗi này
16	int indexOf(int ch) Trả về chỉ mục trong chuỗi này với sự xuất hiện đầu tiên của ký tự đã cho
17	int indexOf(int ch, int fromIndex) Trả về chỉ mục trong chuỗi này với sự xuất hiện đầu tiên của ký tự đã cho, bắt đầu tìm kiếm tại chỉ mục đã cho.
18	int indexOf(String str) Trả về chỉ mục trong chuỗi này với sự xuất hiện đầu tiên của chuỗi phụ đã cho
19	int indexOf(String str, int fromIndex) Trả về chỉ mục trong chuỗi này với sự xuất hiện đầu tiên của chuỗi phụ đã cho, bắt đầu tìm kiếm tại chỉ mục đã cho
20	String intern() Trả về một sự biểu diễn đúng tiêu chuẩn (canonical) cho đối tượng String này
21	int lastIndexOf(int ch) Trả về chỉ mục trong chuỗi này với sự xuất hiện cuối cùng của ký tự đã cho
22	int lastIndexOf(int ch, int fromIndex)

	Trả về chỉ mục trong chuỗi này với sự xuất hiện cuối cùng của ký tự đã cho, bắt đầu tìm kiếm ngược về trước tại chỉ mục đã cho
23	int lastIndexOf(String str) Trả về chỉ mục trong chuỗi này với sự xuất hiện cuối cùng của chuỗi phụ đã cho
24	int lastIndexOf(String str, int fromIndex) Trả về chỉ mục trong chuỗi này với sự xuất hiện cuối cùng của chuỗi phụ đã cho, bắt đầu tìm kiếm ngược về trước tại chỉ mục đã cho.
25	int length() Trả về độ dài của chuỗi này
26	boolean matches(String regex) Có hay không chuỗi này so khớp (match) với regular expression đã cho
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) Kiểm tra nếu hai chuỗi là bằng nhau
28	boolean regionMatches(int toffset, String other, int ooffset, int len) Kiểm tra nếu hai chuỗi là bằng nhau
29	String replace(char oldChar, char newChar) Trả về một chuỗi mới kết quả từ việc thay thế tất cả oldchar trong chuỗi này bằng newchar
30	String replaceAll(String regex, String replacement) Thay thế mỗi chuỗi phụ trong chuỗi này mà so khớp với regular expression với chuỗi thay thế đã cho
31	String replaceFirst(String regex, String replacement)

	Thay thế chuỗi phụ đầu tiên của chuỗi này mà so khớp với regular expression đã cho với chuỗi thay thế đã cho
32	String[] split(String regex) Chia chuỗi này xung quanh các so khớp của Regex đã cho
33	String[] split(String regex, int limit) Chia chuỗi này xung quanh các so khớp của Regex đã cho
34	boolean startsWith(String prefix) Kiểm tra nếu chuỗi bắt đầu với tiền tố đã cho.
35	boolean startsWith(String prefix, int toffset) Kiểm tra nếu chuỗi bắt đầu với tiền tố đã cho bắt đầu từ chỉ mục đã cho
36	CharSequence subSequence(int beginIndex, int endIndex) Trả về một dãy ký tự mới mà là một dãy phụ của dãy này
37	String substring(int beginIndex) Trả về một chuỗi mới mà là một chuỗi phụ của chuỗi này
38	String substring(int beginIndex, int endIndex) Trả về một chuỗi mới mà là một chuỗi phụ của chuỗi này
39	char[] toCharArray() Biến đổi chuỗi này thành một mảng ký tự mới
40	String toLowerCase() Biến đổi tất cả ký tự trong String này thành kiểu chữ thường bởi sử dụng các qui tắc của locale mặc định
41	String toLowerCase(Locale locale) Biến đổi tất cả ký tự trong String này thành kiểu chữ thường bởi sử dụng các qui tắc của locale đã cho
42	String toString() Đối tượng này (mà đã là một chuỗi) được trả về chính nó
43	String toUpperCase()

	Biến đổi tất cả ký tự trong String này thành kiểu chữ hoa bởi sử dụng các qui tắc của locale mặc định
44	StringtoUpperCase(Locale locale) Biến đổi tất cả ký tự trong String này thành kiểu chữ hoa bởi sử dụng các qui tắc của locale đã cho
45	String trim() Trả về một bản sao của chuỗi, với các khoảng trắng ban đầu và kết thúc bị bỏ qua
46	static String valueOf(primitive data type x) Trả về biểu diễn chuỗi của tham số kiểu dữ liệu đã truyền.

Bảng 5.1. Bảng các phương thức của lớp String.

b) Sử dụng lớp String

Lớp String thường được sử dụng trong các bài toán cần xử lý dữ liệu đầu vào như nhập danh sách, nhập số liệu từ tệp, lấy về một địa chỉ của trang Web...Trong đó chúng ta cần sử dụng String để đọc chuỗi kí tự, sau đó cắt chuỗi, ghép chuỗi, kiểm tra độ dài chuỗi hoặc chuyển đổi chuỗi thành số hoặc ngược lại, kiểm tra lỗi chính tả...

Ví dụ 5.2. Nhập vào một xâu. In ra độ dài xâu

```
import java.util.Scanner;
public static void main(String[] args) {
    String chuoi;
    int doDai;
    Scanner scanner = new Scanner(System.in);
    System.out.println("Nhập vào chuỗi bất kỳ từ bàn phím: ");
    chuoi = scanner.nextLine();
    // tính độ dài chuỗi
    doDai = chuoi.length();
    System.out.println("Chuỗi " + chuoi + " có độ dài = " +
doDai); }
```

Kết quả:

```
Nhập vào chuỗi bất kỳ từ bàn phím:  
hello  
Chuỗi hello có độ dài = 5
```

Hình 5.2. Sử dụng hàm Scanner và nextLine() để nhập chuỗi.

Lưu ý: Các đối tượng kiểu *String* có trạng thái không thể thay đổi được (còn gọi là *Immutable Object*). Không cần thiết để sao chép một đối tượng *String* đã có sẵn. Các đối tượng *String* có các kí tự của nó là không đổi sau khi được khởi tạo.

Ví dụ:

```
String s = "Hello";  
s.concat("java");  
System.out.println(s); // sẽ chỉ in ra hello
```

Còn nếu gán lại s như sau:

```
String s = "Hello";  
s = s.concat(" java");  
System.out.println(s); // sẽ in ra Hello java
```

Nhưng chuỗi “Hello” vẫn nằm trong phần bộ nhớ quản lý hằng chuỗi “Hello”. Phần s trả tới sẽ là vùng nhớ của hằng chuỗi + vùng nhớ chứa “java”.

Nếu truy xuất đến chỉ số mảng nhỏ hơn 0 và lớn hơn *s.length()* thì sẽ có một ngoại lệ được ném ra: *String-IndexOutOfBoundsException*

Ví dụ 5.3. Viết chương trình so sánh xâu

```
/*String methods equals, equalsIgnoreCase, compareTo and  
regionMatches.  
*/  
public class StringCompare {  
  
    public static void main( String args[] ) {  
        String s1 = new String( "hello" );  
        // s1 is a copy of "hello"  
        String s2 = "goodbye";  
        String s3 = "Happy Birthday";  
        String s4 = "happy birthday";  
        System.out.printf( "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n\n", s1, s2, s3, s4 );  
  
        // test for equality  
        if ( s1.equals( "hello" ) ) // true  
            System.out.println( "s1 bằng \"hello\"" );  
    }  
}
```

```

        else System.out.println( "s1 không bằng \"hello\"");

        // test for equality with ==
        if ( s1 == "hello" ) // false; khong cung object
            System.out.println("s1 không cùng đối tượng với xâu
\"hello\"");
            else System.out.println("s1 không cùng đối tượng với xâu
\"hello\"");

        // test for equality (ignore case)
        if ( s3.equalsIgnoreCase( s4 ) ) // true
            System.out.printf("%s equals %s with case
ignored\n", s3, s4 );
            else System.out.println( "s3 does not equal s4" );

        // test compareTo
            System.out.printf("\ns1.compareTo( s2 ) is %d",
s1.compareTo( s2 ) );
            System.out.printf("\ns2.compareTo( s1 ) is %d",
s2.compareTo( s1 ) );
            System.out.printf("\ns1.compareTo( s1 ) is %d",
s1.compareTo( s1 ) );
            System.out.printf("\ns3.compareTo( s4 ) is %d",
s3.compareTo( s4 ) );
            System.out.printf("\ns4.compareTo( s3 ) is %d\n\n",
s4.compareTo( s3 ) );

        // test regionMatches (case sensitive)
        if ( s3.regionMatches( 0, s4, 0, 5 ) )
            System.out.println( "First 5 characters of s3 and s4
match" );
            else System.out.println( "First 5 characters of s3 and
s4 do not match" );

        // test regionMatches (ignore case)
        if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
            System.out.println( "First 5 characters of s3 and s4
match" );
            else System.out.println( "First 5 characters of s3 and
s4 do not match" );
        } // end main
    } // end class StringCompare

```

Kết quả:

```

s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

s1.equals("hello")
s1 is not the same object as "hello"
Happy Birthday equals happy birthday with case ignored

s1.compareTo(s2) is 1
s2.compareTo(s1) is -1
s1.compareTo(s1) is 0
s3.compareTo(s4) is -32
s4.compareTo(s3) is 32

First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match

```

Hình 5.3. Kết quả so sánh xâu.

Lưu ý: So sánh hai biến tham chiếu với phép == có thể dẫn tới lỗi logic. Bởi vì so sánh == chỉ so sánh xem 2 biến tham chiếu đó có cùng chỉ tới 1 đối tượng hay không chứ không so sánh xem liệu 2 đối tượng đó có cùng nội dung hay không. Khi hai đối tượng khác nhau được so sánh bởi ==, kết quả trả về sẽ là *false*. Cho nên, khi so sánh hai đối tượng liệu có cùng nội dung hay không phải sử dụng phương thức `equals()`.

Ví dụ 5.4. Lấy một xâu con từ một xâu to

```

// String class substring methods.
public class SubString {

    public static void main( String args[] ) {
        String letters = "abcdefghijklmabcdefghijklm";

        // test substring methods
        System.out.printf( "Substring from index 20 to end is "
"\\"%s\"\n", letters.substring( 20 ) );
        System.out.printf( "%s \"%s\"\n", "Substring from index"
3 up to, but not including 6 is",
            letters.substring( 3, 6 ) );
    } // end main
} // end class SubString

```

Kết quả lấy xâu con:

```

Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including 6 is "def"

```

Ví dụ 5.5. Minh họa một số phương thức *replace*, *toLowerCase*, *toUpperCase*, *trim* và *toCharArray*

```
// String methods replace, toLowerCase, toUpperCase, trim and
// toCharArray.
public class StringMiscellaneous2 {

    public static void main( String args[] ){
        String s1 = new String( "hello" );
        String s2 = new String( "GOODBYE" );
        String s3 = new String( " spaces " );
        System.out.printf( "s1 = %s\ns2 = %s\ns3 = %s\n\n", s1,
s2, s3 );

        // test method replace
        System.out.printf( "Replace 'l' with 'L' in s1: %s\n\n",
s1.replace( 'l', 'L' ) );

        // test toLowerCase and toUpperCase
        System.out.printf( "s1.toUpperCase() = %s\n", s1.toUpperCase() );
        System.out.printf( "s2.toLowerCase() = %s\n\n", s2.toLowerCase() );

        // test trim method
        System.out.printf( "s3 after trim = \"%s\"\n\n", s3.trim() );

        // test toCharArray method
        char charArray[] = s1.toCharArray();
        System.out.print( "s1 as a character array = " );
        for ( char character : charArray )
            System.out.print( character );
        System.out.println();
    } // end main
} // end class StringMiscellaneous2
```

Kết quả:

```
s1 = hello
s2 = GOODBYE
s3 = spaces

Replace 'l' with 'L' in s1: heLLo

s1.toUpperCase() = HELLO
s2.toLowerCase() = goodbye

s3 after trim = "spaces"

s1 as a character array = hello
```

3. Lớp Math

a) Giới thiệu lớp Math

Lớp `java.lang.Math` chứa các phương thức thực hiện các phép toán cơ bản như mũ, logarit, bình phương và lượng giác.

Lớp Math này có định nghĩa tại `java.lang` như sau:

```
public final class Math extends Object
```

Các trường của lớp Math gồm có:

```
Static double E: Đây là giá trị double gần với số e nhất.  
Static double PI: Đây là giá trị double gần với số pi nhất.
```

Các phương thức của lớp này được liệt kê trong Bảng 5.2.

STT	Phương thức và mô tả
1	static double abs(double a) Phương thức này trả lại giá trị tuyệt đối của một số thực kiểu double.
2	static float abs(float a) Phương thức này trả lại giá trị tuyệt đối của một số thực kiểu float
3	static int abs(int a) Phương thức này trả lại giá trị tuyệt đối của một số nguyên kiểu int
4	static long abs(long a) Phương thức này trả lại giá trị tuyệt đối của một số nguyên kiểu long
5	static double acos(double a) Phương thức này trả lại arccos của một giá trị, góc trả về trong khoảng từ 0.0 đến pi
6	static double asin(double a) Phương thức này trả lại arcsin của một giá trị, góc trả về trong khoảng từ pi/2 đến pi/2

	static double atan(double a)
7	Phương thức này trả lại arctan của một giá trị, góc trả về trong khoảng từ pi/2 đến pi/2
8	static double atan2(double y, double x) Phương thức này trả lại góc theta từ phép chuyển tọa độ Decartes (x,y) sang tọa độ cực (r, theta)
9	static double cbrt(double a) Phương thức này trả về giá trị lập phương của một giá trị double.
10	static double ceil(double a) trả về giá trị double là số làm tròn tăng bằng giá trị số nguyên gần nhất
11	static double copySign(double magnitude, double sign) Phương thức này trả lại đối số double đầu tiên với dấu là dấu của đối số double thứ hai trong danh sách tham số truyền vào.
12	static float copySign(float magnitude, float sign) Phương thức này trả lại đối số float đầu tiên với dấu là dấu của đối số float thứ hai trong danh sách tham số truyền vào.
13	static double cos(double a) Phương thức này trả lại giá trị lượng giác cosin của một góc.
14	static double cosh(double x) Phương thức này trả lại giá trị cosin hyperbol của một giá trị double.
15	static double exp(double a) Phương thức này trả lại giá trị e^a
16	static double expm1(double x) Phương thức này trả lại $e^x - 1$
17	static double floor(double a) Phương thức này trả lại giá trị lớn nhất mà gần bằng a.
18	static int getExponent(double d) Phương thức này trả về số mũ không thiêng vị được sử dụng trong biểu diễn của một double.
19	static int getExponent(float f) Phương thức này trả về số mũ không thiêng vị được sử dụng trong biểu diễn của một float
20	static double hypot(double x, double y)

	Phương thức này trả về $\sqrt{x^2 + y^2}$ mà không có tràn hoặc tràn trung gian.
21	static double IEEEremainder(double f1, double f2) Phương pháp này tính toán hoạt động còn lại trên hai đối số theo quy định của tiêu chuẩn IEEE 754.
22	static double log(double a) Phương thức này trả về logarit tự nhiên (cơ số e) của một giá trị kép.
23	static double log10(double a) Phương thức này trả về logarit cơ số 10 của một giá trị kép.
24	static double log1p(double x) Phương thức này trả về logarit tự nhiên của tổng của đối số và 1.
25	static double max(double a, double b) Phương thức này trả về giá trị lớn hơn của hai giá trị double.
26	static float max(float a, float b) Trả về số lớn hơn trong 2 số thực float a và b
27	static int max(int a, int b) Trả về số lớn hơn trong 2 số nguyên int a và b
28	static long max(long a, long b) Trả về số lớn hơn trong 2 số nguyên long a và b
29	static double min(double a, double b) Phương thức này trả về giá trị nhỏ hơn của hai giá trị double.
30	static float min(float a, float b) Trả về số nhỏ hơn trong 2 số thực float a và b
31	static int min(int a, int b) Trả về số nhỏ hơn trong 2 số nguyên int a và b
32	static long min(long a, long b) Trả về số nhỏ hơn trong 2 số nguyên long a và b
33	static double nextAfter(double start, double direction)

	Phương thức này trả về số double bên cạnh đối số đầu tiên start và theo hướng của đối số thứ hai.
34	static float nextAfter(float start, double direction) Phương thức này trả về số dấu chấm động bên cạnh đối số đầu tiên start và theo hướng của đối số thứ hai.
35	static double nextUp(double d) Phương thức này trả về giá trị double liền kề với d theo hướng vô cùng dương.
36	static float nextUp(float f) Phương thức này trả về giá trị dấu chấm động liền kề với d theo hướng vô cùng dương.
37	static double pow(double a, double b) Phương thức này trả về a^b
38	static double random() Phương thức này trả về 1 số ngẫu nhiên trong khoảng 0.0 đến 1.0
39	static double rint(double a) Phương thức này trả về giá trị double có giá trị nguyên gần a nhất
40	static long round(double a) Phương thức này trả về số long làm tròn số a
41	static int round(float a) Phương thức này trả về số int làm tròn số a
42	static double scalb(double d, int scaleFactor) phương thức này trả về $d \times 2^{scaleFactor}$ được làm tròn
43	static float scalb(float f, int scaleFactor) phương thức này trả về $d \times 2^{scaleFactor}$ được làm tròn
44	static double signum(double d) Phương thức này trả về hàm signum của đối số double d, Nếu d =0, trả lại 0, nếu d > 0 trả về 1.0, nếu d <0 trả về -1.0

45	static float signum(float f) Phương thức này trả về hàm signum của đối số float f, Nếu d =0, trả lại 0, nếu d > 0 trả về 1.0, nếu d <0 trả về -1.0
46	static double sin(double a) Phương thức này trả về sin lượng giác của góc a với a là góc radian.
47	static double sinh(double x) Phương thức này trả về sin hyperbol của số x.
48	static double sqrt(double a) Phương thức này trả về căn bậc hai của số a.
49	static double tan(double a) Phương thức này trả về tang lượng giác của góc a.
50	static double tanh(double x) Phương thức này trả về tang hyperbol của số x.
51	static double toDegrees(double angrad) Phương thức này chuyển đổi góc angrad từ radian sang xấp xỉ bằng một giá trị độ nào đó.
52	static double toRadians(double angdeg) Phương thức này chuyển đổi góc angrad từ độ sang xấp xỉ bằng một giá trị radian nào đó.
53	static double ulp(double d) Phương thức này trả lại kích thước của một ulp là số thực double d
54	static double ulp(float f) Phương thức này trả lại kích thước của một ulp là số thực float f.

Bảng 5.2. Các phương thức của Math.

b) Sử dụng lớp Math

Hầu hết các phương thức của Math là *static*. Vì vậy không cần phải tạo thể hiện của lớp này mà gọi trực tiếp `Math.<tên lớp>(<danh sách đối số truyền vào nếu có>)`.

Dưới đây sẽ trình bày một số ví dụ tiêu biểu khi sử dụng các thuộc tính và phương thức của lớp Math.

Ví dụ 5.6. *Computer* là hằng số trong thời gian chạy do `Math.random` khởi tạo giá trị.

```
final int computer =(int) (Math.random()*20);
```

Ví dụ 5.7. Sử dụng hàm `pow` tính a^b với a, b nhập từ bàn phím

```
import static java.lang.System.*;
import java.util.Scanner;

public class MathPower {

    public static Scanner scan;
    public static void main(String[] args) {
        // ask for user input
        out.print("Nhập số cơ sở:");
        scan = new Scanner(System.in);
        // use scanner to get user console input, base
        double baseNumber = scan.nextDouble();
        // use scanner to get user console input, exponent
        System.out.print("Nhập số mũ:");
        scan = new Scanner(System.in);
        double exponent = scan.nextDouble();
        // get the result
        double result = Math.pow(baseNumber, exponent);
        out.println("Số mũ = "+result);
        // close the scanner object to avoid memory leak
        scan.close();
    }
}
```

4. Các collection

a) Giới thiệu các collection

Trong phần này, chúng ta sẽ tìm hiểu về *Java collections framework*, là một khuôn mẫu chứa các cấu trúc dữ liệu đã được đóng gói, các giao diện *Interface* và các thuật toán để làm việc với các cấu trúc dữ liệu. Ví dụ về các *collection* là các thẻ bài trong bộ bài, các bài hát trong một album, các thành viên trong một đội thể thao... Với các *collection* này, người lập trình sử dụng các cấu trúc dữ liệu có sẵn, không cần phải quan tâm nó đã được cài đặt như thế nào. Bản thân *collection* có nghĩa là *bộ sưu tập, hoặc tập hợp*. Nhưng để tránh trùng lặp với *Set* là cấu trúc dữ liệu kiểu *tập hợp* nên chúng ta vẫn sử dụng thuật ngữ *collection*.

Java cung cấp các lớp đặc biệt như *Dictionary*, *Vector*, *Stack* và *Properties* để lưu trữ và thao tác với một nhóm các đối tượng. Mặc dù những lớp này khá hữu dụng, nhưng lại thiếu sự tập trung, thống nhất. Do đó, cách sử dụng *Vector* trong Java khác với cách sử dụng *Properties*.

Khuôn mẫu *collection* (*collection framework*) được thiết kế nhằm đạt được các mục đích như sau:

- Khuôn mẫu phải đạt hiệu năng cao. Tức là khi triển khai các tập hợp cơ bản như các mảng động, danh sách liên kết *linked list*, cây *tree* và bảng băm *hashtable* các khuôn mẫu này được sử dụng với hiệu quả cao.
- Khuôn mẫu phải cho phép các kiểu *collection* khác nhau nhưng làm việc theo một cách tương tự như nhau với độ phân hóa ở mức cao.
- Kế thừa và/hoặc tìm hiểu với các *collection* phải dễ dàng.

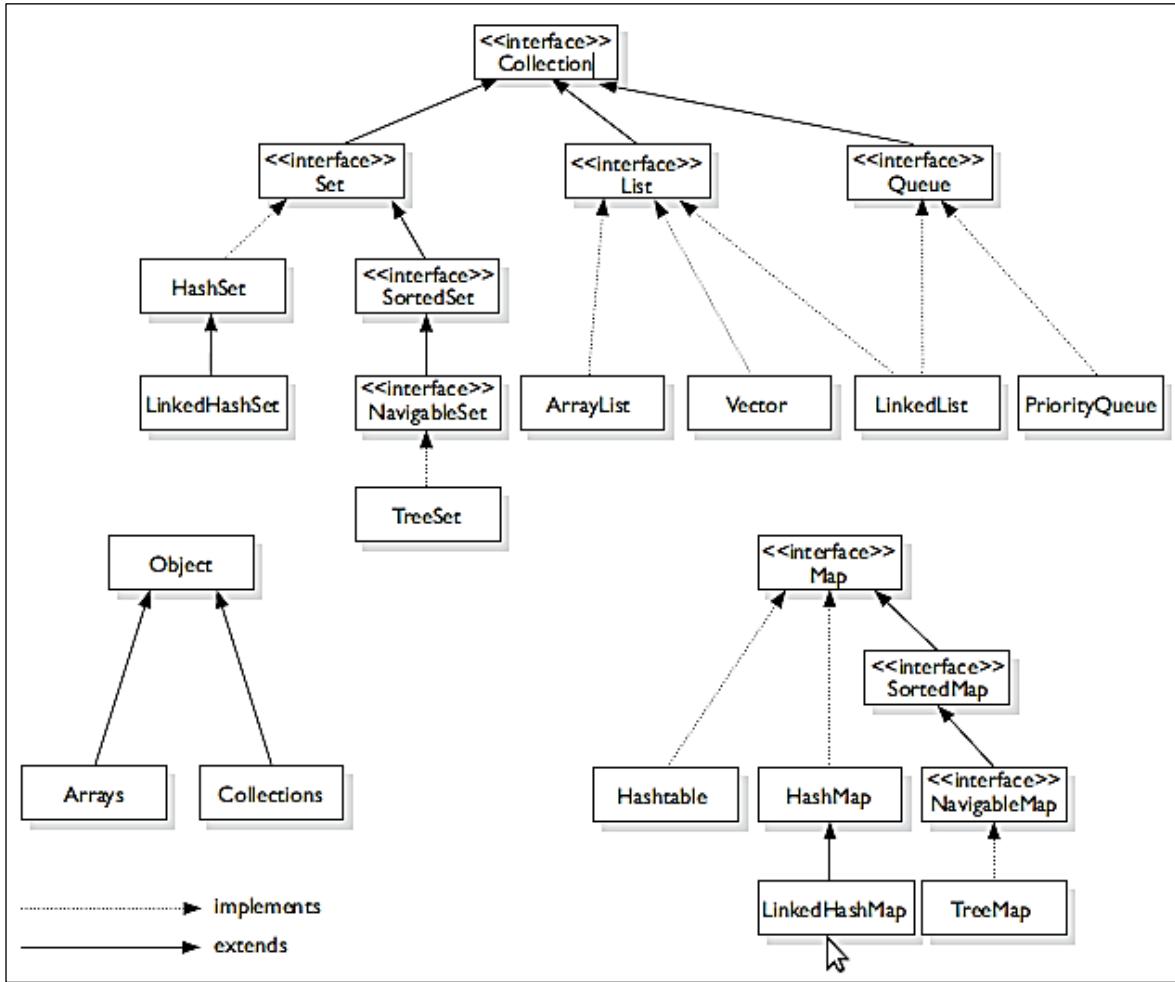
Cuối cùng, toàn bộ khuôn mẫu *collection* này được thiết kế một tập các giao diện *interface* tiêu chuẩn. Một số lớp triển khai như *LinkedList*, *HashSet* và *TreeSet* của những giao diện này được cung cấp sẵn cho người dùng có thể triển khai *collection* nếu được chọn.

Tóm lại, về mặt cấu trúc dữ liệu, một khuôn mẫu *collection* chứa:

- Giao diện *interface*;
- Sự triển khai, ví dụ như các lớp;
- Thuật toán: Đây là các phương thức thực hiện các trình tính toán hữu ích, như tìm kiếm và phân loại, trên các đối tượng triển khai *collection interface*. Các thuật toán được xem như là đa hình, đó là, cùng một phương thức có thể được sử dụng trên nhiều sự triển khai khác nhau của *collection interface*.

Ngoài ra, khuôn mẫu định nghĩa một số giao diện *map* và lớp *Map* lưu giữ các cặp *key/value*. Mặc dù các *map* không là các *collection* về khái niệm, nhưng chúng hoàn toàn tương thích với *collection*.

Hình vẽ dưới đây tổng hợp khuôn mẫu *collection*:



Hình 5.4. Sơ đồ khuôn mẫu *collection*.

Các lớp và giao diện của khuôn mẫu *collection* được cung cấp trong gói *java.util*. Các phiên bản Java cũ, các lớp trong *collection* được tham chiếu từ đối tượng *Object*. Vì vậy người lập trình có thể lưu trữ bất kì đối tượng nào trong một *collection*. Một trong những bất tiện của việc lưu trữ các tham chiếu của *Object* là truy xuất các đối tượng này trong *collection*. Do đó, các tham chiếu của *Object* đến bài toán cụ thể sẽ phải ép kiểu thành kiểu thích hợp để xử lý.

Đến bản J2SE5.0, khuôn mẫu *collection* đã được cải tiến bằng lập trình tổng quát sử dụng *generic* (sẽ được giới thiệu ở phần sau). Có nghĩa là người lập trình có thể xác định chính xác kiểu của thành viên trong một *collection*. Điều này dẫn tới lợi ích về biên dịch, kiểm tra lỗi. Người lập trình xác định kiểu trong *collection*,

tham chiếu phù hợp với kiểu đó. Do đó làm giảm bớt các câu lệnh ép kiểu tường minh mà có thể ném ra ngoại lệ *ClassCastException*.

b) Các giao diện - Interface của collection

Các giao diện được liệt kê trong Bảng 5.3.

Interface	Description
Collection	Đây là giao diện gốc trong cây phân cấp Collection derived.
Set	Một collection trong đó không có thành viên nào giống nhau
List	Một collection có sắp xếp có thể chứa các thành viên giống nhau.
Map	Chỉ ra các khóa key tương ứng với các giá trị value và không có một khóa nào trùng nhau.
Queue	Interface diễn hình theo kiểu vào trước – ra trước, mô tả một hàng đợi; các thứ tự khác có thể xác định được

Bảng 5.3. Bảng các Interface collection.

Một số cài đặt của các giao diện này đã được cung cấp trong khuôn mẫu của *collection*. Nhưng người lập trình vẫn có thể tạo ra các cài đặt của riêng mình. Chúng ta sẽ đi tìm hiểu từng loại giao diện nói trên:

- Giao diện Collection

Giao diện *Collection* định nghĩa những phương thức cơ bản khi làm việc với tập hợp, đây là gốc để từ đó xây dựng lên cả bộ thư viện Java *Collection Framework*. *Collection* được kế thừa từ giao diện *Iterable* nên có thể dễ dàng duyệt qua từng phần tử thông qua việc sử dụng *Iterator*. Có nhiều phương thức được khai báo trong *Collection*.

Phương thức	Mô tả
public boolean add(Object element)	Được sử dụng để chèn một phần tử vào collection.
public boolean addAll(Collection c)	Được sử dụng để chèn các phần tử collection được chỉ định vào collection gọi phương thức này.

public boolean remove(Object element)	Được sử dụng để xóa phần tử từ collection.
public boolean removeAll(Collection c)	Được sử dụng để xóa tất cả các phần tử của collection được chỉ định từ collection gọi phương thức này.
public boolean retainAll(Collection c)	Được sử dụng để xóa tất cả các thành phần từ collection gọi phương thức này, ngoại trừ collection được chỉ định.
public int size()	Trả lại tổng số các phần tử trong collection.
public void clear()	Loại bỏ tổng số của phần tử khỏi collection.
public boolean contains(Object element)	Được sử dụng để tìm kiếm phần tử.
public boolean containsAll(Collection c)	Sử dụng để tìm kiếm collection được chỉ định trong collection.
public Iterator iterator()	Trả về một iterator.
public Object[] toArray()	Chuyển đổi collection thành mảng (array).
public boolean isEmpty()	Kiểm tra nếu collection trống.
public boolean equals(Object element)	So sánh 2 collection.
public int hashCode()	Trả về số hashcode của collection.

Bảng 5.4. Các phương thức của interface collection.

- Giao diện Set

Set (tập hợp) là kiểu dữ liệu mà bên trong nó mỗi phần tử chỉ xuất hiện duy nhất một lần (tương tự như tập hợp trong toán học vậy) và *Set Interface* cung cấp các phương thức để tương tác với một tập hợp. *Set* được kế thừa từ *Collection* nên nó cũng có đầy đủ các phương thức của *Collection Interface*. Một số lớp thực thi *Set interface* thường gặp:

- + *TreeSet*: là 1 lớp thực thi giao diện *Set Interface*, trong đó các phần tử trong tập hợp đã được sắp xếp.
- + *HashSet*: là 1 lớp implement *Set Interface*, mà các phần tử được lưu trữ dưới dạng bảng băm (*hash table*).
- + *EnumSet*: là 1 lớp dạng tập hợp như 2 lớp ở trên, tuy nhiên khác với 2 lớp trên là các phần tử trong set là các dữ liệu kiểu *enum*.

- Giao diện *List*

List (danh sách) là cấu trúc dữ liệu tuyến tính trong đó các phần tử được sắp xếp theo một thứ tự xác định. *List interface* định nghĩa các phương thức để tương tác với danh sách cũng như các phần tử bên trong danh sách. Tương tự như *Set*, *List* cũng được kế thừa và có đầy đủ các phương thức của *Collection*. Một số lớp thực thi *List interface* thường sử dụng:

- + *ArrayList*: là 1 lớp dạng danh sách được cài đặt dựa trên mảng có kích thước thay đổi được.
- + *LinkedList*: là một lớp dạng danh sách hoạt động trên cơ sở của cấu trúc dữ liệu danh sách liên kết đôi (*double-linked list*).
- + *Vector*: là 1 lớp thực thi giao diện *interface List* lưu trữ như mảng tuy nhiên có kích thước thay đổi được, tương tự với *ArrayList*, tuy nhiên điểm khác biệt là *Vector* là *synchronized* (đồng bộ), có thể hoạt động đa luồng mà không cần gọi *synchronize* một cách tường minh. Chi tiết về luồng và lập trình đa luồng tham khảo thêm ở tài liệu “*Java How to program, 10 Edition, Chapter 23*”.
- + *Stack*: cũng là 1 lớp dạng danh sách, *Stack* có cách hoạt động dựa trên cơ sở của cấu trúc dữ liệu ngăn xếp (*stack*) với kiểu vào ra LIFO (*last-in-first-out* hay vào sau ra trước) nổi tiếng.

- Giao diện *Queue*

Queue (hàng đợi) là kiểu dữ liệu nổi tiếng với kiểu vào ra FIFO (*first-in-first-out* hay vào trước ra trước), tuy nhiên với *Queue interface* thì hàng đợi không chỉ còn dừng lại ở mức đơn giản như vậy mà nó cung cấp các phương thức để xây dựng các hàng đợi phức tạp hơn nhiều như *priority queue* (hàng đợi có ưu tiên), *dequeue* (hàng đợi 2 chiều).. Cũng giống như 2 *interface* trước, *Queue interface* cũng kế thừa và mang đầy đủ phương thức từ *Collection interface*. Một số lớp về hàng đợi thường sử dụng:

- + *LinkedList* : danh sách liên kết.
- + *PriorityQueue*: là 1 dạng hàng đợi mà trong đó các phần tử trong hàng đợi sẽ được sắp xếp.
- + *ArrayDeque*: là 1 dạng *dequeue* (hàng đợi 2 chiều) được cài đặt dựa trên mảng.

- Giao diện *Map*

Map (đồ thị/ánh xạ) là kiểu dữ liệu cho phép quản lý dữ liệu theo dạng cặp *key-value*, trong đó *key* là giá trị duy nhất; tương ứng với 1 *key* là một giá trị *value*. *Map interface* cung cấp các phương thức để tương tác với kiểu dữ liệu như vậy. Không giống như các *interface* ở trên, *Map interface* không kế thừa từ *Collection interface* mà đây là một *interface* độc lập với các phương thức của riêng mình. Dưới đây là một số lớp về *map* cần chú ý:

- + *TreeMap*: là lớp thực thi giao diện *Map interface* với dạng cây đỏ đen (*Red-Black tree*) trong đó các *key* đã được sắp xếp. Lớp này cho phép thời gian thêm, sửa, xóa và tìm kiếm 1 phần tử trong *Map* là tương đương nhau và đều là $O(\log(n))$.
- + *HashMap*: là lớp thực thi giao diện *Map interface* với các *key* được lưu trữ dưới dạng bảng băm, cho phép tìm kiếm nhanh $O(1)$.
- + *EnumMap*: cũng là 1 lớp *map*, tuy nhiên các *key* là các *enum* chứ không phải đối tượng như các dạng ở trên. Phần *enum* tham khảo thêm ở tài liệu “*Java Howto program, 10th Edition, Chapter 6, 6.10*”.

+ *WeakHashMap*: tương tự như *HashMap* tuy nhiên có 1 điểm khác biệt đáng chú ý là các *key* trong *map* này chỉ là các *Weak reference* (hay *Weak key*), có nghĩa là khi phần tử sẽ bị xóa khi *key* được giải phóng, hay không còn một biến nào tham chiếu đến *key* nữa.

c) Các lớp collection trong Java

Java cung cấp một tập hợp các lớp *collection* tiêu chuẩn có thể triển khai các *Collection interface*. Bảng 5.5 tổng hợp các lớp *collection* chuẩn.

STT	Các lớp và miêu tả
1	Lớp AbstractCollection trong Java Triển khai tất cả các Collection interface
2	Lớp AbstractList trong Java Kế thừa AbstractCollection và triển khai tất cả phương thức List interface
3	Lớp AbstractSequentialList trong Java Kế thừa AbstractList để sử dụng bởi một Collection mà sử dụng liên tục thay vì truy cập ngẫu nhiên các phần tử của nó
4	Lớp LinkedList trong Java Triển khai một LinkedList bởi kế thừa AbstractSequentialList
5	Lớp ArrayList trong Java Triển khai một mảng động bởi kế thừa AbstractList
6	Lớp AbstractSet trong Java Kế thừa AbstractCollection và triển khai hầu hết Set interface
7	Lớp HashSet trong Java Kế thừa AbstractSet để sử dụng với một hash table
8	Lớp LinkedHashSet trong Java Kế thừa HashSet để cho phép lặp lại thứ tự chèn (insertion-order)
9	Lớp TreeSet trong Java

	Triển khai một tập hợp được lưu trong một tree. Ké thừa AbstractSet
10	Lớp AbstractMap trong Java Triển khai hầu hết Map interface
11	Lớp HashMap trong Java Ké thừa AbstractMap để sử dụng một hash table
12	Lớp TreeMap trong Java Ké thừa AbstractMap để sử dụng một tree
13	Lớp WeakHashMap trong Java Ké thừa AbstractMap để sử dụng một hash table với các khóa weak
14	Lớp LinkedHashMap trong Java Ké thừa HashMap để cho phép lặp lại thứ tự chèn (insertion-order)
15	Lớp IdentityHashMap trong Java Ké thừa AbstractMap và sử dụng tham chiếu ngang bằng khi so sánh các tài liệu

Bảng 5.5. Các lớp collection.

Dưới đây chúng ta sẽ tập trung phân tích một số lớp quan trọng:

- Lớp *LinkedList* kế thừa lớp *AbstractSequentialList* và triển khai *List interface*. Nó cung cấp một cấu trúc dữ liệu danh sách liên kết.

Ví dụ 5.8. Chương trình sau minh họa các phương thức được hỗ trợ bởi lớp *LinkedList* trong Java. Chương trình viết tiếp code cho Người – Cán bộ - Sinh viên

```
import java.util.*;

public class Demo2 {
    public static void main(String []args) {
        LinkedList ll = new LinkedList();
        CanBo cb1 = new CanBo("Nguyễn Văn A", "15/10/1990", "nam", "HN", "TU", "K", 500);
        ll.add(cb1);
        System.out.println(ll);
    }
}
```

```

        CanBo cb2 = new CanBo("Trần Thị B", "01/03/1990", "nữ", "HN", "TU", "K", 500);
        ll.add(cb1);
        ll.add(cb2);

        System.out.println("Xuất danh sách liên kết: ");
        for(int i=0; i<2; i++)
        { CanBo temp = (CanBo) ll.get(i);
            temp.inThongtin();
        }
        System.out.println("Nội dung ban đầu của ds lk: "+ll);
        System.out.println("Xoa các phần tử của ds lk");
        ll.remove(cb1);
        ll.remove(0);
        System.out.println("Nội dung sau xóa của ds lk: "+ll);

        for(int i=0; i<2; i++)
            ll.add(i, cb1);
        CanBo temp =(CanBo) ll.removeFirst();
        temp.inThongtin();
        temp =(CanBo) ll.removeLast();
        temp.inThongtin();
    }
}

```

Kết quả:

- Lớp *ArrayList* trong Java kế thừa *AbstractList* và triển khai *List interface*. Lớp *ArrayList* hỗ trợ các mảng động mà có thể tăng kích cỡ nếu cần.

Các mảng Java chuẩn có độ dài cố định. Sau khi các mảng được tạo, chúng không thể tăng hoặc giảm kích cỡ, nghĩa là bạn phải có bao nhiêu phần tử mà một mảng sẽ giữ.

ArrayList được tạo với một kích cỡ ban đầu. Khi kích cỡ này bị vượt, *collection* tự động được tăng. Khi các đối tượng bị gỡ bỏ, *ArrayList* có thể bị giảm kích cỡ.

Ví dụ 5.9. Minh họa các phương thức được hỗ trợ bởi lớp *ArrayList*. Viết tiếp cây Người- Cán bộ - Sinh viên. Lúc này lớp main sẽ sử dụng một *ArrayList*

ls để tiếp tục thêm các Cán bộ từ mảng `lscb[]` bằng phương thức `add()` và lấy ra các Cán bộ bằng phương thức `get(index)` để in thông tin.

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

/**
 *
 * @author admin
 */
public class Demo2 {
    public static void main(String []args) {
        CanBo[] lscb;
        int n;
        Scanner sc = new Scanner(System.in);
        System.out.println("Nhập số cán bộ: ");
        n = Integer.valueOf(sc.nextLine());
        lscb = new CanBo[n];
        try{

            // Khởi tạo từng phần tử Cán bộ trong danh sách
            for(int i =0; i<n; i++){
                lscb[i] = new CanBo();
                System.out.println("Nhập từng cán bộ: ");
                for(int i =0; i<n; i++){
                    String tmp = new String();
                    System.out.println("Nhập họ tên: ");
                    lscb[i].nhapTen(sc.nextLine());
                    System.out.println("Nhập ngày sinh: ");
                    lscb[i].nhapNgaysinh(sc.nextLine());
                    System.out.println("Nhập giới tính: ");
                    lscb[i].nhapGioitinh(sc.nextLine());
                    System.out.println("Nhập quê quán: ");
                    lscb[i].nhapQuequan(sc.nextLine());
                    System.out.println("Nhập cấp hàm: ");
                    lscb[i].setCapHam(sc.nextLine());
                    System.out.println("Nhập chức vụ: ");
                    lscb[i].setChucVu(sc.nextLine());
                    System.out.println("Nhập lương: ");
                    lscb[i].setLuong(Double.valueOf(sc.nextLine()));
                }
            }

        }      catch      (ArrayIndexOutOfBoundsException |  
NullPointerException | NumberFormatException | ArithmeticException  
e) {
            System.out.println("Có lỗi xảy ra:" + e);
        }

        finally{
            System.out.println("Chương trình đã chạy xong");
        }
    }
    ArrayList ls = new ArrayList();
    for(int i =0; i<n; i++)
```

```

        ls.add(lscb[i]);
        System.out.println("In ra ArrayList:");
        for(int i=0; i< ls.size(); i++)
        {
            CanBo cb = (CanBo)ls.get(i);
            cb.inThongtin();
        }
    }
}

```

Kết quả:

```

Baitapcohuongdan (run) #5 | Baitapcohuongdan (run) #7 x
Nhập cнuc vну:
k
Nhập lуong:
500
Chương trinh dа chay xong
In ra ArrayList:
Họ và tên: a Ngày sinh: 1 Giới tính: na Quê quán:hn Cấp hàm:tu Chức vụ:k Lương:500.000
Họ và tên: b Ngày sinh: 2 Giới tính: nu Quê quán:hd Cấp hàm:tu Chức vụ:k Lương:500.000
BUILD SUCCESSFUL (total time: 30 seconds)

```

- Lớp *HashSet* trong Java kế thừa *AbstractSet* và triển khai *Set interface*. Nó tạo một *collection* mà sử dụng một bảng băm để lưu giữ. Lưu ý trong *HashSet* chỉ chứa các phần tử duy nhất, không chấp nhận 2 phần tử trùng nhau.

Một bảng băm lưu giữ thông tin bởi sử dụng một kỹ thuật được gọi là *băm hashing*. Trong *hashing*, nội dung mang tính thông tin của một *key* được sử dụng để quyết định một *value* duy nhất, được gọi là mã băm *hash code* của nó.

Hash code sau đó được sử dụng như là chỉ số *index*, tại đó dữ liệu liên kết với *key* được lưu giữ. Phép biến đổi của *key* vào trong *hash code* của nó được thực hiện tự động.

Ví dụ 5.10. Minh họa các phương thức được hỗ trợ bởi lớp *HashSet*

```

import java.util.*;
public class HashSetDemo {

    public static void main(String args[]) {
        // tao mot hash set
        HashSet hs = new HashSet();
        // them cac phan tu toi hash set
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
    }
}

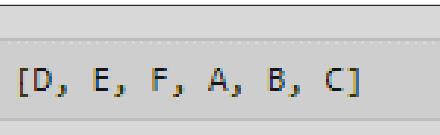
```

```

        hs.add("F");
        hs.add("F");
        //them gia tri F 2 lan nhung ket qua chi xuat hien 1 lan
        System.out.println(hs);
    }
}

```

Kết quả:



- Lớp *TreeSet* trong Java cung cấp một sự triển khai của *Set interface* mà sử dụng một cây cho lưu trữ. Các đối tượng được lưu giữ được xếp thứ tự tăng dần.

Thời gian truy cập và thu nhận dữ liệu là khá nhanh, làm cho *TreeSet* là một lựa tốt khi lưu giữ một lượng lớn thông tin đã xếp thứ tự cần phải được tìm kiếm một cách nhanh chóng.

Ví dụ 5.11. Minh họa các phương thức được hỗ trợ bởi lớp *TreeSet*

```

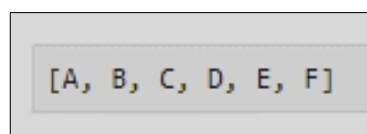
import java.util.*;

public class TreeSetDemo {

    public static void main(String args[]) {
        // Tao mot tree set
        TreeSet ts = new TreeSet();
        // them cac phan tu toi tree set
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
        System.out.println(ts); }
}

```

Kết quả:



- Lớp *HashMap* trong Java sử dụng một bảng băm *hashtable* để triển khai *Map interface*. Điều này cho phép thời gian thực thi các hoạt động cơ bản, như *get()* và *put()*.

Ví dụ 5.12.

```
import java.util.*;

public class HashMapDemo {

    public static void main(String args[]) {
        // Tao mot hash map
        HashMap hm = new HashMap();
        // Dat cac phan tu vao map
        hm.put("Zara", new Double(3434.34));
        hm.put("Mahnaz", new Double(123.22));
        hm.put("Ayan", new Double(1378.00));
        hm.put("Daisy", new Double(99.22));
        hm.put("Qadir", new Double(-19.08));
        // Lay mot tap hop cac entry
        Set set = hm.entrySet();
        // Lay mot iterator
        Iterator i = set.iterator();
        // Hien thi cac phan tu
        while(i.hasNext()) {
            Map.Entry me = (Map.Entry)i.next();
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();
        // Gui 1000 vao trong tai khoan cua Zara
        double balance = ((Double)hm.get("Zara")).doubleValue();
        hm.put("Zara", new Double(balance + 1000));
        System.out.println("Balance hien tai cua Zara la: " +
        hm.get("Zara"));
    }
}
```

Kết quả:

```
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0
Qadir: -19.08

Balance hien tai cua Zara la: 4434.34
```

- Lớp *TreeMap* trong Java triển khai *Map interface* bởi sử dụng một cây. Một *TreeMap* cung cấp các phương thức hiệu quả để lưu giữ các cặp *key/value* trong thứ tự được sắp xếp, và cho phép thu hồi nhanh chóng.

Nên chú ý rằng, không giống một sơ đồ băm *hash map*, một *tree map* bảo đảm rằng các phần tử của nó sẽ được xếp thứ tự theo thứ tự *key* tăng dần.

Ví dụ 5.13.

```
import java.util.*;  
  
public class TreeMapDemo {  
  
    public static void main(String args[]) {  
        // Tao mot hash map  
        TreeMap tm = new TreeMap();  
        // Dat cac phan tu vao map  
        tm.put("Zara", new Double(3434.34));  
        tm.put("Mahnaz", new Double(123.22));  
        tm.put("Ayan", new Double(1378.00));  
        tm.put("Daisy", new Double(99.22));  
        tm.put("Qadir", new Double(-19.08));  
  
        // Lay mot tap hop cac entry  
        Set set = tm.entrySet();  
        // Lay mot iterator  
        Iterator i = set.iterator();  
        // Hien thi cac phan tu  
        while(i.hasNext()) {  
            Map.Entry me = (Map.Entry)i.next();  
            System.out.print(me.getKey() + ": ");  
            System.out.println(me.getValue());  
        }  
        System.out.println();  
        // Gui 1000 vao trong tai khoan cua Zara  
        double balance = ((Double)tm.get("Zara")).doubleValue();  
        tm.put("Zara", new Double(balance + 1000));  
        System.out.println("Balance hien tai cua Zara la: " +  
            tm.get("Zara"));  
    }  
}
```

Như đã nói ở trên, phiên bản Java đầu tiên không bao gồm khuôn mẫu *Collection framework*. Thư viện Java định nghĩa một vài lớp và giao diện cung cấp các phương thức để lưu trữ các đối tượng. Khi *Collection framework* được thêm vào trong J2SE 1.2, các lớp gốc đã được tái cấu trúc để hỗ trợ các *collection*.

interface. Các lớp này còn được gọi là lớp *legacy*. Tất cả các lớp và giao diện *legacy* được thiết kế lại bởi JDK 5 để hỗ trợ lớp tổng quát *Generics*.

- Các lớp *legacy*:

+ Lớp *Vector*: Lớp *Vector* trong Java triển khai một mảng động. Nó tương tự như *ArrayList*, nhưng với hai điểm khác biệt:

- * Vector được đồng bộ synchronization.
- * Vector chứa các phương thức *legacy*, là các phương thức không là một phần của Collection framework.

Lưu ý: Sử dụng lớp *Vector* khi thực hiện các bài toán liên quan đến danh sách hoặc mảng mà chúng ta chưa rõ kích cỡ của mảng hoặc cần thay đổi kích cỡ thông qua vòng đời của một chương trình.

Ví dụ 5.14. Sử dụng *Vector* tạo danh sách các phần tử với nhiều kiểu số

```
import java.util.*;  
  
public class VectorDemo {  
    public static void main(String args[]) {  
        // capacity ban dau la 3, incr la 2  
        Vector v = new Vector(3, 2);  
        System.out.println("Size ban dau: " + v.size());  
        System.out.println("Capacity ban dau: " +  
v.capacity());  
        v.addElement(new Integer(1));  
        v.addElement(new Integer(2));  
        v.addElement(new Integer(3));  
        v.addElement(new Integer(4));  
        System.out.println("Capacity sau 4 lan cong la: " +  
v.capacity());  
        v.addElement(new Double(5.45));  
        System.out.println("Capacity hien tai: " + v.capacity());  
        v.addElement(new Double(6.08));  
        v.addElement(new Integer(7));  
        System.out.println("Capacity hien tai: " + v.capacity());  
        v.addElement(new Float(9.4));  
        v.addElement(new Integer(10));  
        System.out.println("Capacity hien tai: " + v.capacity());  
        v.addElement(new Integer(11));  
        v.addElement(new Integer(12));  
        System.out.println("Phan tu dau tien: " +  
(Integer)v.firstElement());  
        System.out.println("Phan tu cuoi cung: " +  
(Integer)v.lastElement());  
    }  
}
```

```

        if(v.contains(new Integer(3)))
            System.out.println("Vector chua 3.");
        // tinh toan so phan tu trong vector.
        Enumeration vEnum = v.elements();
        System.out.println("\nCac phan tu trong Vector:");
        while(vEnum.hasMoreElements())
            System.out.print(vEnum.nextElement() + " ");
        System.out.println();}}
```

Kết quả:

```

Size ban dau: 0
Capacity ban dau: 3
Capacity sau 4 lan cong la: 5
Capacity hien tai: 5
Capacity hien tai: 7
Capacity hien tai: 9
Phan tu dau tien: 1
Phan tu cuoi cung: 12
Vector chua 3.

Cac phan tu trong Vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

Phân tích: `Vector v = new Vector(3, 2);` tạo 1 thê hiện *Vector* *v* có dung lượng là 3, số phần tử được cấp phát mỗi khi *v* thay đổi kích thước là 2. Đến câu lệnh `v.addElement(new Integer(3));` đã dùng hết 3 phần tử đã có ở *v*. Nên gấp câu `v.addElement(new Integer(4))` *v* phải thay đổi kích thước, khi đó nó sẽ tự động thêm 2 phần tử nữa vào danh sách. Do vậy dung lượng của *v* lúc này sẽ là $3 + 2 = 5$ tức là nó chưa được tối đa 5 phần tử. Các trường hợp sau suy luận tương tự.

+ Lớp *Stack*: Lớp *Stack* là triển khai một danh sách kiểu *last-in-first-out* (LIFO) tức là vào sau ra trước. Đây là danh sách kiểu ngăn xếp, trong đó phần tử đầu tiên được đẩy vào đáy ngăn xếp, các phần tử tiếp theo đẩy vào lần lượt cho đến đỉnh ngăn xếp. Các phần tử lấy ra được lấy từ đỉnh, xuống dần đến đáy ngăn xếp. Lớp *Stack* của Java được thừa kế từ lớp *Vector*.

Ví dụ 5.15.

```

import java.util.*;

public class StackDemo {
    static void showpush(Stack st, int a) {
```

```

        st.push(new Integer(a));
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }
    static void showpop(Stack st) {
        System.out.print("pop -> ");
        Integer a = (Integer) st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }
    public static void main(String args[]) {
        Stack st = new Stack();
        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);
        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("empty stack");
        }
    }
}

```

Kết quả:

```

stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack

```

+ Lớp Dictionary: Lớp Dictionary trong Java là một lớp ảo biểu diễn một kho lưu giữ các cặp key/value và hoạt động khá giống lớp Map trong Java. Với một cặp key/value đã cung cấp, có thể lưu giữ giá trị trong đối tượng Dictionary. Khi giá trị được lưu, truy cập giá trị value bởi sử dụng key của nó. Như vậy, giống

như lớp *Map*, một *Dictionary* có thể được xem như là một danh sách các cặp *key/value*.

+ Lớp *Hashtable*: Lớp *Hashtable* trong Java là một phần của *java.util* gốc và là một sự triển khai cụ thể của một *Dictionary*.

Tuy nhiên, Java 2 đã thiết kế lại *Hashtable* để nó cũng triển khai *Map interface*. Vì thế, lớp *Hashtable* bây giờ được tích hợp vào trong *Collection Framework*. Nó tương tự như *HashMap*, nhưng nó được đồng bộ *synchronization*. Giống như *HashMap*, lớp *Hashtable* lưu giữ các cặp *key/value* trong một bảng băm. Khi sử dụng một *Hashtable*, trình biên dịch đã xác định một đối tượng mà được sử dụng như là một *key*, và *value* là giá trị muốn liên kết tới *key* đó. *Key* này sau đó được băm, và giá trị băm kết quả được sử dụng như là chỉ mục, tại đó *value* được lưu giữ bên trong bảng băm đó.

Ví dụ 5.16. Sử dụng *Hashtable*

```
import java.util.*;  
  
public class HashTableDemo {  
  
    public static void main(String args[]) {  
        // tao mot hash map  
        Hashtable balance = new Hashtable();  
        Enumeration names;  
        String str;  
        double bal;  
  
        balance.put("Zara", new Double(3434.34));  
        balance.put("Mahnaz", new Double(123.22));  
        balance.put("Ayan", new Double(1378.00));  
        balance.put("Daisy", new Double(99.22));  
        balance.put("Qadir", new Double(-19.08));  
  
        // Hien thi tat ca balance trong hash table.  
        names = balance.keys();  
        while(names.hasMoreElements()) {  
            str = (String) names.nextElement();  
            System.out.println(str + ": " +  
                balance.get(str));  
        }  
        System.out.println();  
        // Gui 1,000 vao tai khoan cua Zara  
        bal = ((Double)balance.get("Zara")).doubleValue();  
        balance.put("Zara", new Double(bal+1000));  
    }  
}
```

```

        System.out.println("Balance moi cua Zara la: " +
balance.get("Zara"));
    }
}

```

Kết quả:

```

Qadir: -19.08
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0

Balance moi cua Zara la: 4434.34

```

+ Lớp *Properties*: Lớp *Properties* trong Java là lớp con của lớp *Hashtable*.

Nó được sử dụng để duy trì các danh sách của các *value* trong đó *key* là một *String* và *value* cũng là một *String*. Lớp *Properties* được sử dụng bởi nhiều lớp Java khác. Ví dụ, nó là kiểu đối tượng được trả về bởi `System.getProperties()` khi đạt được các *value* môi trường.

+ Lớp *BitSet*: Lớp *BitSet* trong Java tạo một kiểu mảng đặc biệt mà giữ các giá trị *bit*. Mảng *BitSet* này có thể tăng giảm kích cỡ nếu cần. Điều này làm nó tương tự như một *vector* của các *bit*. Đây là một lớp *legacy* nhưng nó đã hoàn toàn được thiết kế lại trong Java 2, phiên bản 1.4.

d) Một số phương thức với collection

Collection framework định nghĩa một số thuật toán có thể được áp dụng cho các *collection* và *map*. Những thuật toán này được định nghĩa như là các phương thức tĩnh *static* bên trong lớp *Collection*.

Một số phương thức có thể ném một ngoại lệ *ClassCastException*, xảy ra khi có gắng so sánh các kiểu không tương thích, hoặc một *UnsupportedOperationException*, xảy ra khi cố gắng sửa đổi một *Unmodifiable Collection*.

Các *Collection* định nghĩa 3 biến static là: *EMPTY_SET*, *EMPTY_LIST*, và *EMPTY_MAP*. Tất cả là không thể thay đổi.

Bảng 5.6 liệt kê các phương thức được định nghĩa trong thuật toán của *Collection framework* trong Java.

STT	Phương thức và Miêu tả
1	static int binarySearch(List list, Object value, Comparator c) Tìm kiếm value trong list được sắp xếp theo c. Trả về vị trí của value trong list; hoặc trả về -1 nếu value không được tìm thấy
2	static int binarySearch(List list, Object value) Tìm kiếm value trong list. List phải được xếp thứ tự. Trả về vị trí của value trong list; trả về -1 nếu không tìm thấy value
3	static void copy(List list1, List list2) Sao chép các phần tử của list2 vào list1
4	static Enumeration enumeration(Collection c) Trả về một bản liệt kê qua c
5	static void fill(List list, Object obj) Gán obj tới mỗi phần tử của list
6	static int indexOfSubList(List list, List subList) Tìm kiếm trong list về sự xuất hiện đầu tiên của subList. Trả về chỉ mục của so khớp đầu tiên; trả về -1 nếu không có so khớp được tìm thấy
7	static int lastIndexOfSubList(List list, List subList) Tìm kiếm trong list về sự xuất hiện cuối cùng của subList. Trả về chỉ mục của so khớp đầu tiên; trả về -1 nếu không có so khớp được tìm thấy
8	static ArrayList list(Enumeration enum) Trả về một ArrayList mà chứa các phần tử của enum
9	static Object max(Collection c, Comparator comp) Trả về phần tử tối đa trong c như được xác định bởi comp
10	static Object max(Collection c) Trả về phần tử tối đa trong c như được xác định bởi thứ tự tự nhiên. Collection không cần xếp theo thứ tự

STT	Phương thức và Miêu tả
11	static Object min(Collection c, Comparator comp) Trả về phần tử tối thiểu trong c như được xác định bởi comp. Collection không cần xếp theo thứ tự
12	static Object min(Collection c) Trả về phần tử tối thiểu trong c như được xác định bởi thứ tự tự nhiên
13	static List nCopies(int num, Object obj) Trả về num bản sao của obj được chứa trong một list không đổi. num phải lớn hơn hoặc bằng 0
14	static boolean replaceAll(List list, Object old, Object new) Thay thế tất cả old bởi new trong list. Trả về true nếu có ít nhất một sự thay thế xảy ra. Nếu không là false
15	static void reverse(List list) Đảo ngược dãy trong list
16	static Comparator reverseOrder() Trả về một comparator đảo ngược
17	static void rotate(List list, int n) Quay list bởi n vị trí tới bên phải. Để quay sang bên trái, sử dụng một giá trị âm cho n
18	static void shuffle(List list, Random r) Xáo trộn các phần tử trong list bởi sử dụng r như là một nguồn của các số ngẫu nhiên
19	static void shuffle(List list) Xáo trộn các phần tử trong list
20	static Set singleton(Object obj) Trả về obj như là một set không thay đổi. Đây là một cách dễ dàng để biến đổi một đối tượng đơn vào trong một set
21	static List singletonList(Object obj) Trả về obj như là một list không thay đổi. Đây là một cách dễ dàng để biến đổi một đối tượng đơn vào trong một list

STT	Phương thức và Miêu tả
22	static Map singletonMap(Object k, Object v) Trả về cặp key/value (k/v) như là một map không thay đổi. Đây là một cách dễ dàng để biến đổi một cặp key/value đơn vào trong một map
23	static void sort(List list, Comparator comp) Xếp thứ tự các phần tử trong list như đã xác định bởi comp
24	static void sort(List list) Xếp thứ tự các phần tử trong list như đã xác định bởi thứ tự tự nhiên
25	static void swap(List list, int idx1, int idx2) Trao đổi các phần tử trong list tại các chỉ mục được xác định bởi idx1 và idx2
26	static Collection synchronizedCollection(Collection c) Trả về một Collection an toàn luồng (thread-safe) bởi c
27	static List synchronizedList(List list) Trả về một thread-safe list bởi list
28	static Map synchronizedMap(Map m) Trả về một thread-safe map bởi m
29	static Set synchronizedSet(Set s) Trả về một thread-safe set bởi s
30	static SortedMap synchronizedSortedMap(SortedMap sm) Trả về một thread-safe sorted set bởi sm
31	static SortedSet synchronizedSortedSet(SortedSet ss) Trả về một thread-safe set bởi ss
32	static Collection unmodifiableCollection(Collection c) Trả về một unmodifiable collection bởi c
33	static List unmodifiableList(List list) Trả về một unmodifiable list bởi list
34	static Map unmodifiableMap(Map m) Trả về một unmodifiable map bởi m
35	static Set unmodifiableSet(Set s)

STT	Phương thức và Miêu tả
	Trả về một unmodifiable set bởi s
36	static SortedMap unmodifiableSortedMap(SortedMap sm) Trả về một unmodifiable sorted map bởi sm
37	static SortedSet unmodifiableSortedSet(SortedSet ss) Trả về một unmodifiable sorted set bởi ss

Bảng 5.6. Bảng các phương thức mô tả thuật toán trong collection.

Ví dụ 5.17. Minh họa các thuật toán của *Collection* trong Java

```

import java.util.*;

public class AlgorithmsDemo {
    public static void main(String args[]) {
        // Tao va khai tao linked list
        LinkedList ll = new LinkedList();
        ll.add(new Integer(-8));
        ll.add(new Integer(20));
        ll.add(new Integer(-20));
        ll.add(new Integer(8));

        // Ta mot comparator voi thu tu dao nguoc
        Comparator r = Collections.reverseOrder();
        // Sap xep list boi su dung comparator
        Collections.sort(ll, r);
        // Lay iterator
        Iterator li = ll.iterator();
        System.out.print("List duoc sap xep theo thu tu dao nguoc
la: ");
        while(li.hasNext()) {
            System.out.print(li.next() + " ");
        }
        System.out.println();
        Collections.shuffle(ll);
        // Hien thi danh sach sap ngau nhien
        li = ll.iterator();
        System.out.print("List sau khi bi xao tron la: ");
        while(li.hasNext()) {
            System.out.print(li.next() + " ");
        }
        System.out.println();
        System.out.println("Minimum: " + Collections.min(ll));
        System.out.println("Maximum: " + Collections.max(ll));
    }
}

```

Kết quả:

```
List duoc sap xep theo thu tu dao nguoc la: 20 8 -8 -20
List sau khi bi xao tron la: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

f) Sử dụng đối tượng Iterator với các collection

Thông thường trong nhiều trường hợp người sử dụng muốn duyệt tuần hoàn qua các phần tử trong một tập hợp. Ví dụ duyệt danh sách, hiển thị các phần tử trong danh sách. Cách đơn giản nhất để thực hiện điều này là sử dụng một *Iterator*, là một đối tượng mà triển khai *ListIterator interface*.

Iterator cho khả năng tuần hoàn qua một tập hợp, tìm kiếm và xóa bỏ các phần tử. *ListIterator* kế thừa *Iterator* để cho phép tìm theo song hướng một danh sách và sửa đổi các phần tử. Sử dụng *Iterator* để tuần hoàn các phần tử trong một *collection* như sau:

- Đặt *Iterator* ở đầu *collection* : gọi phương thức *iterator()* của *collection*
- Thiết lập vòng lặp gọi tới *hasNext()* cho tới khi *hashNext()* trả về *true*
- Trong vòng lặp, thu được mỗi phần tử bởi gọi *next()*

Với các *collection* triển khai *List*, thường gọi *ListIterator*. Một số phương thức tiêu biểu của *Iterator* (Bảng 5.7).

STT	Phương thức và Miêu tả
1	boolean hasNext() Trả về true nếu có nhiều phần tử. Nếu không là false
2	Object next() Trả về phần tử kế tiếp. Ném NoSuchElementException nếu không có một phần tử kế tiếp
3	void remove()

	Gỡ bỏ phần tử hiện tại. Ném IllegalStateException nếu cố gắng gọi phương thức remove() mà không được đặt trước một triệu hồi tới next()
--	---

Bảng 5.7. Một số phương thức của Iterator.

Phương thức được khai báo bởi ListIterator trong Java (Bảng 5.8).

STT	Phương thức và Miêu tả
1	void add(Object obj) Chèn obj vào trong List ở trước phần tử mà sẽ được trả về bởi lần triệu hồi tiếp theo tới next()
2	boolean hasNext() Trả về true nếu có một phần tử kế tiếp. Nếu không là false
3	boolean hasPrevious() Trả về true nếu có một phần tử ở trước. Nếu không là false
4	Object next() Trả về phần tử kế tiếp. Ném NoSuchElementException nếu không có phần tử đó
5	int nextIndex() Trả về chỉ mục của phần tử kế tiếp. Nếu không có phần tử này, trả về kích cỡ của list
6	Object previous() Trả về phần tử trước. Ném NoSuchElementException nếu không có phần tử đó
7	int previousIndex() Trả về chỉ mục của phần tử ở trước. Nếu không có phần tử này, trả về -1
8	void remove() Gỡ bỏ phần tử hiện tại từ list. Ném IllegalStateException nếu remove() được triệu hồi trước khi next() hoặc previous() được gọi

9	void set(Object obj) Gán obj tới phần tử hiện tại. Đây là phần tử cuối cùng được trả về bởi một triệu hồi tới next() hoặc previous()
---	--

Bảng 5.8. Một số phương thức của ListIterator.

Sau đây là ví dụ minh họa cả *Iterator* và *ListIterator*. Nó sử dụng một đối tượng *ArrayList*, nhưng các qui tắc chung áp dụng tới bất kỳ kiểu *collection* nào.

Ví dụ 5.18.

```
Import java.util.*;
public class Demo2 {
    public static void main(String []args) {

        CanBo[] lscb;
        int n;
        Scanner sc = new Scanner(System.in);
        System.out.println("Nhập số cán bộ: ");
        n = Integer.valueOf(sc.nextLine());
        lscb = new CanBo[n];
        try{

            // Khởi tạo từng phần tử Cán bộ trong danh sách
            for(int i =0; i<n; i++){
                lscb[i] = new CanBo();
                System.out.println("Nhập từng cán bộ: ");
                for(int i =0; i<n; i++){
                    String tmp = new String();
                    System.out.println("Nhập họ tên: ");
                    lscb[i].nhapTen(sc.nextLine());
                    System.out.println("Nhập ngày sinh: ");
                    lscb[i].nhapNgaysinh(sc.nextLine());
                    System.out.println("Nhập giới tính: ");
                    lscb[i].nhapGioitinh(sc.nextLine());
                    System.out.println("Nhập quê quán: ");
                    lscb[i].nhapQuequan(sc.nextLine());
                    System.out.println("Nhập cấp hàm: ");
                    lscb[i].setCapHam(sc.nextLine());
                    System.out.println("Nhập chức vụ: ");
                    lscb[i].setChucVu(sc.nextLine());
                    System.out.println("Nhập lương: ");
                    lscb[i].setLuong(Double.valueOf(sc.nextLine()));
                }
            }

        } catch (ArrayIndexOutOfBoundsException | 
        NullPointerException | NumberFormatException | ArithmeticException e) {
            System.out.println("Có lỗi xảy ra:" + e);
        }
    }
}
```

```

        finally{
            System.out.println("Chương trình đã chạy xong");
        }

        ArrayList ls = new ArrayList();
        for(int i =0; i<n; i++)
            ls.add(lscb[i]);
        System.out.println("Sử dụng Iterator để In ra
ArrayList:");
        Iterator it = ls.iterator();
        while(it.hasNext()){
            CanBo temp = (CanBo) it.next();
            temp.inThongtin();
        }
        ListIterator lt = ls.listIterator();
        while(lt.hasNext()){
            CanBo temp =(CanBo) lt.next();
            temp.inThongtin();
        }
    }
}

```

Tóm lại, *Iterator* cho phép duyệt một *collection*, trong khi duyệt ta vừa thêm/sửa/xóa được *collection* đó, chiếm ưu thế hơn *for*.

II. LẬP TRÌNH TỔNG QUÁT

1. Giới thiệu về lập trình tổng quát

a) Khái niệm lập trình tổng quát

Lập trình tổng quát có tên thuật ngữ là lập trình *generics*. Với *generics* có nghĩa *tham số hóa kiểu dữ liệu*. Tham số hóa kiểu dữ liệu cho phép tạo ra các lớp, giao diện và các phương thức với nhiều kiểu dữ liệu khác nhau.

Một lớp, giao diện hay một phương thức mà thực hiện trên một kiểu tham số xác định thì gọi là *generic*. Như vậy khái niệm tổng quát – *generics* có nghĩa là *cách thức lập trình tổng quát cho phép một đối tượng hoạt động với nhiều kiểu dữ liệu khác nhau*.

Để dễ hình dung, lấy ví dụ minh họa việc sử dụng *ArrayList* với các kiểu dữ liệu khác nhau trong ví dụ dưới đây

Ví dụ 5.19. Ví dụ về cách sử dụng lớp *ArrayList* cho phép thêm nhiều loại đối tượng.

```
package generic;

import java.util.ArrayList;

public class SampleGeneric1 {

    public static void main(String[] args) {
        // Sử dụng ArrayList với các kiểu dữ liệu khác nhau
        ArrayList mylist = new ArrayList();

        // Thêm vào array
        mylist.add(10);
        mylist.add("Hello");
        mylist.add(true);
        mylist.add(15.75);
        // Lấy ra
        int i = (Integer)mylist.get(0);
        String s = (String)mylist.get(1);
        boolean b = (boolean)mylist.get(2);
        double d = (double)mylist.get(3);

        // Hiển thị
        System.out.println("Phan tu thu nhat la: " + i);
        System.out.println("Phan tu thu hai la : " + s);
        System.out.println("Phan tu thu ba la : " + b);
        System.out.println("Phan tu thu tu la : " + d);
    }
}
```

Biến *mylist* là đối tượng thuộc lớp *ArrayList*. Biến này làm việc được với cả số nguyên *mylist.add(10)*, xâu kí tự *mylist.add("Hello")*, giá trị logic *mylist.add(true)* và số thực *mylist.add(15.75)*. Chính nhờ khả năng *generic* của lớp *ArrayList* cho phép *mylist* thêm được nhiều kiểu đối tượng vào danh sách của *mylist*.

Ví dụ tiếp theo cho thấy việc sử dụng *ArrayList* chỉ với các dữ liệu kiểu *Integer*

Ví dụ 5.20.

```
package generic;

import java.util.ArrayList;

public class SampleGeneric2 {
```

```

public static void main(String[] args) {
    // Sử dụng ArrayList với các kiểu dữ liệu Integer
    ArrayList<Integer> mylist = new ArrayList<Integer>();

    // Thêm vào array
    mylist.add(10);
    mylist.add("Hello"); //Error

    // Lấy ra
    int i = mylist.get(0);

    // Hiển thị
    System.out.println("So nguyen: " + i);
}
}

```

Trong ví dụ này, `ArrayList<Integer> mylist = new ArrayList<Integer>()` đã xác định kiểu tham số là `Integer`, *generic* ở đây là lớp `Integer`. Nên `mylist` sẽ chỉ làm việc với các biến kiểu `Integer`. Giả sử nếu có một khai báo khác `ArrayList<String> mylist2 = new ArrayList<String>()` thì lúc đó `mylist2` cũng sẽ chỉ làm việc với các biến kiểu `String`, *generic* được xác định ở đây là `String`. Nhờ có khả năng tham số hóa kiểu dữ liệu nên `ArrayList` đã rất linh hoạt trong việc tạo ra các danh sách phù hợp với nhiều kiểu dữ liệu khác nhau. Ví dụ 5.19 và 5.20 minh họa việc sử dụng *collection* với một *generic*.

b) Lợi ích của lập trình tổng quát

Lợi ích đầu tiên của lập trình tổng quát chính là kiểm tra kiểu dữ liệu trong thời điểm biên dịch. Trình biên dịch của Java áp dụng kiểm tra đoạn mã chỉ ra kiểu dữ liệu xác định – đoạn mã *generic* để phát hiện các vấn đề như vi phạm an toàn kiểu dữ liệu. Việc sửa lỗi tại thời gian biên dịch dễ dàng hơn nhiều khi sửa lỗi tại thời điểm chạy chương trình.

Ví dụ 5.21. Đoạn mã dưới đây, khi không dùng *generic* phải ép kiểu:

```

ArrayList ls = new ArrayList();
for(int i =0; i<n; i++)
    ls.add(lsrb[i]);
System.out.println("In ra ArrayList:");
for(int i=0; i< ls.size(); i++)
{
    CanBo cb = (CanBo)ls.get(i);
}

```

```
        cb.inThongtin();
    }
```

Khi chỉ ra *generic*, việc ép kiểu đã được loại bỏ:

```
ArrayList<CanBo> ls = new ArrayList<CanBo>();
for(int i =0; i<n; i++)
    ls.add(lscb[i]);
System.out.println("In ra ArrayList:");
for(int i=0; i< ls.size(); i++)
{
    CanBo cb = ls.get(i);
    cb.inThongtin();
}
System.out.println("Sử dụng Iterator để In ra ArrayList:")
Iterator<CanBo> it = ls.iterator();
while(it.hasNext()){
    CanBo temp = it.next();
    temp.inThongtin();
}
ListIterator<CanBo> lt = ls.listIterator();
while(lt.hasNext()){
    CanBo temp =lt.next();
    temp.inThongtin();}
```

Ngược lại, trong nhiều trường hợp, Java cho phép lập trình viên thực hiện các xử lý tổng quát. Bằng cách sử dụng *generics*, người lập trình có thể thực hiện các thuật toán tổng quát với các kiểu dữ liệu tùy chọn khác nhau như ví dụ nêu trên.

2. Lớp tổng quát

a) *Tổng quát hóa một lớp*

Khai báo một lớp tổng quát giống như khai báo một lớp thông thường nhưng theo sau tên lớp là một kiểu dữ liệu tổng quát.

Trong trường hợp có nhiều tham số thì mỗi tham số được phân cách bởi dấu phẩy. Những lớp này còn được biết như là các lớp tham số hóa hoặc các kiểu tham số hóa bởi vì chúng chấp nhận một hoặc nhiều tham số.

Ví dụ 5.22. Định nghĩa một lớp kiểu *generic*

```
Public class Box<T> {
    private T t ;
    public void add(T t) {
```

```

        this.t = t;
    }

    public T get() {
        return t;
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        Box<String> stringBox = new Box<String>();

        integerBox.add(new Integer(10));
        stringBox.add(new String("Hello World"));

        System.out.printf("Gia tri integer la :%d\n\n",
integerBox.get());
        System.out.printf("Gia tri string la :%s\n",
stringBox.get());
    }
}

```

Kết quả :

```

Gia tri integer la :10
Gia tri string la :Hello World

```

Lớp tổng quát với 1 tham số kiểu

Ví dụ 5.23. Tạo một cặp tham số kiểu với 2 số cùng kiểu

```

public class Pair<T>
{
    public Pair(){ f
        first = null; second = null;
    }
    public Pair(T first, T second){
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public T getSecond() {
        return second;
    }
    public void setFirst(T newValue) {
        first = newValue;
    }
    public void setSecond(T newValue) {
        second = newValue;
    }
    private T first;
    private T second;
}

```

```

...// Khi sử dụng
Pair<String> mm = new Pair<String> ("1st", "2nd");
System.out.println(mm.getFirst() + "," + mm.getSecond());

```

Lớp tổng quát với 2 tham số kiểu

Ví dụ 5.24. Tạo một cặp 2 số với 2 kiểu khác nhau

```

public class Pair<T, U> {
    public Pair() {
        first = null; second = null;
    }
    public Pair(T first, U second) {
        this.first = first; this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public U getSecond() {
        return second;
    }
    public void setFirst(T newValue) {
        first = newValue;
    }
    public void setSecond(U newValue) {
        second = newValue;
    }
    private T first;
    private U second;
}
...// Sử dụng
Pair<String, Integer> mm = new Pair<String, Integer> ("1st", 1);
System.out.println(mm.getFirst() + "," + mm.getSecond());
System.out.println(mm.getFirst() + "," + mm.getSecond());

```

b) Các kiểu dữ liệu chưa xác định – Raw types

Với ví dụ 5.21 xác định Box với kiểu dữ liệu Integer và Double. Nhưng cũng có thể tạo một biến kiểu Box như sau:

```

Box x = new Box();
x.add(new Integer(20));
System.out.println("Giá trị Box chứa":+x.get());

```

Kết quả in ra là 20.

Trong trường hợp này, x được nói rằng đã có một kiểu dữ liệu chưa xác định, trình biên dịch hoàn toàn sử dụng kiểu *Object* xuyên suốt trong lớp tổng quát

generic. Đây là một điểm quan trọng để các bản cũ tương thích với các bản mới của Java. Ví dụ, trong *Java Collection*, mọi cấu trúc dữ liệu đều tham chiếu tới các kiểu dữ liệu *Object*, nhưng các phiên bản sau được cài đặt như là các kiểu *generic*.

Một kiểu dữ liệu chưa xác định *Box* có thể gán một *Box* khác đã xác định tham số kiểu, như `Box<Integer>` như sau

```
Box x = new Box<Integer>();
```

Bởi vì kiểu *Integer* là lớp con của *Object*. Phép gán trên hợp lệ bởi vì các thành phần của `Box<Integer>` có kiểu *Integer*, cũng sẽ có kiểu *Object*. Tương tự như vậy, một biến *Box* xác định được tham số kiểu cũng có thể gán với một *Box* chưa xác định kiểu :

```
Box< Integer > y = new Box();
```

Lưu ý: Mặc dù phép gán này hợp lệ, nhưng nó lại không an toàn bởi vì *Box* của kiểu dữ liệu không xác định có thể chứa các kiểu dữ liệu khác ngoài *Integer*. Trong trường hợp này, trình biên dịch sẽ đưa ra một thông điệp cảnh báo – *warning* để chỉ ra sự không an toàn của phép gán này.

3. Phương thức tổng quát

Phương thức tổng quát là phương thức mà có thể được gọi với nhiều kiểu dữ liệu khác nhau. Dựa vào kiểu tham số truyền vào, trình biên dịch sẽ xử lý mỗi lời gọi phương thức sao cho phù hợp.

Cú pháp chung khi tạo phương thức tổng quát như sau :

```
<quyền truy cập> <tên tham số kiểu> <kiểu phương thức> <tên  
phương thức> [<danh sách tham số>] {....}
```

Tên tham số kiểu đặt theo quy ước như sau

Ký tự	Ý nghĩa
E	Element – phần tử

K	Key – khóa
V	Value – giá trị
T	Type – kiểu dữ liệu
N	Number – số

Bảng 5.1. Kí hiệu tổng quát và ý nghĩa.

Ví dụ dưới đây minh họa cách định nghĩa một phương thức tổng quát và cách sử dụng phương thức này

Ví dụ 5.25. Tham số kiểu ở đây là T

```
package generic;
public class GenericMethodTest {
    // generic method printArray
    public <T> void printArray(T[] inputArray) {
        // In ra cac bien trong mang ra man hinh
        for ( T element : inputArray ) {
            System.out.print(element);
        }
        System.out.println();
    }
    public static void main( String args[] ){
        // Tao cac mang Integer, Double va Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Mang intArray bao gom:" );
        printArray( intArray ); // truyen mot mang Integer

        System.out.println( "\nMang doubleArray bao gom:" );
        printArray( doubleArray ); // truyen mot mang Double

        System.out.println( "\nMang charArray bao gom:" );
        printArray( charArray ); // truyen mot mang Character
    }
}
```

Kết quả:

```
Mang intArray bao gom:
1 2 3 4 5 6

Mang doubleArray bao gom:
2.2 3.3 4.4
Mang charArray bao gom:
H E L L O
```

4. Các ký tự đại diện generic – wildcards

Trong phần này, chúng tôi giới thiệu một khái niệm tổng quát rất mạnh là *wildcard – ký tự đại diện*. Trước khi giới thiệu về ký tự đại diện, chúng ta cùng tìm hiểu một ví dụ dưới đây.

Giả sử có một phương thức tổng quát *sum* tính tổng các số trong một *collection ArrayList*. Ta cần thêm một số nguyên vào trong *collection* này. Bởi vì các lớp tổng quát *generics* có thể dùng với lớp hoặc *interface*, trong Java các số nguyên có thể được đóng gói tự động trong các lớp đối tượng. Ví dụ, số nguyên có kiểu *int*, *int* cũng có thể được đóng gói thành đối tượng *Integer*. Số thực *double* thành đối tượng *Double*. Nhưng chúng ta lại muốn tính tổng của tất cả các số trong *collection* bất kể kiểu của số là gì. Vì lý do này, chúng ta định nghĩa *collection ArrayList* với kiểu tham số là *Number*, đây là lớp cha của cả lớp *Integer* và *Double*.Thêm vào đó, phương thức *sum* sẽ nhận các tham số của kiểu *ArrayList<Number>* và tổng của chúng. Chương trình để tính tổng các *Number* trong *ArrayList* có thể cài đặt như sau:

Ví dụ 5.26.

```
// Summing the elements of an ArrayList.
Import java.util.ArrayList;
public class TotalNumbers {
    public static void main( String args[] ) {
        /* create, initialize and output ArrayList of Numbers
containing both Integers and Doubles, then display total of the
elements */
        Number[] numbers = { 1, 2.4, 3, 4.1 };
        ArrayList<Number> numberList=new ArrayList<Number>();

        for ( Number element:numbers ) numberList.add( element );
        System.out.printf("numberList contains: %s\n",
numberList );
        System.out.printf("Total of the elements in numberList:
%.1f\n",sum(numberList));
    } // end main
    // calculate total of ArrayList elements
    public static double sum( ArrayList< Number > list ) {
        double total = 0; // initialize total
        // calculate sum
        for (Number element:list ){
```

```

        total += element.doubleValue();
        return total;
    } // end method sum
} // end class TotalNumbers

```

Kết quả là một dãy với nhiều kiểu số:

```

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5

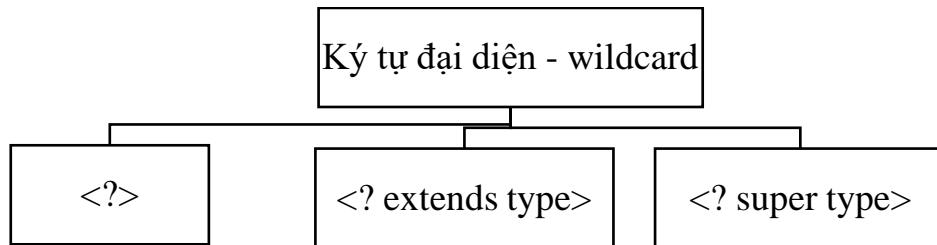
```

numbers là một mảng các số nguyên và số thực. Số nguyên 1 kiểu *int* đã tự động đóng gói thành kiểu *Integer*, tương tự kiểu *double* đã tự động đóng kiểu thành kiểu *Double*. *numberList* sẽ được gán các số trong *Numbers*. Trong phương thức *sum*, nhận các *Number* trong *ArrayList* để tính tổng trong danh sách này. Phương thức này nhận các giá trị *double* để tính và trả về số thực *double*. Biến *total* được khởi tạo giá trị 0. Tiếp đến câu lệnh *for* làm nhiệm vụ như sau: biến *element* sẽ duyệt danh sách từ đầu đến cuối, mỗi lần duyệt sẽ lấy ra giá trị của *element* và chuyển thành *double* *total+=element.doubleValue()*. Vậy với ví dụ trên *sum* đã tính được tổng các số kể cả số nguyên và số thực. Giả sử chúng ta mong muốn phương thức *sum* này cũng có thể làm việc với một *ArrayList* chứa các thành phần chỉ chứa một loại dữ liệu kiểu số, giả sử là *ArrayList<Integer>*. Khi đó chúng ta sửa lớp *TotalNumbers* để tạo ra một *ArrayList* của các *Integer* để đưa các số vào phương thức *sum*. Khi đó, biên dịch sẽ báo lỗi: *sum(java.util.ArrayList<java.lang.Number>) in TotalNumbersErrors cannot be applied to (java.util.ArrayList<java.lang.Integer>)*.

Mặc dù *Number* là lớp cha của *Integer*, trình biên dịch vẫn không thể coi kiểu của *ArrayList<Number>* thành kiểu *ArrayList<Integer>*. Giả sử có thể chuyển kiểu được trong trường hợp này, muốn thêm một số *Double* vào trong *ArrayList<Integer>* này lại không khả thi. Từ đó, người ta mới nghĩ đến tạo ra một phương thức *sum* linh hoạt hơn, có thể tính tổng của tất cả các thành phần của mọi kiểu *ArrayList* là kiểu con của *Number*? Do đó kiểu tham số *ký tự đại diện – wildcard* đã được đề xuất.

Kí tự đại diện cho phép người lập trình xác định tham số đầu vào, giá trị trả về, các biến hoặc các trường... có thể hoạt động giống như *siêu kiểu* của các kiểu đã được tham số hóa.

Các kí tự đại diện *wildcard* trong Java có thể chia thành các loại như sau:



Hình 5.5. Các kí tự đại diện.

Chúng ta sẽ đi tìm hiểu lần lượt từng kiểu kí tự đại diện.

a) Kí tự đại diện <?>

Xét ví dụ sau:

Ví dụ 5.27.

```
private boolean checkEquals(RestrictExample<T> e) {  
    if(number.doubleValue() == e.number.doubleValue()) {  
        return true;  
    }  
    return false;  
}  
private boolean checkEquals2(RestrictExample<?> e) {  
    if(number.doubleValue() == e.number.doubleValue()) {  
        return true;  
    }  
    return false;  
}
```

Trong ví dụ trên khi sử dụng, phương thức *checkEquals* chỉ chấp nhận tham số e có kiểu dữ liệu giống với biến *number*. Phương thức *checkEqual2* chấp nhận tham số e có kiểu dữ liệu khác với biến *number*.

b) Kí tự đại diện <? extends type>

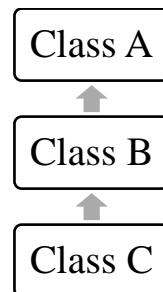
Với kí tự đại diện mở rộng <? extends type> chấp nhận bất kỳ một đối tượng nào có kiểu *type* hoặc thừa kế từ *type*. Ví dụ:

```

public void processElement(List<? extends A> elements) {
    ...
}

```

Trong đó:



Hình 5.6. Thừa kế trên nhiều lớp.

Sử dụng phương thức *processElement* sẽ hợp lệ khi các đối tượng thuộc lớp A hoặc thừa kế từ lớp A (B hoặc C). Có nghĩa là các câu lệnh dưới đây hợp lệ:

```

List<A> listA = new ArrayList<A>();
processElement(listA);

List<B> listB = new ArrayList<B>();
processElement(listB);

List<C> listC = new ArrayList<C>();
processElement(listC);

```

c) Kí tự đại diện $<? super type>$

Với kí tự đại diện kiểu này, khi sử dụng, các đối tượng nào có kiểu *type* hoặc là cha của *type* sẽ được chấp nhận. Ví dụ:

```

public void processElement(List<? Super A> elements) {
    ...
}
List<A> listA = new ArrayList<A>();
processElement(listA);

List<Object> listO = new ArrayList<Object>();
processElement(listO);

```

Ứng dụng

5. Ứng dụng của lập trình tổng quát

Lập trình tổng quát có thể ứng dụng trong một số trường hợp như sau:

- Giới hạn cho các tham số kiểu

Có một số trường hợp chúng ta muốn hạn chế kiểu dữ liệu của các tham số. Chẳng hạn tạo phương thức chỉ chấp nhận tham số với kiểu dữ liệu là số nguyên hoặc số thực.

Ví dụ 5.28.

```
package generic;

public class RestrictExample <T extends Number> {
    private T number;

    public RestrictExample(T number) {
        this.number = number;
    }

    public double reciprocal() {
        return 1/number.doubleValue();
    }

    public static void main(String[] args) {
        RestrictExample<Integer> n1 = new
RestrictExample<Integer>(5);
        System.out.println("Reciprocal: " + n1.reciprocal());
        RestrictExample<Double> n2 = new
RestrictExample<Double>(7.5);
        System.out.println("Reciprocal: " + n2.reciprocal());
        //error
        /*RestrictExample<String> s1 = new
RestrictExample<String>("Hello"); */
    }
}
```

- Xây dựng một phương thức xử lý với nhiều kiểu dữ liệu khác nhau.

Ví dụ 5.29. Tạo lớp tên *MaximumTest* trong package *generic*

```
package generic;
public class MaximumTest {
    // determines the largest of three Comparable objects
    public <T extends Comparable<T>> T maximum(T x, T y, T z) {
        T max = x; // assume x is initially the largest
        if (y.compareTo(max) > 0) {
            max = y; // y is the largest so far
        }
        if (z.compareTo(max) > 0) {
            max = z; // z is the largest now
        }
        return max; // returns the largest object
    }
    public static void main(String args[]) {
        // Create object
        MaximumTest mt = new MaximumTest();
```

```

        System.out.println("Max of 3, 4, 5 is " + mt.maximum(3,
4, 5));
        System.out.println("Maxm of 6.6, 8.8, 7.7 is " +
mt.maximum(6.6, 8.8, 7.7));
        System.out.println("Max of pear, apple, orange is " +
mt.maximum("pear", "apple", "orange"));
    }
}

```

Kết quả khi sau khi chạy là:

```

Max of 3, 4, 5 is 5
Max of 6.6, 8.8, 7.7 is 8.8
Max of pear, apple, orange is pear

```

6. Hạn chế của lập trình tổng quát

Lập trình tổng quát trong Java còn có một số hạn chế như sau:

- Không thể khởi tạo *generic* với dữ liệu kiểu nguyên thủy

```

Pair<int, char> p = new Pair<>(8, 'a'); //error
Pair<Integer, Character> p = new Pair<>(8, 'a');

```

- Không thể tạo thẻ hiện – *instance* cho kiểu dữ liệu

```

class Gen<T>{
    T obj;
    Gen() {
        obj= new T(); //Illegal (error)
    }
}

```

- Không thể là *static* trong *class*

```

class Gen<T>{
    static T obj; //Kiểu T không thể là static
    static T getObj() { //Phương thức không thể static
        return obj;
    }
}

```

- Không thể tạo mảng

```

Gen<Integer> gens[] = new Gen<Integer>[10]; //error

Gen<?> gens[] = new Gen<?>[10]; //ok
Gens[0] = new Gen<Integer>(25);
Gens[1] = new Gen<String>("Hello");

```

- Cuối cùng, với ngoại lệ *exception*, không thể tạo lớp ngoại lệ là *generic*.

Câu hỏi và bài tập cuối chương

Câu 1. Tạo class tên MyArrayList và thực hiện các công việc sau:

- Thêm vào ArrayList một số nguyên
- Thêm vào ArrayList một số thực
- Thêm vào ArrayList một giá trị boolean
- Thêm vào ArrayList một chuỗi
- In ra màn hình 4 giá trị trên

Câu 2. Tạo class tên MyGenericArrayList và thực hiện các công việc sau:

- Tham số hoá dữ liệu cho ArrayList là Integer.
- Sử dụng vòng lặp để nhập các số từ 1 đến 10 vào ArrayList.
- In ra màn hình các giá trị trong ArrayList

Câu 3. Tạo class tên Student có các thuộc tính như id, name, age. Viết các phương thức constructor, setter, getter, toString.

Câu 4. Tạo class tên Employee có các thuộc tính như id, name, salary. Viết các phương thức constructor, setter, getter, toString.

Câu 5. Tạo class PersonModel và thực hiện các công việc sau:

```
package generic;

import java.util.ArrayList;

/**
 *
 * @author giasutinhoc.vn
 */
public class PersonModel <T> {
    private ArrayList<T> al = new ArrayList<T>();
    public void add(T obj) {
        al.add(obj);
    }
    public void display() {
        for(T o : al){
            System.out.println(o);
        }
    }
}
```

```
public static void main(String[] args) {  
    //Viết xử lý cho phương thức main  
}  
}
```

Đoạn code cần viết thêm vào phương thức main thực hiện các công việc sau:

- Tạo đối tượng PersonModel<Student>
- Gọi phương thức add để nhập vào 2 sinh viên
- Gọi phương thức display để hiển thị thông tin của 2 sinh viên vừa nhập
- Tạo đối tượng PersonModel<Employee>
- Gọi phương thức add để nhập vào 2 nhân viên
- Gọi phương thức display để hiển thị thông tin của 2 nhân viên vừa nhập
- Tạo đối tượng PersonModel<String>
- Gọi phương thức add để nhập vào họ tên của 2 người.
- Gọi phương thức display để hiển thị họ tên vừa nhập.

Chương 6. GIỚI THIỆU LẬP TRÌNH GIAO DIỆN VÀ MẪU THIẾT KẾ

I. APPLET VÀ LẬP TRÌNH GIAO DIỆN GUI

1. Applet

a) Giới thiệu về Applet

Applet là kiểu chương trình Java đặc biệt, được nhúng vào trang Web và được chạy bởi trình duyệt. Khi người dùng quan sát một trang HTML có chứa một *applet* thì đoạn mã của *applet* này được tải tới thiết bị của người sử dụng. Mọi chương trình *applet* đều là các lớp con của lớp *Applet*.

Lớp *Applet* nằm trong gói *java.applet* và kế thừa từ lớp *Panel* nên *Applet* cũng là một lớp khung chứa. Lớp này chứa các phương thức thực thi một *applet*; không cần hàm *main()*. Để tạo một *applet*, ta cần khai báo sử dụng hai gói sau:

```
import java.applet;
import java.awt.*;
```

Để thực thi một applet, chúng ta có hai cách:

Cách 1: sử dụng Netbean

Với cách sử dụng này chúng ta không cần tạo file html và xem dưới chế độ AppletViewer bằng cách biên dịch file như thông thường

Cách 2: sử dụng trình duyệt Web

Tạo một html với đoạn mã :

```
<applet codebase="classes" code="FirstApplet.class" width=350 height=200></applet>
```

Sau đó sao chép lớp applet và file html vào thư mục gốc Root của Webserver đã được cài đặt trong thư mục Netbean.

- Cấu trúc của một *applet*

Có 4 sự kiện xảy ra với một *applet* trong suốt quá trình tồn tại và thi hành của nó. Cấu trúc của một *applet* được định rõ từ bốn sự kiện này. Với mỗi sự kiện, một phương thức tương ứng được gọi thực hiện tự động.

Bốn sự kiện xảy ra với *applet* trong suốt quá trình tồn tại của nó là:

Khởi tạo (initialize)
Khởi động (start)
Dừng (stop)
Hủy (destroy)

Bảng 6.1 liệt kê các phương thức của một *applet*:

Phương thức	Vai trò
init()	Được gọi khi applet khởi tạo (ví dụ khi người dùng bắt đầu mở một trang Web có chứa applet). Trong quá trình khởi tạo, các đối tượng mà applet cần được tạo ra. Phương thức này được dùng để tải ra môi trường đồ họa, khởi tạo các biến, và tạo các đối tượng.
start()	Được gọi khi applet được khởi động. Ngay khi quá trình khởi tạo kết thúc, applet được khởi động và chạy. Phương thức này cũng được dùng để khởi động lại applet sau khi nó bị dừng.
stop()	Được gọi khi sự thực hiện của applet bị tạm dừng (ví dụ khi người dùng mở trang HTML khác hay khi thu nhỏ cửa sổ trình duyệt về thanh taskbar).
destroy()	Được gọi khi người dùng kết thúc sự thực hiện của applet (khi đóng trình duyệt hay tắt máy). Phương thức này giải phóng tài nguyên hệ thống cung cấp để chạy applet.

Bảng 6.1. Bảng các phương thức trong vòng đời của Applet

Ngoài 4 phương thức cơ bản trên, *applet* còn có thêm 1 số phương thức khác gồm:

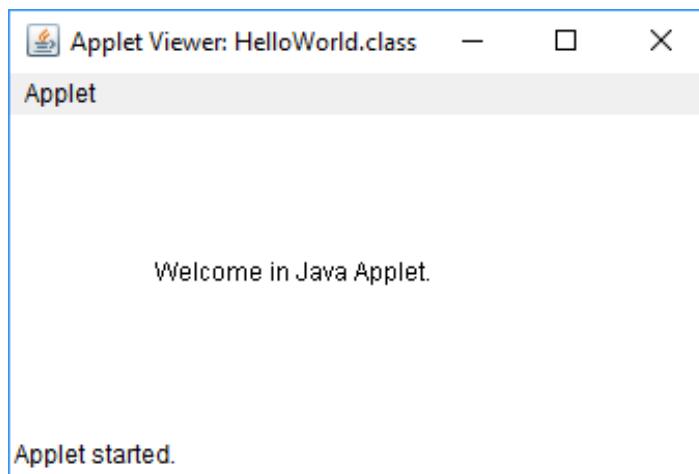
+ Phương thức *paint()* dùng để hiển thị kí tự, các hình vẽ hay ảnh trên *applet*. Tham số truyền cho phương thức *paint()* là một đối tượng *Graphics*. Lớp *Graphics* nằm trong gói *java.awt*.

- + Phương thức *repaint()* được dùng khi khung nhìn của *applet* cần được tự động cập nhật lại.
- + Phương thức *showStatus()* hiển thị thông tin lên thanh trạng thái (*status bar*) của trình duyệt.
- + Phương thức *getAppletInfo()* trả về thông tin của *applet*. Nó trả về một đối tượng *String*. Ta có thể viết đè phương thức này để xác định thông tin cho *applet*.

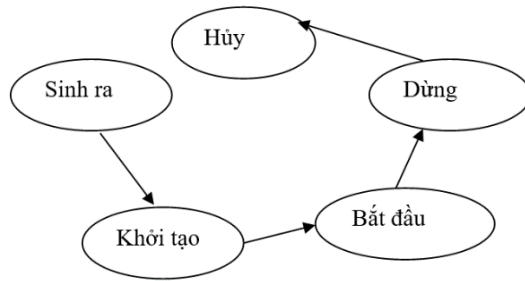
Ví dụ 6.1: Ví dụ minh họa các phương thức trong lớp *applet*

```
// tên file HelloWorld
import java.applet.*;
import java.awt.*;
public class HelloWorld extends Applet{
    @Override
    public void paint(Graphics g){
        g.drawString("Welcome in Java Applet.",70,80);
    }
}
```

Kết quả chạy chương trình bằng *AppletViewer* như Hình 6.1.



Hình 6.1. Chạy applet với *AppletViewer*.



Hình 6.2. Vòng đời của applet.

Ví dụ 6.2: Minh họa các sự kiện diễn ra trong vòng đời của *applet*:

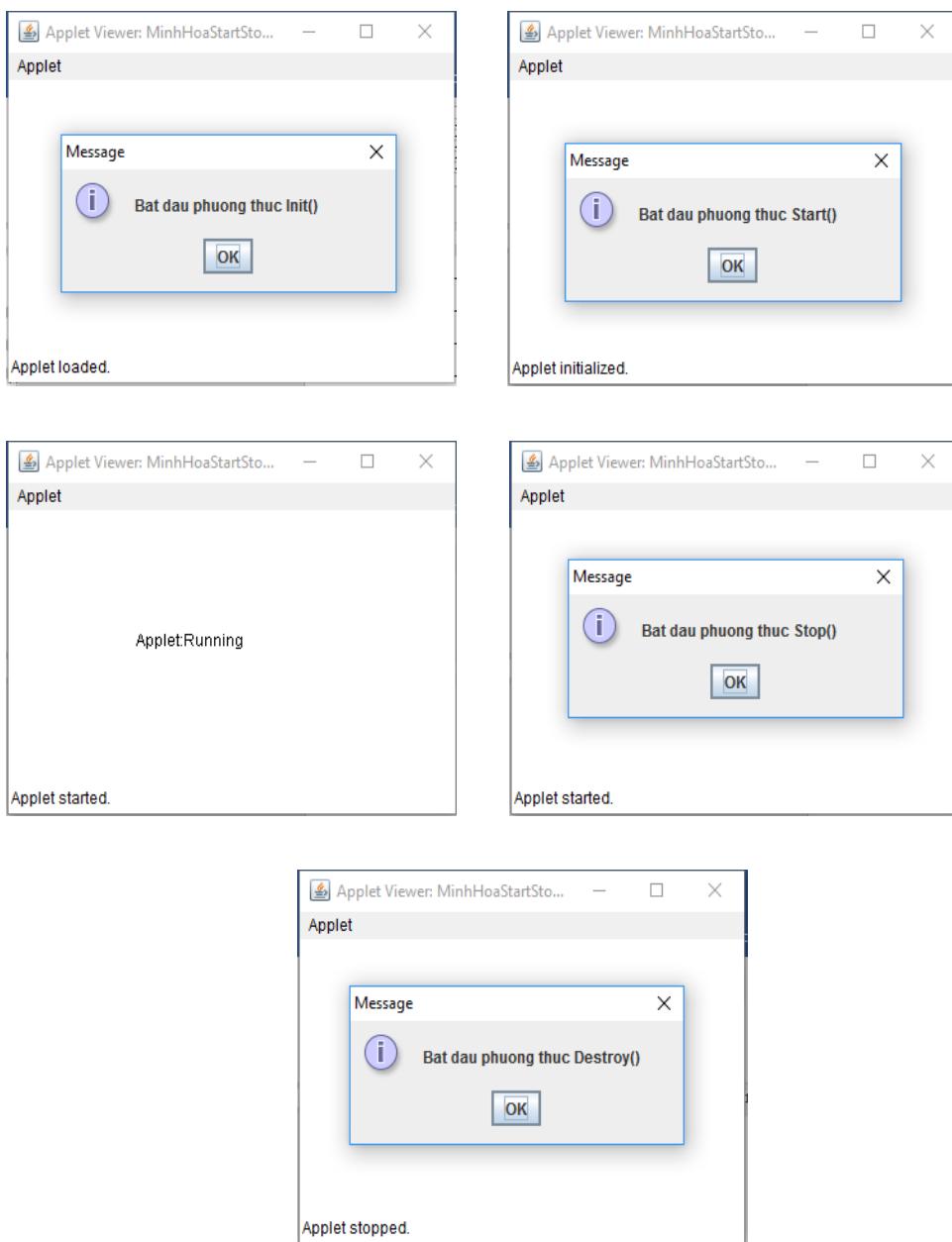
```

import java.applet.Applet;
import java.awt.Graphics;
import javax.swing.JOptionPane;
public class MinhHoaStartStopInit extends Applet{
    @Override
    public void init(){
        JOptionPane.showMessageDialog(null, "Bat dau phuong thuc
Init()");
    }
    @Override
    public void start(){
        JOptionPane.showMessageDialog(null, "Bat dau phuong thuc
Start()");
    }
    @Override
    public void stop(){
        JOptionPane.showMessageDialog(null, "Bat dau phuong thuc
Stop()");
    }
    @Override
    public void paint(Graphics g){
        g.drawString("Applet:Running",100,100);
    }
    @Override
    public void destroy(){
        JOptionPane.showMessageDialog(null, "Bat dau phuong thuc
Destroy()");
    }
}

```

```
}
```

Kết quả chạy applet InitStartStop bằng *AppletViewer* như Hình 6.3.



Hình 6.3. Kết quả chạy InitStartStop.

b) Sử dụng applet

- Truyền tham số cho *applet*

Do nhu cầu sử dụng, người dùng nhiều khi muốn tùy biến sự thể hiện của *applet* theo yêu cầu của mình. Giống như khi truyền tham số cho một hàm, giá trị trả về của hàm sẽ phụ thuộc vào giá trị của tham số mà ta truyền. Với *applet* cũng

vậy, người dùng cũng muốn tùy biến màu sắc, kích cỡ của các đối tượng đồ họa được vẽ ra, nhãn của một nút bấm, tên tệp ảnh cần in ra... mà không cần phải sửa đổi mã nguồn chương trình. Java cung cấp cơ chế truyền tham số cho *applet* từ tệp siêu văn bản để thực hiện điều đó.

Để truyền tham số cho *applet*, ta chèn các thẻ `<PARAM>` vào thân khối thẻ `<Applet>` trong trang Web. Thẻ `<PARAM>` có hai thuộc tính: thuộc tính `NAME` dùng để định danh thẻ `PARAM` trong thân chương trình *applet*, thuộc tính `VALUE` chứa giá trị sẽ truyền cho *applet*.

Ví dụ 6.3: Truyền tham số cho applet có tên là *ParamApplet* bằng thẻ `<PARAM>` có thuộc tính `NAME` là "ButtonLabel" và thuộc tính `VALUE` là "Hien Thi"

```
<applet code = "ParamApplet" width = "300" height = "400">
<PARAM NAME = "ButtonLabel" VALUE = "Hien Thi">
</applet>
```

Để lấy được các giá trị tham số này, trong tệp mã nguồn *ParamApplet.java* ta dùng phương thức `getParameter()` với đối số là tên của thẻ `PARAM`. Khi đó giá trị trả về sẽ là giá trị tương ứng của thuộc tính `VALUE` trong thẻ `PARAM`. Tệp mã nguồn *ParamApplet.java* như sau:

```
//tên file: ParamApplet.java
import java.applet.*;
import java.awt.*;
public class ParamApplet extends Applet{
    Button b;
    public void init(){
        b=new Button();
        String str=getParameter("ButtonLabel");
        b.setLabel(str);
        add(b);
    }
}
```

Trong chương trình trên, dòng lệnh:

```
String str=getParameter("ButtonLabel");
```

sẽ lấy giá trị của thẻ `PARAM` có `NAME="ButtonLabel"` và gán cho đối tượng `str` có kiểu `String`.

Tiếp theo, dòng lệnh: `b.setLabel(str)` sẽ gán nhãn cho đối tượng `Button` b là `str`. Như vậy nhãn của đối tượng b sẽ tùy thuộc vào tham số ta truyền cho `applet` bằng thẻ `PARAM`.

Tệp `ParamApplet.htm` nhúng `applet ParamApplet` trên như sau:

```
<applet code="ParamApplet" height=300 width=400>
    <PARAM NAME="ButtonLabel" VALUE="Hien Thi">
</applet>
```

Khi dùng trình duyệt mở file `ParamApplet.htm`, nút bấm trên `applet` sẽ có nhãn là giá trị ta truyền cho thuộc tính `VALUE` của thẻ `PARAM` có `NAME="ButtonLabel"`. Ở đây nhãn của nút bấm sẽ là "*Hien thi*".

Ta có thể truyền nhiều tham số cho `applet` từ tệp siêu văn bản, vì thế số lượng các thẻ `PARAM` được dùng là không hạn chế. Các thẻ `PARAM` được phân biệt nhau bởi thuộc tính `NAME`.

- Sự khác nhau giữa `applet` và các ứng dụng Java khác
 - + Các ứng dụng Java khác được chạy với trình thông dịch Java, trong khi applet chạy trên bất kì trình duyệt nào hỗ trợ Java, hoặc dùng trình appletviewer trong bộ JDK;
 - + Các ứng dụng Java khác chạy bắt đầu với phương thức `main()`, trong khi applet thì không;
 - + Điều quan trọng cần nhớ là một chương trình Java có thể vừa là applet vừa là một ứng dụng Java thông thường (application). Khi đó, chương trình Java này có thể chạy theo hai cách khác nhau.

Ví dụ 6.4: Chương trình `AWTApplet.java` sau vừa là một applet, tức là có thể được chạy nhúng trong trang HTML, lại vừa là một ứng dụng AWT thông thường, có thể chạy bằng trình thông dịch java của JDK:

```

//Tên file: AWTApplet.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AWTApplet extends Applet{
    Button b;
    public void init(){
        b=new Button("Nhập");
        add(b);
    }
    public void paint(Graphics g){
        g.drawString("Chương trình vừa là applet vừa là AWT", 50, 50);
    }
    public static void main(String arg[]){
        Frame f=new Frame("Chương trình demo");
        f.addWindowListener(new WindowAdapter());
    }

    public void windowClosing(WindowEvent e){
        System.exit(0);
        AWTApplet bsa=new AWTApplet();
        f.add(bsa);
        f.setSize(300,400);
        f.setVisible(true);
    }
}

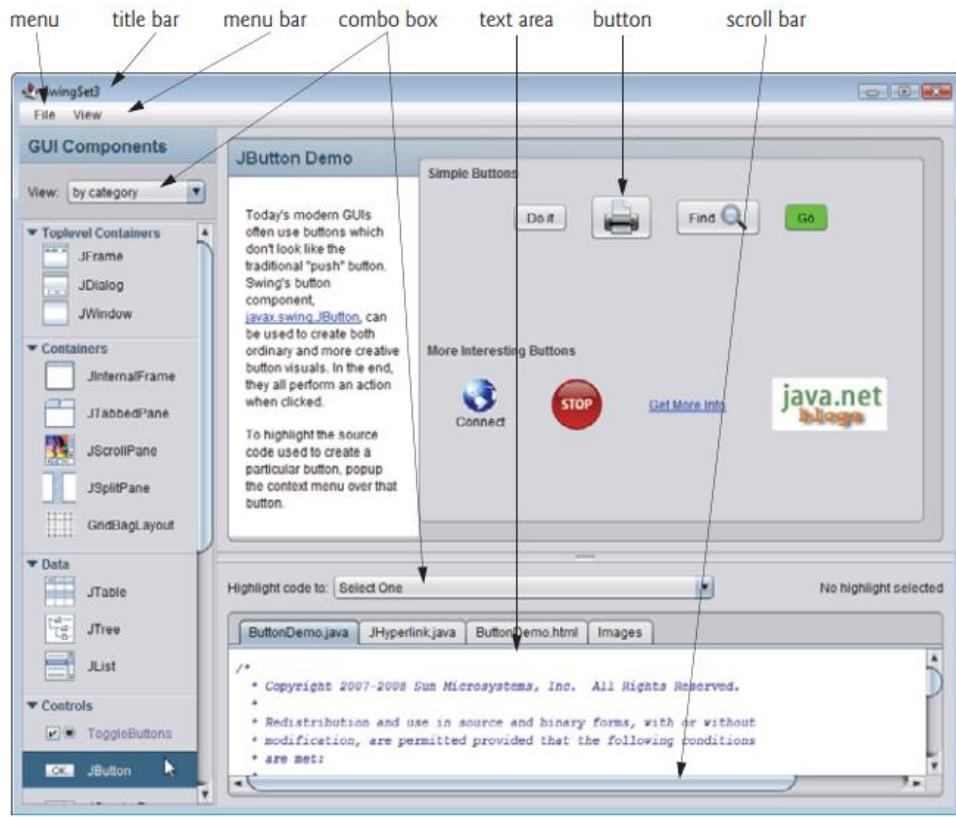
```

2. Lập trình giao diện GUI

a) Giới thiệu

GUI (*Graphical User Interface*) là giao diện đồ họa người dùng sử dụng những kỹ thuật để hiển thị những giao diện giúp người dùng khai thác các ứng dụng của máy tính. GUI được xây dựng từ các thành phần của GUI. Một thành phần của GUI là một đối tượng được người dùng tương tác thông qua chuột, bàn phím hoặc các thiết bị đầu vào khác.

Để lập trình giao diện GUI với Java, chúng ta có thể sử dụng hai gói là AWT (*Abstract Windowing Toolkit*) trong *java.awt* và gói *Swing* trong *javax.swing*.

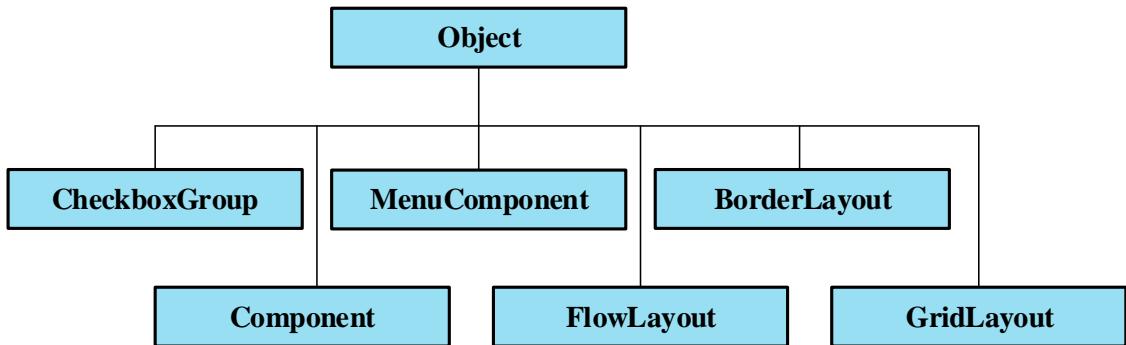


Hình 6.4. Các thành phần khác nhau tạo nên một giao diện người dùng.

AWT là viết tắt của *Abstract Windowing Toolkit*, nó là một tập các lớp Java cho phép xây dựng giao diện đồ họa. Các thành phần đồ họa trong AWT được xây dựng từ các thành phần đồ họa của hệ điều hành tương ứng, do đó GUI được xây dựng từ AWT có giao diện được quan sát *look-and-feel* ở mỗi hệ điều hành là khác nhau. AWT cung cấp các nhóm đối tượng khác nhau để xây dựng giao diện đồ họa, gồm có 4 nhóm như sau:

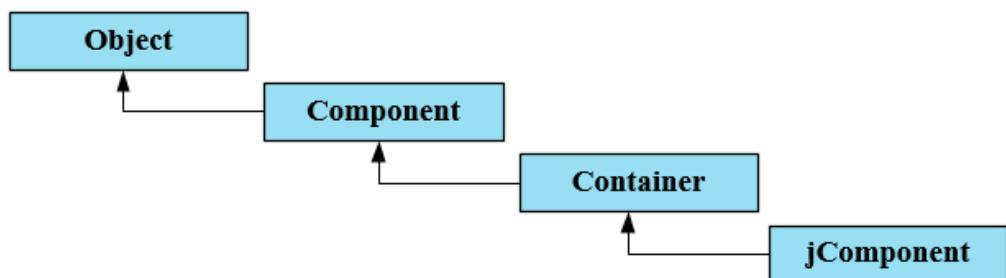
- *Container (khung chứa)*: là một đối tượng khung chứa dùng để quản lý và nhóm các đối tượng con trong lớp Component;
- *Component (thành phần)*: là một đối tượng có biểu diễn đồ họa được hiển thị trên màn hình mà người dùng có thể tương tác được. Chẳng hạn như những nút bấm (button), những ô checkbox, những thanh cuộn scrollbar,... Lớp Component là một lớp trừu tượng;

- *Layout manager (quản lý bố cục)*: khung chứa (Container) nhận các đối tượng từ bên ngoài đưa vào và nó phải biết làm thế nào để tổ chức, sắp xếp vị trí cho các đối tượng đó. Mỗi đối tượng khung chứa đều có một bộ quản lý bố cục;
- *Events (các sự kiện)*: là các lớp giúp xử lý các sự kiện chuột và bàn phím.



Hình 6.5. Sơ đồ phân cấp các lớp trong gói AWT.

Java Swing là một phần của *Java Foundation Classes (JFC)* được sử dụng để tạo các ứng dụng trên hệ điều hành Windows (*Window-Based*). Nó được xây dựng dựa trên *AWT* và được viết hoàn toàn bằng Java. Nhưng không giống như *AWT*, *Java Swing* trong gói *javax.swing* cung cấp cho người dùng các thành phần (Component) giúp thiết kế giao diện gọn nhẹ và độc lập với nền tảng các hệ điều hành.



Hình 6.6. Các siêu lớp phổ biến của các thành phần Swing.

Lớp *Component* của gói *java.awt* là một siêu lớp, khai báo các thành phần (component) xây dựng GUI trong gói *java.awt* và *javax.swing*. Bất kỳ một đối tượng *Container* nào cũng có thể được sử dụng để tổ chức thành các *Component* nếu gắn các *Component* vào *Container*.

Lớp JComponent của gói *javax.swing* là lớp con của Container. JComponent cũng là một siêu lớp và bởi vì JComponent là lớp con của Container nên tất cả các thành phần trong thư viện Swing cũng đều là Container. Lớp này gồm có một số thành phần cơ bản như Bảng 6.2:

Thành phần	Mô tả
JLabel	Hiển thị nhãn cho đối tượng
JTextField	nhận dữ liệu của người dùng nhập vào
JButton	Nút bấm nhận lệnh khi nhán chuột
JCheckBox	Đánh dấu vào ô tùy chọn
JComboBox	Một danh sách các lựa chọn thả xuống để chọn một giá trị
JList	một danh sách các lựa chọn được liệt kê,có thể chọn 1 hoặc nhiều giá trị trong danh sách đó
JPanel	Một vùng cửa sổ để tổ chức và đưa các thành phần vào

Bảng 6.2. Các thành phần cơ bản của gói Swing.

b) *Khung chứa trong lập trình GUI*

Container (khung chứa) là thành phần chủ chốt trong các thành phần của *swing GUI*. Một khung chứa cung cấp một không gian để đặt các thành phần của GUI và bên thân nó cũng chính là một thành phần, có khác chăng là nó có khả năng thêm các thành phần khác vào trong nó. Các lớp con của khung chứa là: *JFrame*, *JPanel* và *JWindow*.

- *JFrame*

Lớp JFrame kế thừa từ *java.awt.Frame* bổ sung các hỗ trợ cho cấu trúc thành phần *JFC/Swing*. Cú pháp khai báo:

```
public class JFrame
    extends Frame
    implements WindowConstants, Accessible, RootPaneContainer
```

JFrame dùng để xây dựng các cửa sổ ứng dụng đầu tiên, có chứa các thanh menu, toolbar, statusbar, và vùng nội dung.

Lớp *JFrame* này có các *constructor* sau:

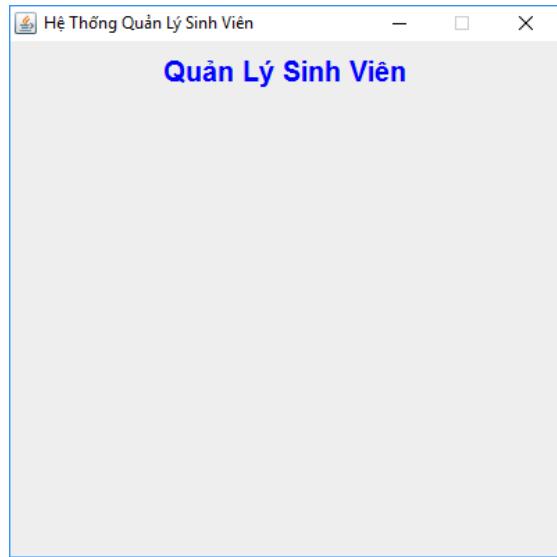
- + *JFrame()*: Xây dựng một *Frame* mới, ban đầu là không nhìn thấy (*invisible*);
- + *JFrame(GraphicsConfiguration gc)*: Tạo một *Frame* trong *GraphicsConfiguration* đã cho của một thiết bị màn hình và một tiêu đề trống;
- + *JFrame(String title)*: Tạo một *Frame* mới, ban đầu là không nhìn thấy (*invisible*) với tiêu đề đã cho;
- + *JFrame(String title, GraphicsConfiguration gc)*: Tạo một *Frame* với tiêu đề đã cho và *GraphicsConfiguration* đã cho của một thiết bị màn hình;

Ví dụ 6.5: Chương trình minh họa cách tạo một khung chứa *JFrame*.

```
package swingexercise;

import java.awt.*;
import javax.swing.*;

class SwingExercise extends JFrame
{
    public SwingExercise()
    {
        super("Hệ Thống Quản Lý Sinh Viên");
        setSize(new Dimension(400,400));
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false);
        setLayout(null);
        Header();
    }
    void Header()
    {
        JLabel lbttitle=new JLabel("Quản Lý Sinh Viên");
        lbttitle.setFont(new Font ("Arial",Font.BOLD,20));
        lbttitle.setBounds(110,10,180,25);
        lbttitle.setForeground(Color.blue);
        this.add(lbttitle);
    }
}
```



Hình 6.7. Kết quả chương trình minh họa JFrame.

- JPanel

Lớp JPanel là một khung chứa tổng quát và gọn nhẹ. Panel không tự đỗ màu cho cửa sổ ngoại trừ màu nền mặc định của nó và người dùng có thể dễ dàng thêm các đường biên phân chia giữa các thành phần với nhau và đỗ màu cho từng khu vực riêng rẽ. Cú pháp như sau:

```
public class JPanel  
    extends Jcomponent  
    implements Accessible
```

Lớp này bao gồm các *constructor* sau:

- + *JPanel()*: Tạo một JPanel mới với một double buffer và một Flow Layout;
- + *JPanel(boolean isDoubleBuffered)*: Tạo một JPanel mới với Flow Layout và trình đệm đã xác định;
- + *JPanel(LayoutManager layout)*: Tạo một JPanel mới với Layout Manager đã cho;
- + *JPanel(LayoutManager layout, boolean isDoubleBuffered)*: Tạo một JPanel mới với Layout Manager đã cho và trình đệm đã xác định;

- JWindow:

Lớp *JWindow* là khung chưa được dùng để tạo ra vùng cửa sổ trên màn hình nhưng không có đường biên và vì vậy người dùng thẻ không di chuyển cũng như thay đổi kích thước của cửa sổ được giống như với Frame và Panel. *JWindow* thường được sử dụng để tạo ra các popup, tooltip.

Cú pháp khai báo cho lớp *JWindow*:

```
public class JWindow  
    extends Window  
    implements Accessible, RootPaneContainer
```

Lớp *JWindow* có các *constructor* sau:

- + *JWindow()*: Tạo một cửa sổ mà không xác định khung sở hữu nó;
- + *JWindow(Frame owner)*: Tạo một cửa sổ với khung riêng đã cho;
- + *JWindow(GraphicsConfiguration gc)*: Tạo một cửa sổ với *GraphicsConfiguration* đã cho của một thiết bị màn hình;
- + *JWindow(Window owner)*: Tạo một cửa sổ với cửa sổ riêng đã cho;
- + *JWindow(Window owner, GraphicsConfiguration gc)*: Tạo một cửa sổ với cửa sổ sở hữu nó đã cho và *GraphicsConfiguration* đã cho của một thiết bị màn hình.

c) Các thành phần giao diện GUI cơ bản

- *JLabel*:

JLabel là một siêu lớp của *JComponent*, chỉ dùng để hiển thị nhãn dạng text, ảnh hoặc cả hai. Các ứng dụng hiếm khi thay đổi nội dung của nhãn sau khi tạo nó.

Cú pháp khai báo lớp *JLabel*:

```
public class JLabel  
    extends JComponent  
    implements SwingConstants, Accessible
```

Các constructor của lớp *JLabel* trong Java Swing:

- + *JLabel()*: Tạo một instance của JLabel, không có hình ảnh, và với một chuỗi trống cho title;
- + *JLabel(Icon image)*: Tạo một instance của JLabel với hình ảnh đã cho;
- + *JLabel(Icon image, int horizontalAlignment)*: Tạo một instance của JLabel với hình ảnh và căn chỉnh ngang đã cho;
- + *JLabel(String text)*: Tạo một instance của JLabel với text đã cho;
- + *JLabel(String text, Icon icon, int horizontalAlignment)*: Tạo một instance của JLabel với text, hình ảnh, và căn chỉnh ngang đã cho;
- + *JLabel(String text, int horizontalAlignment)*: Tạo một instance của JLabel với text và căn chỉnh ngang đã cho;

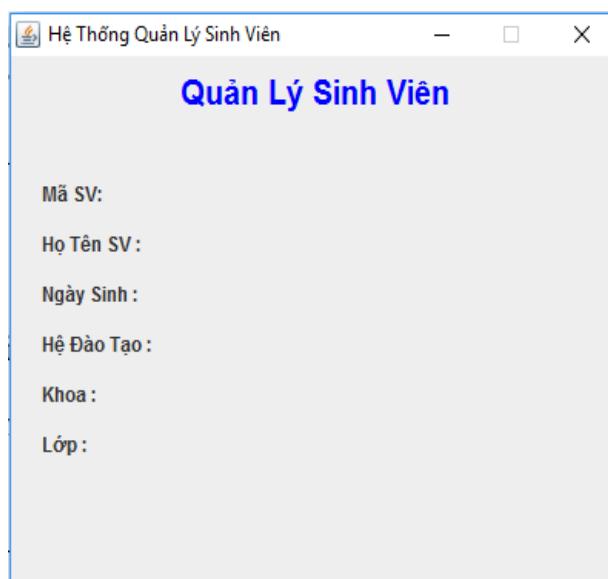
Các phương thức thường được sử dụng của lớp JLabel

- + *void setDisabledIcon(Icon disabledIcon)*: Thiết lập icon để được hiển thị nếu JLabel này là "disabled" (JLabel.setEnabled(false));
- + *void setHorizontalAlignment(int alignment)*: Thiết lập sự căn chỉnh nội dung của label theo trục X;
- + *void setVerticalAlignment(int alignment)*: Thiết lập sự căn chỉnh nội dung của label theo trục Y;
- + *void setHorizontalTextPosition(int textPosition)*: Thiết lập vị trí theo chiều ngang của phần text của label, cân xứng với hình ảnh của nó;
- + *void setIcon(Icon icon)*: Định nghĩa icon mà thành phần này sẽ hiển thị;
- + *void setIconTextGap(int iconTextGap)*: Nếu cả hai thuộc tính icon và text được thiết lập, thuộc tính này xác định khoảng trống giữa chúng;
- + *void setText(String text)*: Định nghĩa trường text dòng đơn mà thành phần này sẽ hiển thị;

+ void setVerticalTextPosition(int textPosition): Thiết lập vị trí theo chiều dọc của phần text của label, cân xứng với hình ảnh của nó.

Ví dụ 6.6: Tạo GUI với JLabel

```
void LabelFrame ()  
{  
    lbmasv = new JLabel("Mã SV:");  
    lbmasv.setBounds(20,70,100,25);  
    add(lbmasv);  
  
    lbhoten=new JLabel("Họ Tên SV :");  
    lbhoten.setBounds(20,100,100,25);  
    add(lbhoten);  
  
    lbngaysinh=new JLabel("Ngày Sinh :");  
    lbngaysinh.setBounds(20,130,100,25);  
    add(lbngaysinh);  
  
    lbhedaotao =new JLabel("Hệ Đào Tạo :");  
    lbhedaotao.setBounds(20,160,100,25);  
    add(lbhedaotao);  
  
    lbkhoa=new JLabel("Khoa :");  
    lbkhoa.setBounds(20,190,100,25);  
    add(lbkhoa);  
  
    lblogp = new JLabel("Lớp :");  
    lblogp.setBounds(20,220,100,25);  
    add(lblogp);  
}  
.....
```



Hình 6.8: Ví dụ tạo Jlabel.

- JTextField:

JTextField kế thừa lớp *JTextComponent* trong gói *javax.swing.text*, cho phép người dùng nhập dữ liệu từ bàn phím trên một dòng đơn, các ứng dụng cũng có thể trả về dữ liệu hiển thị trên *JTextField* này.

Cú pháp khai báo lớp *javax.swing.JTextField*:

```
public class JTextField  
    extends JTextComponent  
    implements SwingConstants
```

Các *constructor* của lớp *JTextField* trong Java Swing:

- + *JTextField()*: Xây dựng một TextField mới;
- + *JTextField(Document doc, String text, int columns)*: Xây dựng một JTextField mới mà sử dụng mô hình lưu trữ text đã cho và số cột đã cho;
- + *JTextField(int columns)*: Xây dựng một TextField mới và trống với số cột đã cho;
- + *JTextField(String text)*: Xây dựng một TextField mới được khởi tạo với text đã cho;
- + *JTextField(String text, int columns)*: Xây dựng một TextField mới được khởi tạo với text và các cột đã cho;

Các phương thức được sử dụng phổ biến của lớp *JTextField*

- + *void setHorizontalAlignment(int alignment)*: Thiết lập căn chỉnh ngang cho text;
- + *void setActionCommand(String command)*: Thiết lập chuỗi lệnh được sử dụng cho action event;
- + *void addActionListener(ActionListener l)*: Thêm action listener đã cho để nhận các action event từ textfield này;

- + *void removeActionListener(ActionListener l)*: Xóa action listener đã cho để nó không bao giờ nhận action event từ textfield này nữa;
- + *void scrollRectToVisible(Rectangle r)*: Cuốn truwong này sang trái hoặc phải.

Ví dụ 6.7: Tạo GUI với JTextField

```

void LabelFrame ()
{
    lbmasv = new JLabel("Mã SV:");
    lbmasv.setBounds(20,70,100,25);

    add(lbmasv);

    tfmasv=new JTextField();
    tfmasv.setBounds(120,70,100,25);
    add(tfmasv);
    getContentPane().add(tfmasv);

    lbhoten=new JLabel("Họ Tên SV :");
    lbhoten.setBounds(20,100,100,25);
    add(lbhoten);

    tfhoten=new JTextField();
    tfhoten.setBounds(120,100,200,25);
    add(tfhoten);
    getContentPane().add(tfhoten);

    lbngaysinh=new JLabel("Ngày Sinh :");
    lbngaysinh.setBounds(20,130,100,25);
    add(lbngaysinh);

    tfngaysinh = new JTextField();
    tfngaysinh.setBounds(120,130,200,25);
    add(tfngaysinh);
    getContentPane().add(tfngaysinh);

    lbhedaotao =new JLabel("Hệ Đào Tạo :");
    lbhedaotao.setBounds(20,160,100,25);
    add(lbhedaotao);

    tfhedaotao = new JTextField();
    tfhedaotao.setBounds(120,160,100,25);
    add(tfhedaotao);
    getContentPane().add(tfhedaotao);

    lbkhoa=new JLabel("Khoa :");
    lbkhoa.setBounds(20,190,100,25);
    add(lbkhoa);

    tfkhoa = new JTextField();
    tfkhoa.setBounds(120,190,200,25);
    add(tfkhoa);
}

```

```

        getContentPane().add(tfkhoa);

        lblop = new JLabel("Lớp :");
        lblop.setBounds(20, 220, 100, 25);
        add(lblop);

        tflop=new JTextField();
        tflop.setBounds(120, 220, 100, 25);

        add(tflop);
        getContentPane().add(tflop);
    }
}

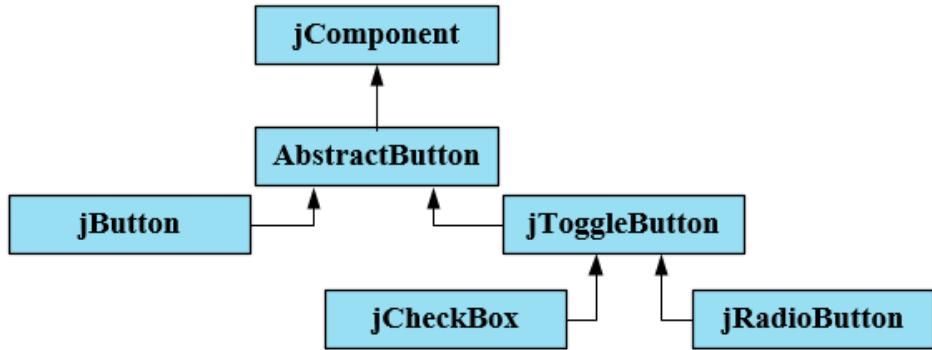
```



Hình 6.9: Ví dụ cách tạo JTextField.

- JButton

Button là một thành phần cho phép người dùng nhấp vào để thực hiện một action. Một ứng dụng Java có thể sử dụng một vài đối tượng nút bấm (button) như: *command buttons, checkboxes, toggle buttons và radio buttons*. Hình 6.7 là sơ đồ thể hiện sự kế thừa của các nút bấm, tất cả các loại nút bấm đều là các lớp con của *AbstractButton* (gói *javax.swing*), khai báo những tính năng chung của các nút bấm trong Swing. Trong phần này chúng ta chỉ tập trung vào các loại nút bấm điển hình trong việc khởi tạo lệnh.



Hình 6.10. Sơ đồ thể hiện sự kế thừa của các button trong jComponent.

Một nút bấm sẽ sinh ra một ActionEvent khi người dùng nhấn chọn nó. Các button lệnh được tạo ra bởi lớp *JButton*. Dòng chữ hiển thị trên nút bấm được gọi là nhãn của nút bấm đó. Một GUI có thể có nhiều JButton, nhưng mỗi một nhãn của nút bấm là duy nhất.

Cú pháp khai báo cho lớp *javax.swing.JButton*:

```

public class JButton
    extends AbstractButton
    implements Accessible

```

Lớp JButton có các *constructor* sau:

- + *JButton()*: Tạo một nút bấm *button* mà không thiết lập *text* hoặc *icon*;
- + *JButton(Action a)*: Tạo một *button* tại đây các thuộc tính được nhận từ Action đã cung cấp;
- + *JButton(Icon icon)*: Tạo một button với một icon;
- + *JButton(String text)*: Tạo một button với text;
- + *JButton(String text, Icon icon)*: Tạo một button gồm text và icon.

Một số phương thức được sử dụng phổ biến của lớp *AbstractButton* được liệt kê dưới đây:

- + *public void setText(String s)*: dùng để thiết lập text đã cho trên button;
- + *public String getText()*: được sử dụng để trả về text của button;

- + *public void setEnabled(boolean b)*: dùng để kích hoạt hoặc vô hiệu hóa button;
- + *public void setIcon(Icon b)*: được sử dụng để thiết lập Icon đã cho trên button;
- + *public void addActionListener(ActionListener a)*: được sử dụng để bộ lắng nghe sự kiện tới đối tượng này.

Ví dụ 6.8: Chương trình sau minh họa cách tạo nút bấm với JButton

```

.....
bntThem=new JButton("Thêm");
bntThem.setBounds(50,270,75,25);
add(bntThem);
getContentPane().add(bntThem);

bntLuu=new JButton("Lưu");
bntLuu.setBounds(125,270,75,25);
add(bntLuu);
getContentPane().add(bntLuu);

bntSua=new JButton("Sửa");
bntSua.setBounds(200,270,75,25);
add(bntSua);
getContentPane().add(bntSua);

bntTimkiem=new JButton("Tìm SV");
bntTimkiem.setBounds(275,270,75,25);
add(bntTimkiem);
getContentPane().add(bntTimkiem);
.....

```



Hình 6.11. Kết quả chương trình minh họa JButton.

- JComboBox

ComboBox hay còn có tên gọi khác là danh sách thả xuống (*drop-down list*) cho phép người dùng chọn một giá trị từ một danh sách; các *combobox* được triển khai từ lớp *JComboBox* kế thừa lớp *JComponent*. *JComboBox* sinh ra các *ItemEvents* giống như *JCheckBox* và *JRadioButton*.

Cú pháp khai báo cho lớp *JComboBox* là:

```
public class JComboBox  
    extends JComponent  
    implements ItemSelectable, ListDataListener,  
    ActionListener, Accessible
```

Các *constructor* được sử dụng phổ biến của lớp *JComboBox*:

- + *JComboBox()*: Tạo một *JComboBox* với mẫu dữ liệu mặc định;
- + *JComboBox(Object[] items)*: Tạo một *JComboBox* mà chứa các phần tử trong mảng đã cho;
- + *JComboBox(Vector<?> items)*: Tạo một *JComboBox* mà chứa các phần tử trong *Vector* đã cho;

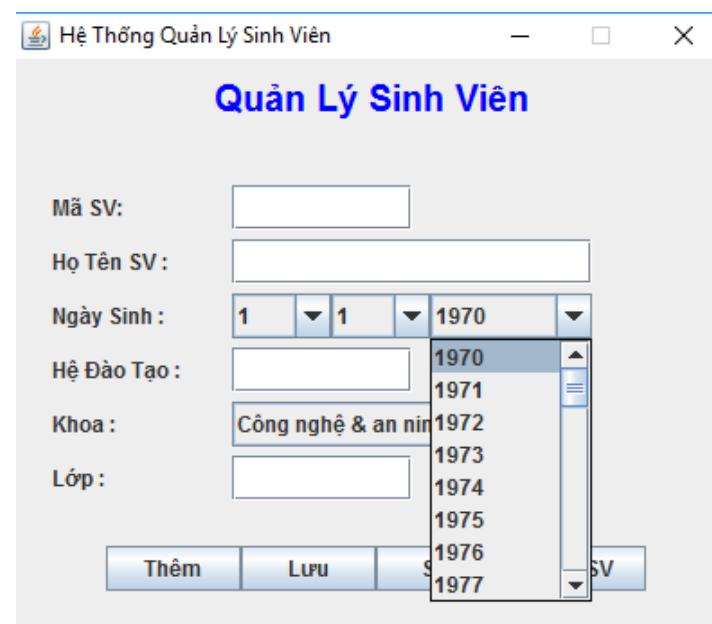
Các phương thức được sử dụng phổ biến của lớp *JComboBox*:

- + *public void addItem(Object anObject)*: được sử dụng để thêm một giá trị vào danh sách;
- + *public void removeItem(Object anObject)*: được sử dụng để xóa một giá trị vào danh sách;
- + *public void removeAllItems()*: được sử dụng để xóa tất cả một giá trị vào danh sách;
- + *public void setEditable(boolean b)*: được sử dụng để xác định xem có hay không *JComboBox* có thể chỉnh sửa được;
- + *public void addActionListener(ActionListener a)*: được sử dụng để thêm bộ lắng nghe các sự kiện hoạt động;

+ *public void addItemListener(ItemListener a)*: được sử dụng để thêm ItemListener;

Ví dụ 6.9: Ví dụ sau đây minh họa cách sử dụng một JComboBox để cung cấp một danh sách các tuỳ chọn ngày tháng năm sinh

```
.....  
cbngay=new JComboBox();  
cbngay.setBounds(120,130,55,25);  
cbthang=new JComboBox();  
cbthang.setBounds(175,130,55,25);  
  
cbnam=new JComboBox();  
cbnam.setBounds(230,130,90,25);  
for(int i=1;i<=31;i++)  
{  
    if(i<=12) cbthang.addItem(i);  
    cbngay.addItem(i);  
}  
for(int j=1970;j<=2010;j++)  
{  
    cbnam.addItem(j);  
}  
add(cbngay);  
getContentPane().add(cbngay);  
add((cbthang));  
getContentPane().add(cbthang);  
add(cbnam);  
getContentPane().add(cbnam);  
.....
```



Hình 6.12. Kết quả chạy thử nghiệm minh họa JComboBox.

- JList

Là một danh sách hiển thị một dãy các giá trị người dùng có thể chọn một hoặc nhiều giá trị. Các giá trị này được tạo ra bởi lớp *JList* kế thừa lớp *JComponent*. Lớp *JList* hỗ trợ cả những danh sách chỉ cho phép chọn một giá trị và chọn nhiều giá trị.

Cú pháp khai báo của lớp *JList* là:

```
public class JList  
    extends JComponent  
    implements Scrollable, Accessible
```

Các *constructor* của lớp *JList* trong *Java Swing*:

- + *JList()*: Xây dựng một *JList* với một mẫu là *empty, read-only*.
- + *JList(ListModel dataModel)*: Xây dựng một *JList* mà hiển thị các phần tử từ mô hình đã cho;
- + *JList(Object[] listData)*: Xây dựng một *JList* mà hiển thị các phần tử trong mảng đã cho;
- + *JList(Vector<?> listData)*: Xây dựng một *JList* mà hiển thị các phần tử trong *Vector* đã cho.

Các phương thức phổ biến của lớp *JList*

- + *void clearSelection()*: Xóa lựa chọn sau khi gọi phương thức này, trạng thái *isSelectionEmpty* sẽ trả về *true*;
- + *void setFixedCellHeight(int height)*: Thiết lập một giá trị cố định để được sử dụng cho chiều cao của mỗi ô trong danh sách;
- + *void setLayoutOrientation(int layoutOrientation)*: Định nghĩa cách các ô trong danh sách được bố trí;
- + *void addListSelectionListener(ListSelectionListener listener)*: Thêm một bộ lắng nghe sự kiện tới danh sách, để được thông báo mỗi khi xuất hiện một thay đổi tới lựa chọn; đây là cách ưu tiên để nghe các trạng thái của thay đổi;

d) Quản lý bố cục

Layout managers – quản lý bố cục là các đối tượng quản lý cách sắp xếp các thành phần trong một GUI. Chúng ta sử dụng quản lý bố cục để thực hiện sắp xếp cho toàn bộ các thành phần trên giao diện thay vì phải chỉ ra chính xác vị trí và kích thước của từng đối tượng trên giao diện. Tất cả các quản lý bố cục được triển khai từ giao diện *interface LayoutManager* của gói *java.awt*. Phương thức *setLayout* của lớp *Container* lấy một đối tượng ở giao diện *LayoutManager* như một đối số. Có một số loại quản lý bố cục như sau:

- *FlowLayout*: sắp xếp các đối tượng từ trái qua phải và từ trên xuống dưới. Các đối tượng đều giữ nguyên kích thước của nó.

Ví dụ 6.10: Chương trình minh họa cách sắp xếp bố cục với *FlowLayout*

```
public class SwingExercise extends JFrame
{
    private FlowLayout layout;
    private Container container;

    public SwingExercise()
    {
        super("Hệ Thống Quản Lý Sinh Viên");
        layout = new FlowLayout(); // tạo FlowLayout
        container = getContentPane();
        setLayout(layout);
        Header();
        LabelFrame();
    }
    void Header()
    {
        .....
    }
    void LabelFrame()
    {
        .....
    }
}
```



Hình 6.13. Kết quả thu được với bố cục FlowLayout.

- *BorderLayout*: Các đối tượng được đặt theo các đường viền khung chứa theo các cạnh đông, tây, nam, bắc và trung tâm (hay trái, phải, trên, dưới và giữa).

Ví dụ 6.11: Ví dụ minh họa về bố cục BorderLayout

```

import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class BorderLayoutFrame extends JFrame implements ActionListener {
    private JButton buttons[]; // khai báo một mảng các button
    private final String names[] = { "Hide North", "Hide South",
        "Hide East", "Hide West", "Hide Center" };
    private BorderLayout layout; // đối tượng BorderLayout

    // set up GUI and event handling
    public BorderLayoutFrame() {
        super( "BorderLayout Demo" );

        layout = new BorderLayout( 5, 5 ); // khoảng trống giữa
        các button là 5 pixels
        setLayout( layout ); // thiết lập bố cục khung
        buttons = new JButton[ names.length ]; // thiết lập kích
        thước mảng

        // tạo JButton và đăng ký các bộ lắng nghe cho chúng
        for ( int count = 0; count < names.length; count++ ) {
            buttons[ count ] = new JButton( names[ count ] );
            buttons[ count ].addActionListener( this );
        }
    }
}

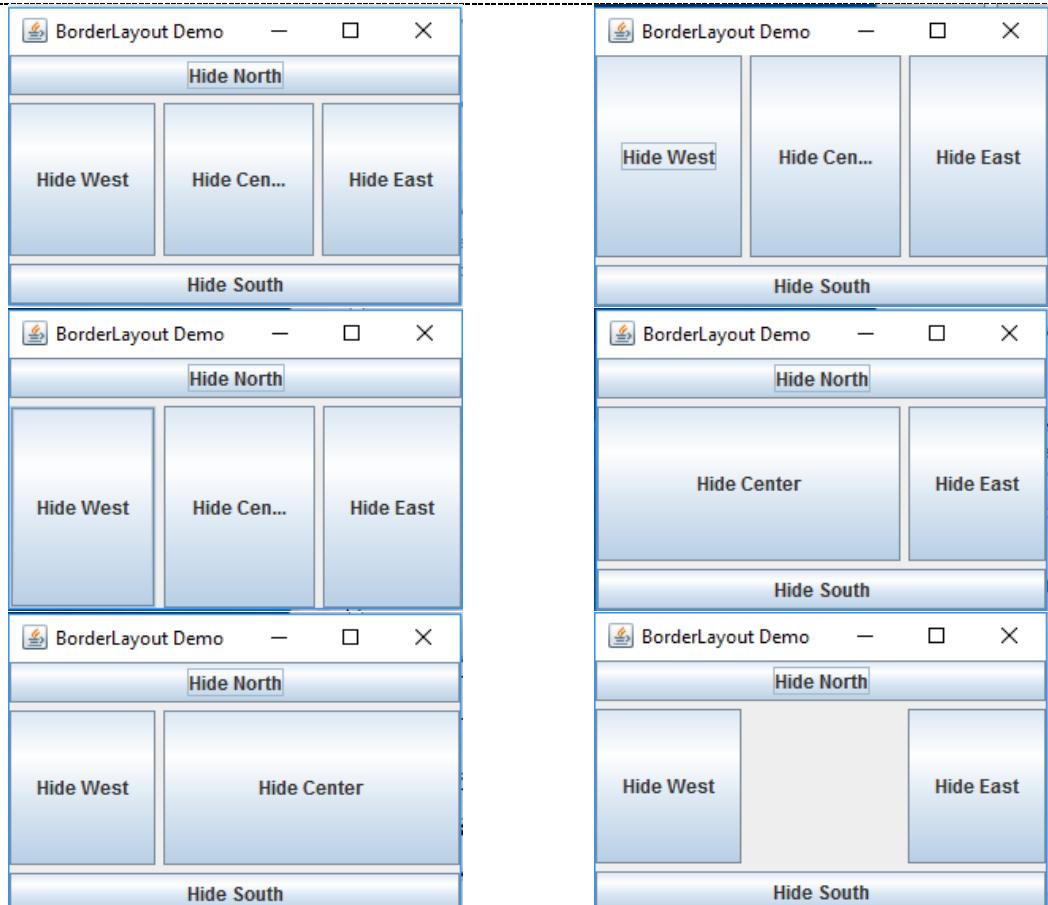
```

```

        add( buttons[ 0 ], BorderLayout.NORTH );
        add( buttons[ 1 ], BorderLayout.SOUTH );
        add( buttons[ 2 ], BorderLayout.EAST );
        add( buttons[ 3 ], BorderLayout.WEST );
        add( buttons[ 4 ], BorderLayout.CENTER );
    }

    // xử lý các sự kiện button
    public void actionPerformed( ActionEvent event ){
        // kiểm tra nguồn sự kiện và thanh nội dung bối cục tương ứng
        for ( JButton button : buttons ){
            if ( event.getSource() == button )
                button.setVisible( false ); // ẩn các button đã chọn
            else
                button.setVisible( true ); // hiển thị các button khác
        }
        layout.layoutContainer( getContentPane() ); // thanh nội dung bối cục
    }
}

```



Hình 6.14. Kết quả thu được với bối cục BorderLayout.

- *GridLayout*: Tạo một khung lưới vô hình với các ô bằng nhau. Các đối tượng sẽ đặt vừa kích thước với từng ô đó. Thứ tự sắp xếp cũng từ trái qua phải và từ trên xuống dưới.

Ví dụ 6.12: Ví dụ minh họa bố cục GridLayout

```

import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class GridLayoutFrame extends JFrame implements ActionListener {
    private JButton buttons[]; // mảng các button
    private final String names[] =
        { "one", "two", "three", "four", "five", "six" };
    private boolean toggle = true; // kết nối hai bộ cục
    private Container container; // khung chứa
    private GridLayout gridLayout1; // bộ cục lưới 1
    private GridLayout gridLayout2; // bộ cục lưới 2

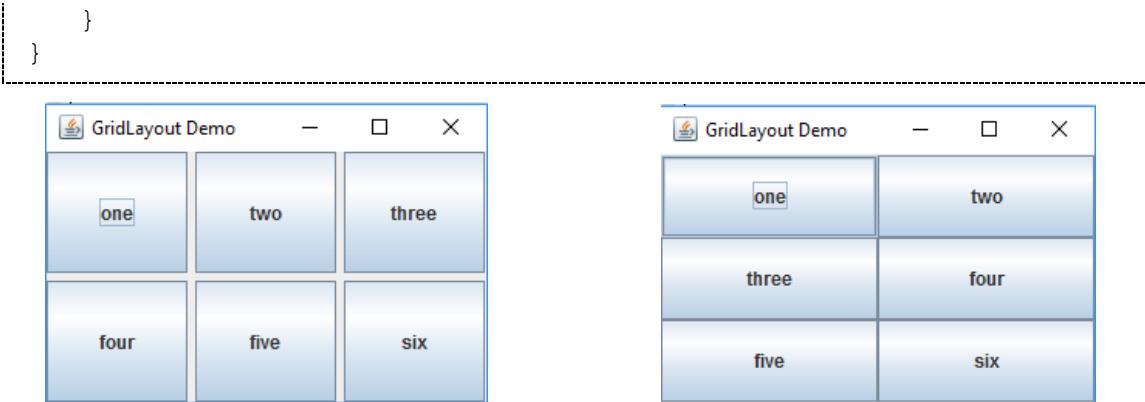
    public GridLayoutFrame() {
        super( "GridLayout Demo" );
        gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // thiết lập
        lưới 2 hàng 3 cột; khoảng cách giữa các button là 5 pixels
        gridLayout2 = new GridLayout( 3, 2 ); // thiết lập lưới 3
        hàng 2 cột; không có khoảng cách giữa các button
        container = getContentPane(); // lấy thanh nội dung
        setLayout( gridLayout1 ); // thiết lập bộ cục JFrame
        buttons = new JButton[ names.length ]; // tạo mảng JButton

        for ( int count = 0; count < names.length; count++ ) {
            buttons[ count ] = new JButton( names[ count ] );
            buttons[ count ].addActionListener( this ); // đăng ký
            bộ lắng nghe sự kiện
            add( buttons[ count ] ); // thêm nút vào JFrame
        }
    }

    // xử lý sự kiện để kết nối các bộ cục
    public void actionPerformed( ActionEvent event ) {
        if ( toggle )
            container.setLayout( gridLayout2 ); // thiết lập bộ cục 2
        else
            container.setLayout( gridLayout1 ); // thiết lập bộ cục 1

        toggle = !toggle; // thiết lập liên kết 2 button tương ứng
        trong hai bộ cục
        container.validate(); // hiển thị lại container sau thay
        đổi
    }
}

```

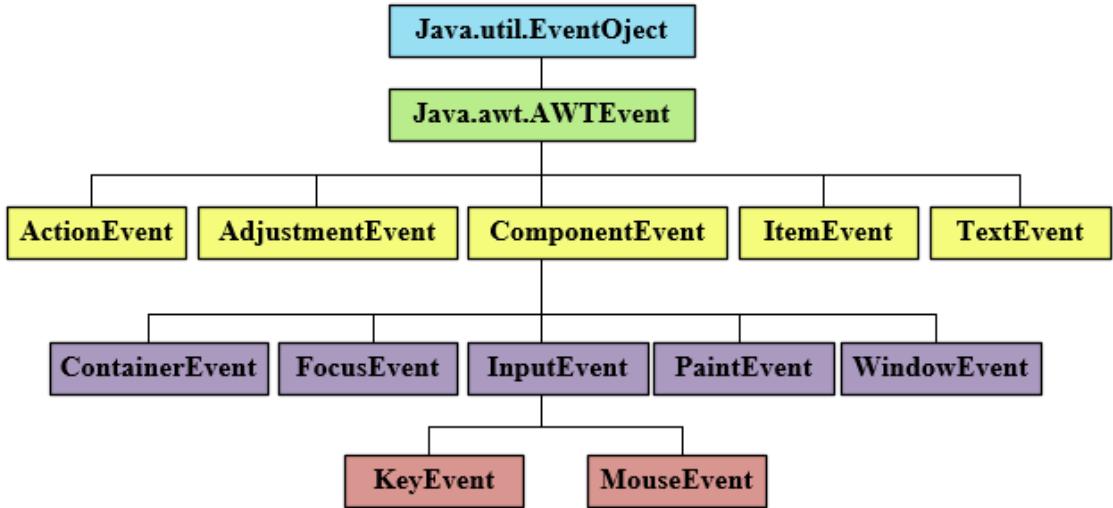


Hình 6.15: Kết quả minh họa bố cục GridLayout.

- *NullLayout*: cho phép trình bày các đối tượng không theo một quy tắc nào. Tất cả đều do người dùng tự định vị và thiết lập kích thước cho mỗi đối tượng.

e) Xử lý các sự kiện

Ở trên chúng ta chỉ đề cập đến vấn đề thiết kế giao diện chương trình ứng dụng mà chưa đề cập đến vấn đề xử lý sự kiện. Những sự kiện được phát sinh khi người dùng tương tác với giao diện chương trình. Những tương tác thường gặp như: di chuyển, nhấp chuột, chọn một mục trong hệ thống thực đơn, nhập dữ liệu trong một ô văn bản, đóng cửa sổ ứng dụng... Khi có một tương tác xảy ra thì một sự kiện được gửi đến chương trình. Thông tin về sự kiện thường được lưu trữ trong một đối tượng dẫn xuất từ lớp *AWTEvent*. Những kiểu sự kiện trong gói *java.awt.event* có thể dùng cho cả những *component AWT* và *JFC*. Đối với thư viện *JFC* thì có thêm những kiểu sự kiện mới trong gói *java.swing.event*. Sơ đồ dưới đây minh họa phân cấp nhiều lớp sự kiện từ gói *java.awt.event*.

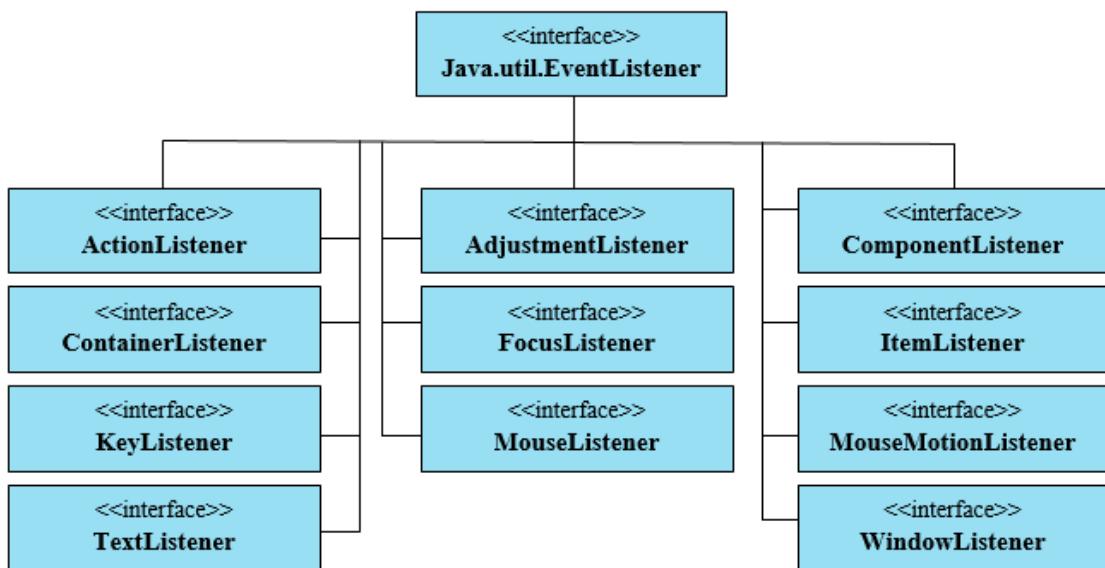


Hình 6.16. Nhữn lớp sự kiện của gói `java.awt.events`.

Khi xử lý sự kiện, có ba yếu tố quan trọng cần phải chỉ rõ là:

- Nguồn phát sinh sự kiện (*event source*): là thành phần của giao diện mà người dùng tác động;
- Sự kiện (*event object*): Tóm tắt thông tin về sự kiện xảy ra, bao gồm tham chiếu đến nguồn gốc phát sinh sự kiện và thông tin sự kiện sẽ gửi đến cho bộ lắng nghe sự kiện;
- Bộ lắng nghe sự kiện (*event listener*): là một đối tượng của một lớp hiện thực một hay nhiều *interface* của gói `java.awt.event` hay `java.swing.event`. Khi được thông báo, bộ lắng nghe nhận sự kiện và xử lý. Nguồn phát sinh sự kiện phải cung cấp những phương thức để đăng ký hoặc hủy bỏ một bộ lắng nghe. Nguồn phát sinh sự kiện luôn phải gắn với một bộ lắng nghe, và nó sẽ thông báo với bộ lắng nghe đó khi có sự kiện phát sinh đó;

Bên cạnh đó, cần thực hiện hai công việc trong xử lý sự kiện đó là: Tạo và đăng ký một bộ lắng nghe cho một *component* trên *GUI*; cài đặt các phương thức quản lý và xử lý sự kiện.



Hình 6.17. Các giao diện lắng nghe sự kiện của gói `java.util.event`.

Một đối tượng *Event-Listener* lắng nghe những sự kiện khác nhau phát sinh từ các *component* của giao diện chương trình. Với mỗi sự kiện khác nhau phát sinh thì phương thức tương ứng trong những *Event-Listener* sẽ được gọi thực hiện.

Mỗi giao diện *Event-Listener* gồm một hay nhiều các phương thức mà chúng cần cài đặt trong các lớp triển khai của giao diện đó. Những phương thức trong các giao diện là trừu tượng vì vậy lớp (bộ lắng nghe) nào hiện thực các giao diện này thì phải cài đặt tất cả những phương thức của giao diện đó, nếu không thì các bộ lắng nghe sẽ trở thành các lớp trừu tượng.

Dưới đây chúng ta sẽ đi tìm hiểu xử lý sự kiện chuột và xử lý sự kiện bàn phím:

- Xử lý sự kiện chuột

Java cung cấp hai giao diện lắng nghe là `MouseListener` và `MouseMotionListener` để quản lý và xử lý các sự kiện liên quan đến thiết bị chuột. Những sự kiện chuột có thể sử dụng cho bất kỳ thành phần nào trên GUI mà dẫn xuất từ `java.awt.Component`.

Các phương thức của *interface MouseListener* và *MouseMotionListener* gồm có:

- + *public void mousePressed(MouseEvent event)*: được gọi khi một nút chuột được nhấn ở trên component;
- + *public void mouseClicked(MouseEvent event)*: được gọi khi một nút chuột được nhấn và nhả trên component mà không di chuyển chuột;
- + *public void mouseReleased(MouseEvent event)*: được gọi khi một nút chuột nhả ra khi kéo rẽ;
- + *public void mouseEntered(MouseEvent event)*: được gọi khi con trỏ chuột vào trong đường biên của một component;
- + *public void mouseExited(MouseEvent event)*: được gọi khi con trỏ chuột ra khỏi đường biên của một component.

Các phương thức của *interface MouseMotionListener*:

- + *public void mouseDragged(MouseEvent event)*: phương thức này được gọi khi người dùng nhấn một nút chuột và kéo trên một thành phần;
- + *public void mouseMoved(MouseEvent event)*: phương thức này được gọi khi di chuyển chuột trên một thành phần;

Mỗi phương thức xử lý sự kiện chuột có một tham số *MouseEvent* chứa thông tin về sự kiện chuột phát sinh như: tọa độ x, y nơi sự kiện chuột xảy ra. Những phương thức tương ứng trong các giao diện sẽ tự động được gọi khi chuột tương tác với một thành phần. Để biết được người dùng đã nhấn nút chuột nào, chúng ta dùng những phương thức, những hằng số của lớp *InputEvent* (là lớp cha của lớp *MouseEvent*).

Ví dụ 6.13: Xây dựng chương trình MouseTracker minh họa việc sử dụng các giao diện *MouseListener* và *MouseMotionListener*. Trong đó, cần phải khai báo 7 phương thức để triển khai. Mỗi sự kiện chuột trong ví dụ này hiển thị tên sự kiện chuột đã thực hiện trong *JLabel* được gọi bởi *statusBar*.

```
Import java.awt.Color;
import java.awt.BorderLayout;
```

```

import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MouseTrackerFrame extends JFrame {
    private JPanel mousePanel; // khai báo khung chứa nơi xảy ra
sự kiện chuột
    private JLabel statusBar; // nhãn hiển thị thông tin sự kiện

    // Hàm khởi tạo MouseTrackerFrame thiết lập GUI và
    // đăng ký xử lý sự kiện chuột
    public MouseTrackerFrame() {
        super( "Demonstrating Mouse Events" );
        mousePanel = new JPanel(); // tạo khung
        mousePanel.setBackground( Color.WHITE ); // thiết lập màu
nền
        add( mousePanel, BorderLayout.CENTER ); // thêm một khung
vào JFrame
        statusBar = new JLabel( "Mouse outside JPanel" );
        add( statusBar, BorderLayout.SOUTH );//thêm nhãn vào
JFrame
        //tạo và đăng ký bộ lắng nghe cho sự kiện chuột và di
chuyển chuột
        MouseHandler handler = new MouseHandler();
        mousePanel.addMouseListener( handler );
        mousePanel.addMouseMotionListener( handler );
    }
    private class MouseHandler implements MouseListener,
MouseMotionListener {
        // Xử lý sự kiện MouseListener
        // Xử lý sự kiện khi chuột được nhả ra sau khi nhấn
        public void mouseClicked( MouseEvent event ) {
            statusBar.setText( String.format( "Clicked at [%d,
%d]", event.getX(), event.getY() ) );
        }

        // Xử lý sự kiện khi nhấn chuột
        public void mousePressed( MouseEvent event ) {
            statusBar.setText( String.format( "Pressed at [%d,
%d]", event.getX(), event.getY() ) );
        }

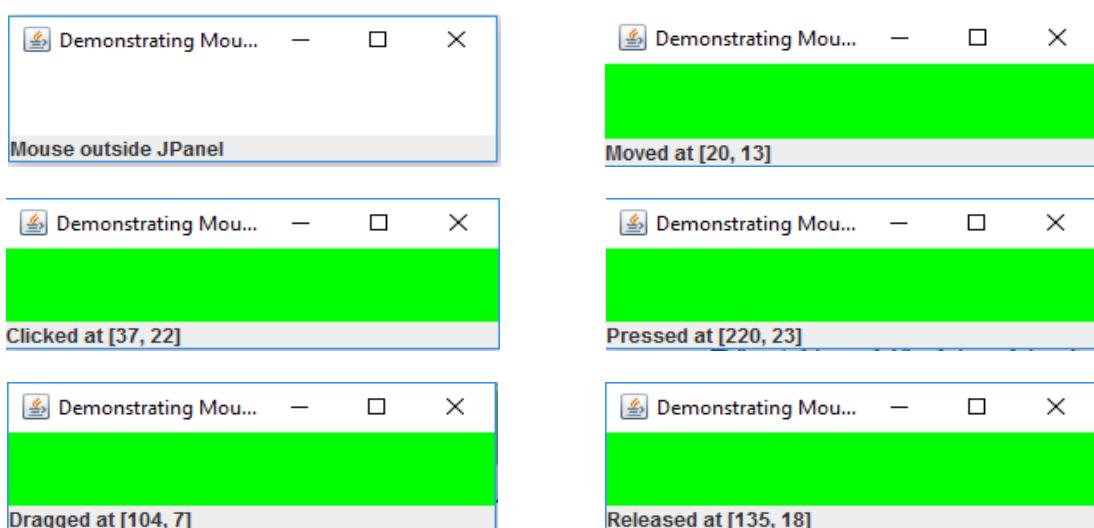
        // xử lý sự kiện khi chuột nhả ra sau khi kéo
    }
}

```

```

        public void mouseReleased( MouseEvent event ){
            statusBar.setText( String.format( "Released at [%,d,
%,d]", event.getX(), event.getY() ) );
        }
        // trả về tọa độ chuột khi chuột vào cửa sổ
        public void mouseEntered( MouseEvent event ){
            statusBar.setText( String.format( "Mouse entered at
[%,d, %d]", event.getX(), event.getY() ) );
            mousePanel.setBackground( Color.GREEN );
        }
        // xử lý sự kiện khi chuột ra khỏi cửa sổ
        public void mouseExited( MouseEvent event ){
            statusBar.setText( "Mouse outside JPanel" );
            mousePanel.setBackground( Color.WHITE );
        }
        // xử lý sự kiện MouseMotionListener
        // xử lý sự kiện khi chuột kéo với nút nhấn
        public void mouseDragged( MouseEvent event ){
            statusBar.setText( String.format( "Dragged at [%,d,
%,d]", event.getX(), event.getY() ) );
        }
        // xử lý sự kiện khi chuột bị di chuyển
        public void mouseMoved( MouseEvent event ){
            statusBar.setText( String.format( "Moved at [%,d,
%,d]", event.getX(), event.getY() ) );
        }
    }
}

```



Hình 6.18. Kết quả kiểm thử chương trình MouseTrakerFrame.

- Xử lý sự kiện bàn phím

Để xử lý sự kiện bàn phím trong java hỗ trợ một bộ lắng nghe sự kiện là giao

diện *KeyListener*. Một sự kiện bàn phím được phát sinh khi người dùng nhấn và nhả một phím trên bàn phím. Một lớp hiện thực *KeyListener* phải cài đặt các phương thức *keyPressed*, *keyReleased* và *keyTyped*. Mỗi phương thức này có một tham số là một đối tượng kiểu *KeyEvent*. *KeyEvent* là lớp con của lớp *InputEvent*. Các phương thức của *interface KeyListener* được liệt kê như sau:

- + Phương thức *keyPressed* được gọi khi một phím bất kỳ được nhấn;
- + Phương thức *keyTyped* được gọi thực hiện khi người dùng nhấn một phím không phải “phím hành động” (như phím mũi tên, phím Home, End, Page Up, Page Down, các phím chức năng như: Num Lock, Print Screen, Scroll Lock, Caps Lock, Pause);
- + Phương thức *keyReleased* được gọi thực hiện khi nhả phím nhấn sau khi sự kiện *keyPressed* hoặc *keyTyped*;

Ví dụ 6.14: Chương trình *KeyDemoFrame* minh họa phương thức *KeyListener*. Lớp *KeyDemoFrame* triển khai giao diện *KeyListener*, vì vậy cả 3 phương thức đã giới thiệu được khai báo trong chương trình. Phương thức *addKeyListener* được khai báo trong lớp *Component*, mỗi lớp con của *Component* có thể thông báo cho *KeyListener* các sự kiện bàn phím đã sử dụng.

```
Import java.awt.Color;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JTextArea;

public class KeyDemoFrame extends JFrame implements KeyListener {
    private String line1 = ""; //dòng đầu tiên trong vùng kết quả
    private String line2 = ""; //dòng thứ hai trong vùng kết quả
    private String line3 = ""; //dòng thứ ba trong vùng kết quả
    private JTextArea textArea; //vùng hiển thị kết quả

    // Hàm khởi tạo KeyDemoFrame
    public KeyDemoFrame() {
        super( "Demonstrating Keystroke Events" );
        textArea = new JTextArea( 10, 15 ); // thiết lập JTextArea
```

```

        textArea.setText( "Press any key on the keyboard..." );
        textArea.setEnabled( false ); // disable textarea
        textArea.setDisabledTextColor( Color.BLACK );//thiết lập
màu chữ
        add( textArea ); // thêm vùng hiển thị vào JFrame

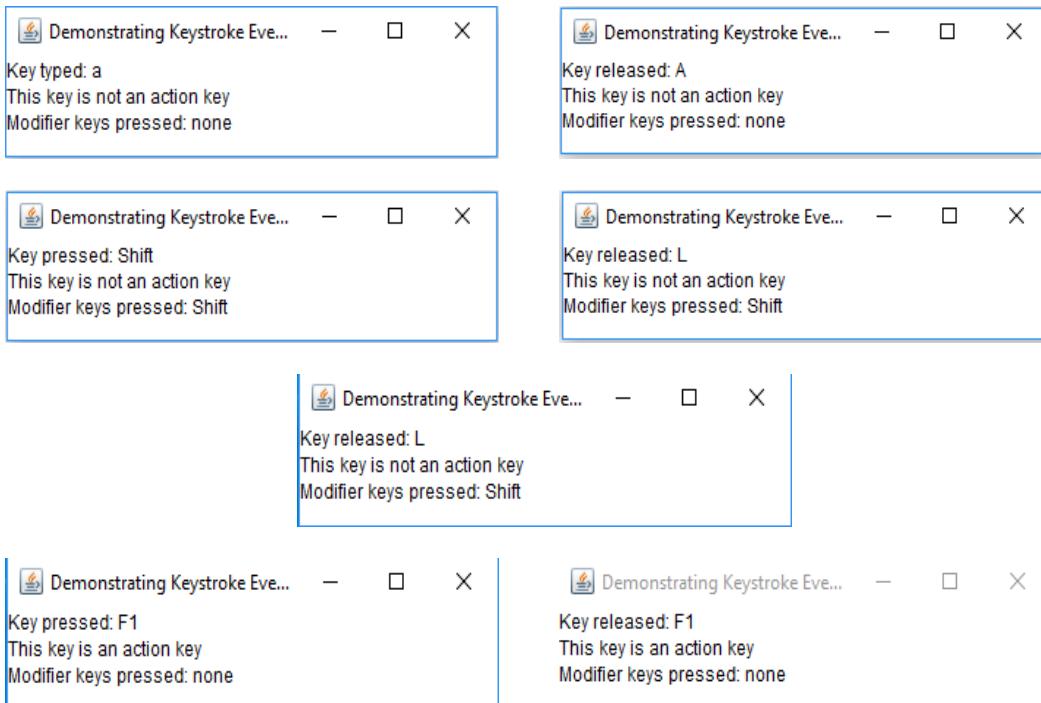
        addKeyListener( this ); //cho phép khung vào xử lý sự kiện
bàn phím
    }
    // xử lý sự kiện nhấn phím bất kỳ
    public void keyPressed( KeyEvent event ){
        line1 = String.format( "Key pressed: %s",
            event.getKeyText( event.getKeyCode() ) ); //hiển thi
kết quả nút nào đã được nhấn
        setLines2and3( event ); // thiết lập số dòng dữ liệu kết
quả hiển thị
    }
    // xử lý sự kiện nhả phím
    public void keyReleased( KeyEvent event ){
        line1 = String.format( "Key released: %s",
            event.getKeyText( event.getKeyCode() ) ); // hiển thi
kết quả nút nào đã được nhả
        setLines2and3( event ); // thiết lập số dòng dữ liệu kết
quả hiển thị
    }

    // xử lý sự kiện nhấn phím chức năng
    public void keyTyped( KeyEvent event ) {
        line1 = String.format( "Key typed: %s", event.getKeyChar()
);
        setLines2and3( event ); // thiết lập số dòng dữ liệu kết
quả hiển thị
    }
    //hàm thiết lập số dòng dữ liệu kết quả hiển thị
    private void setLines2and3( KeyEvent event ){
        line2 = String.format( "This key is %san action key",
            ( event.isActionKey() ? "" : "not " ) );

        String temp = event.getKeyModifiersText(
event.getModifiers() );
        line3 = String.format( "Modifier keys pressed: %s",
            ( temp.equals( "" ) ? "none" : temp ) ); // bộ chỉnh
sửa kết quả

        textArea.setText( String.format( "%s\n%s\n%s\n",
line1, line2, line3 ) );
    }
}

```



Hình 6.19. Kết quả thực thi chương trình KeyDemoFrame.

II. MẪU THIẾT KẾ (DESIGN PATTERNS)

1. Tổng quan về mẫu thiết kế

a) Khái niệm

- Lịch sử hình thành

Vào cuối những năm 70, kiến trúc sư Christopher Alexander đã bắt đầu đưa ra khái niệm về mẫu. Christopher Alexander tuy là một kỹ sư và kiến trúc sư nhưng những công trình của ông lại là nơi bắt nguồn ý tưởng cho cộng đồng lập trình hướng đối tượng xây dựng các mẫu thiết kế; và một số lượng lớn những mẫu thiết kế đã được ra đời từ đây giúp cho công việc thiết kế phần mềm trở nên đơn giản hơn. Kent Beck và Ward Cunningham là một trong số những người đã trình bày tập các mẫu thiết kế trong Hội nghị OOPSLA. James Coplien là một trong số những người tích cực thúc đẩy phát triển nguyên lý về mẫu; chẳng bao lâu sau, cộng đồng mẫu thiết kế bắt đầu phát triển tại OOPSLA, và nó đã tạo ra một môi trường cho các thành viên chia sẻ những ý tưởng, những sự đổi mới về các mẫu. Một diễn đàn quan trọng khác giúp thúc đẩy sự phát triển của phong trào này đó

là diễn đàn Hillside, được thành lập bởi Kent Beck và Grady Booch.

Trong phần này sẽ trình bày về khái niệm mẫu thiết kế, lợi ích của nó và các mẫu thiết kế thông dụng, cách sử dụng các mẫu thiết kế thông dụng này.

- Khái niệm

Mẫu thiết kế là các mô tả về một giải pháp chung cho cùng một vấn đề thường gặp trong nhiều chương trình trong việc thiết kế phần mềm.

Trong bất kỳ hệ thống phần mềm hướng đối tượng nào chúng ta cũng có thể bắt gặp các vấn đề lặp lại và một mẫu thiết kế có thể được xem như một “khuôn mẫu” có sẵn áp dụng được cho nhiều tình huống khác nhau để giải quyết một vấn đề cụ thể. Các mẫu thiết kế là các mô tả không phải là các đoạn mã chương trình, nó có thể sử dụng giống như một hình mẫu để áp dụng vào giải quyết vấn đề.

- Lợi ích của các mẫu thiết kế

Khi lập trình hướng đối tượng, chúng ta có thể nghĩ rằng các đoạn mã của chúng ta đã sử dụng được hết những lợi ích do ngôn ngữ hướng đối tượng mang đến, các đoạn mã được viết ra có thể đủ linh hoạt để thay đổi trong từng hoàn cảnh cụ thể, có thể tái sử dụng được khi gặp một vấn đề tương tự; chúng ta có thể duy trì đoạn mã một cách dễ dàng và bất kỳ sự thay đổi nào trong chương trình cũng sẽ không ảnh hưởng đến đoạn chương trình khác. Nhưng không may thay, những lợi ích này lại phụ thuộc vào kinh nghiệm và khả năng của chính người phát triển phần mềm. Lúc này sử dụng mẫu thiết kế vừa có thể tái sử dụng được với những vấn đề xảy ra phổ biến mà vẫn tận dụng được kinh nghiệm và khả năng của các chuyên gia phát triển phần mềm;

Mẫu thiết kế giúp cho các lập trình viên tái sử dụng được mã lệnh và dễ dàng mở rộng. Nó là tập hợp hơn những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong kỹ thuật phát triển phần mềm; giúp người dùng giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh;

Mẫu thiết kế cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển

phần mềm bằng cách đưa ra các mô hình kiểm thử, mô hình phát triển đã qua kiểm nghiệm;

Dùng lại các mẫu thiết kế giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp, bảo trì về sau;

Cuối cùng, mẫu thiết kế còn giúp cho các lập trình viên có thể hiểu mã lệnh của người khác một cách nhanh chóng;

b) Phân loại mẫu thiết kế

Có thể chia mẫu thiết kế thành các loại như sau:

+ *Các mẫu khởi tạo (Creational patterns)*: Nhóm này cung cấp phương pháp tạo ra các đối tượng một cách linh hoạt, quyết định đối tượng nào được tạo ra tùy thuộc vào trường hợp sử dụng nhất định;

+ *Các mẫu cấu trúc (Structural patterns)*: Nhóm này liên quan đến việc thiết lập, định nghĩa quan hệ giữa các đối tượng;

+ *Các mẫu hành vi (Behavioral patterns)*: Nhóm mẫu thiết kế này trình bày phương pháp thiết kế liên quan đến thực hiện các hành vi của các đối tượng.

Bảng 6.3 liệt kê các mẫu theo từng nhóm:

STT	Tên	Mục đích
Nhóm khởi tạo		
1	Abstract Factory	Cung cấp một Interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định hay xác định lớp cụ thể. Tần suất sử dụng: cao
2	Builder	Chia một công việc khởi tạo phức tạp của một đối tượng ra riêng rẽ từ đó có thể tiến hành khởi tạo đối tượng ở các hoàn cảnh khác nhau. Tần suất sử dụng: trung bình thấp

3	Factory Method	Định nghĩa Interface để tạo ra đối tượng nhưng cho phép lớp con quyết định lớp nào được dùng để sinh ra đối tượng. Factory method cho phép một lớp trì hoãn sự thể hiện một lớp con. Tần suất sử dụng: cao
4	Prototype	Chỉ định ra một đối tượng đặc biệt để khởi tạo, nó sử dụng một thể nghiệm sơ khai rồi sau đó sao chép ra các đối tượng khác từ mẫu đối tượng này. Tần suất sử dụng: trung bình
5	Singleton	Đảm bảo một class chỉ có một thể hiện và cung cấp một điểm truy xuất toàn cục đến nó. Tần suất sử dụng: cao trung bình
Nhóm Structural (nhóm cấu trúc)		
6	Adapter	Thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp do vấn đề tương thích. Tần suất sử dụng: cao trung bình
7	Bridge	Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt, mục đích để cả hai phần (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau. Tần suất sử dụng: trung bình
8	Composite	Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; tất cả các đối tượng trong cấu trúc được thao tác theo một cách thống nhất và được gộp lại, làm tăng khả năng tổng quát hóa trong lập trình và dễ phát triển, nâng cấp, bảo trì. Tần suất sử dụng: cao trung bình
9	Decorator	Gán thêm trách nhiệm cho đối tượng trong lúc

		thực thi. Tần suất sử dụng: trung bình
10	Facade	Cung cấp một interface đồng nhất cho một tập hợp các interface trong một “hệ thống con” (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn. Tần suất sử dụng: cao
11	Flyweight	Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng nhỏ (chẳng hạn paragraph, dòng, cột, ký tự...). Tần suất sử dụng: thấp
12	Proxy	Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy. Tần suất sử dụng: cao trung bình
Nhóm Behavioral (nhóm tương tác)		
13	Chain of Responsibility	Khắc phục việc ghép cặp giữa bộ gửi và bộ nhận thông điệp; các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi này đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cứng giữa phần tử gửi yêu cầu với phần tử nhận và xử lý yêu cầu bằng cách cho phép hơn một đối tượng có cơ hội xử lý yêu cầu. Liên kết các đối tượng nhận yêu cầu thành một dây chuyền rồi chuyển yêu cầu này xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể. Tần suất sử dụng: trung bình thấp
14	Command	Đóng gói yêu cầu vào trong một đối tượng, các

		yêu cầu sẽ được lưu trữ và gửi đi như các đối tượng nhờ. Tần suất sử dụng: cao trung bình
15	Interpreter	Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.Tần suất sử dụng: thấp
16	Iterator	Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, ...) mà không phụ thuộc vào biểu diễn bên trong của các phần tử. Tần suất sử dụng: cao
17	Mediator	Đóng gói cách thức tương tác của một tập hợp các đối tượng. Giảm bớt liên kết và cho phép thay đổi cách thức tương tác giữa các đối tượng. Tần suất sử dụng: trung bình thấp
18	Memento	Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm nguyên tắc đóng gói.Tần suất sử dụng: thấp
19	Observer	Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo. Tần suất sử dụng: cao
20	State	Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong nó thay đổi. đối tượng sẽ xuất hiện để thay đổi các lớp của nó.Tần suất sử dụng: trung bình
21	Strategy	Định nghĩa một họ các thuật toán, đóng gói mỗi thuật toán đó và làm cho chúng có khả năng thay đổi dễ dàng, nó cho phép giải thuật

		tùy biến một cách độc lập tại các Client sử dụng nó Tần suất sử dụng: cao trung bình
22	Template method	Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa. Tần suất sử dụng: trung bình
23	Visitor	Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó. Tần suất sử dụng: thấp

Bảng 6.3. Các mẫu thiết kế cụ thể trong từng nhóm.

Dưới đây giới thiệu về một vài mẫu thiết kế phổ biến, mỗi nhóm mẫu sẽ chọn ra hai mẫu cơ bản có tần suất sử dụng cao theo bảng phân loại ở trên.

2. Các mẫu thiết kế phổ biến

a) *Mẫu Abstract Factory*

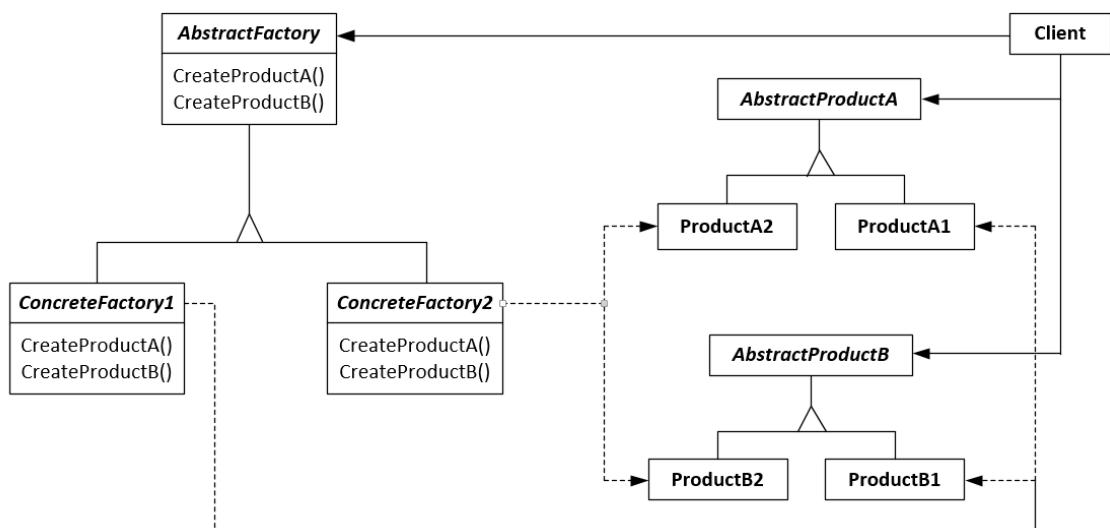
- Ý nghĩa

Là mẫu cung cấp giao diện nhằm tạo ra một họ các đối tượng phụ thuộc/có liên quan mà không cần phải xây dựng các lớp đó. Để hiểu thêm ý nghĩa của mẫu *Abstract Factory* ta xem xét trường hợp cụ thể sau: Trong các hệ điều hành giao diện đồ họa, một bộ công cụ muốn cung cấp một giao diện người dùng dựa trên chuẩn *look-and-feel*, chẳng hạn như chương trình trình diễn tài liệu Microsoft Office PowerPoint. Có rất nhiều kiểu giao diện *look-and-feel* và cả những hành vi giao diện người dùng khác nhau được thể hiện trên ứng dụng như thanh cuộn tài liệu (*scroll bar*), cửa sổ (*window*), nút bấm (*button*), hộp soạn thảo (*editbox*).. Nếu xem chúng là các đối tượng thì chúng ta thấy chúng có một số đặc điểm và hành

vi khá giống nhau về mặt hình thức nhưng lại khác nhau về cách thực hiện. Chẳng hạn đối tượng *button*, *window* và *editbox* có cùng các thuộc tính là chiều rộng, chiều cao, tọa độ... Có các phương thức là *Resize()*, *SetPosition()*... Tuy nhiên các đối tượng này không thể gộp chung vào một lớp được vì theo nguyên lý xây dựng lớp thì các đối tượng thuộc lớp phải có các phương thức hoạt động giống nhau. Trong khi ở đây tuy rằng các đối tượng có cùng giao diện nhưng cách thực hiện các hành vi tương ứng lại hoàn toàn khác nhau.

Vấn đề đặt ra là phải xây dựng một lớp tổng quát, có thể chứa hết được những điểm chung của các đối tượng này để từ đó có thể dễ dàng sử dụng lại, ta gọi lớp này là *WidgetFactory*. Các lớp của các đối tượng *window*, *button*, *editbox* kế thừa từ lớp này. Trong thiết kế hướng đối tượng, xây dựng một mô hình các lớp như thế được tối ưu hoá bằng cách sử dụng mẫu *Abstract Factory* theo như mô hình ở Hình 6.20.

- Mô hình cấu trúc mẫu



Hình 6.20. Mô hình cấu trúc mẫu *Abstract Factory*.

Các thành phần tham gia mô hình như sau:

- + *AbstractFactory* (*WidgetFactory*): khai báo một giao diện trừu tượng
- + *ConcreteFactory* (*MotifWidgetFactory*, *PMWidgetFactory*): cài đặt phương thức để tạo ra các đối tượng

- + AbstractProduct (Window, ScrollBar): khai báo một giao diện cho từng đối tượng được sinh ra
- + ConcreteProduct (MotifWindow, MotifScrollBar): cài đặt giao diện cho AbstractProduct
- + Client: chỉ sử dụng các giao diện được khai báo bởi các lớp AbstractFactory và AbstractProduct.
- Các trường hợp áp dụng mẫu Abstract Factory:
 - + Một hệ thống cần độc lập với cách thức tạo ra các đối tượng và cách thể hiện các đối tượng này
 - + Một hệ thống được cấu hình với một trong số các đối tượng có nhiều ràng buộc khác
 - + Một họ các đối tượng có liên quan nhau được thiết kế để sử dụng cùng nhau, và chúng ta cần ngăn cản sự ràng buộc giữa chúng
 - + Muốn cung cấp một lớp thư viện các đối tượng nhưng không muốn cách cài đặt để tạo nên các đối tượng này bị nhìn thấy, chỉ cho phép nhìn thấy giao diện của chúng mà thôi.

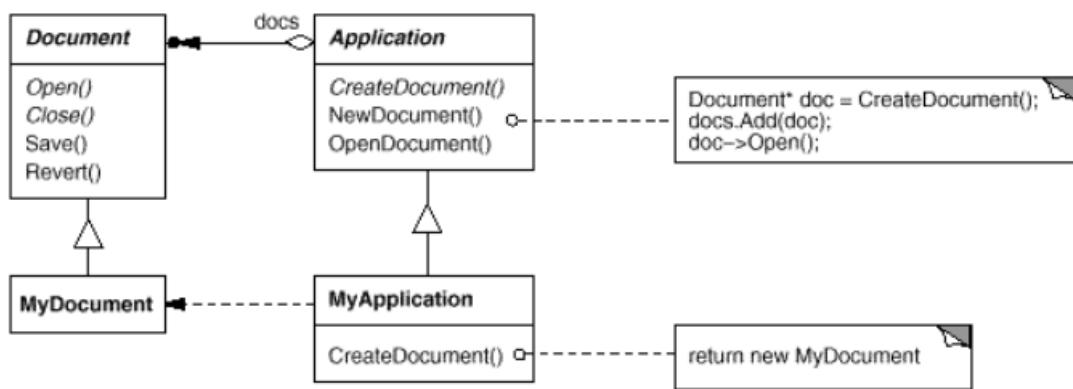
b) Mẫu Factory Method

- Ý nghĩa:

Mẫu *Factory Method* là mẫu định nghĩa một giao diện để tạo ra một đối tượng nhưng cho phép các lớp con quyết định lớp nào được khởi tạo và cho phép một lớp trì hoãn sự thể hiện của các lớp con. Ví dụ: Các *framework* sử dụng các lớp trừu tượng để định nghĩa và duy trì các mối quan hệ giữa các đối tượng. Một *framework* thường đảm nhiệm việc tạo ra những đối tượng hoàn chỉnh. Chúng ta hãy xem xét một *framework* của những ứng dụng có thể biểu diễn nhiều tài liệu cho người dùng. Có hai loại lớp trừu tượng chủ chốt trong *framework* này là lớp Application và lớp Document. Cả hai lớp đều là lớp trừu tượng và các trình khách

phải phân lớp chúng để hiện thực hóa thành các ứng dụng cài đặt cụ thể. Các lớp và phương thức của nó được biểu diễn trong Hình 6.24. Chẳng hạn chúng ta định nghĩa hai lớp DrawingApplication và DrawingDocument. Lớp Application đảm nhiệm việc quản lý các Document và sẽ tạo ra chúng khi có yêu cầu từ người dùng; các yêu cầu đó có thể là các thao tác chọn Open/New trên thanh thực đơn của người dùng.

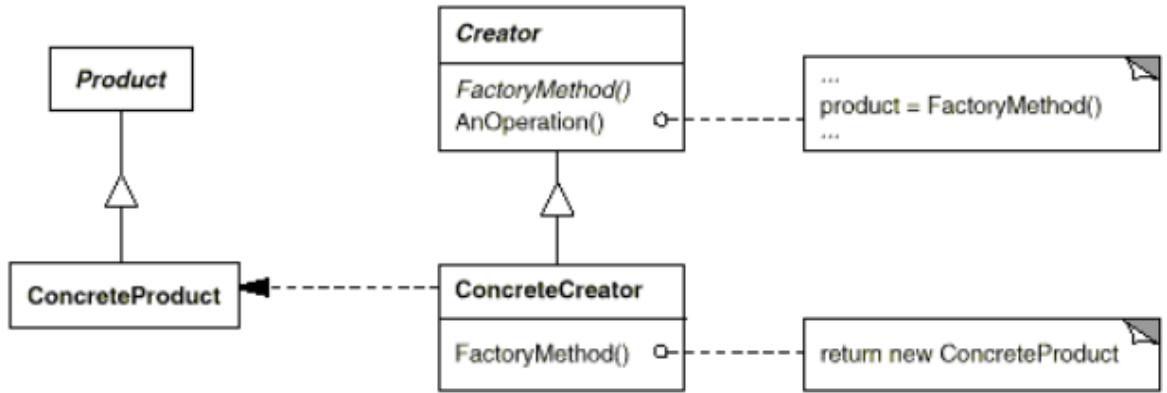
Bởi vì lớp Document thực hiện phân lớp để thể hiện là một ứng dụng cụ thể, nên lớp Application không thể đoán trước được đó là lớp con nào của Document để thể hiện nó, lớp Application chỉ có thể biết khi nào thì một tài liệu mới được tạo ra, và loại tài liệu nào không được tạo ra mà thôi. Sự việc này gây ra một vấn đề: framework phải thể hiện các lớp nhưng nó chỉ biết được các lớp trừu tượng chứ không biết được các thể hiện của chúng. Và mẫu Factory Method sẽ đưa ra giải pháp để khắc phục. Nó đóng gói những gì mà các lớp con của Document tạo ra và đưa gói này ra ngoài *framework*.



Hình 6.21. Các lớp và phương thức xây dựng một trình soạn thảo văn bản.

Các lớp con của Application sẽ định nghĩa lại một phương thức trừu tượng CreateDocument trên Application để trả về lớp con Document thích hợp. Mỗi một lần lớp con của Application được thể hiện thì sau đó nó có thể thể hiện được các Document cụ thể mà không cần biết đến các lớp của các Document này. Chúng ta gọi CreateDocument lúc này là một factory method vì nó đảm nhận việc “tạo ra” một đối tượng.

- Mô hình cấu trúc mẫu



Hình 6.22. Mô hình cấu trúc mẫu *factory method*.

Các thành phần tham gia mô hình như sau:

- + Product (Document): định nghĩa giao diện của các đối tượng mà factory method tạo ra
- + ConcreteProduct (MyDocument): triển khai giao diện Product
- + Creator (Application): khai báo factory method, trả về một đối tượng kiểu Product. Creator cũng có thể định nghĩa một đối tượng ConcreteProduct mặc định. Có thể gọi factory method để tạo một đối tượng kiểu Product.
- + ConcreteCreator (MyApplication): ghi đè phương thức factory method để trả về một thể hiện của ConcreteProduct.
- Các trường hợp áp dụng mẫu Method Factory
 - + Một lớp không thể dự đoán được các lớp của đối tượng mà nó phải tạo
 - + Một lớp muốn các lớp con của nó xác định các đối tượng mà nó tạo ra
 - + Các lớp ủy quyền cho các lớp con hỗ trợ, và muốn định vị lớp con hỗ trợ nào được ủy quyền.

c) Mẫu Adapter

- Ý nghĩa

Mẫu *Adapter* là mẫu biến đổi giao diện của một lớp thành một giao diện khác khi các trình khách yêu cầu, cho phép các lớp được làm việc với nhau mặc dù các giao diện không tương thích. *Adapter* hoạt động giống như là cầu nối giữa 2 giao diện không phù hợp với nhau. Ví dụ: Xem xét một chương trình hỗ trợ vẽ các hình ảnh, sơ đồ từ các đối tượng hình học cơ bản (đường thẳng, đa giác, văn bản,...). Sự trừu tượng của các chương trình này chính là các đối tượng hình học, các đối tượng này có thể chỉnh sửa được. Giao diện của các đối tượng hình học được định nghĩa bởi một lớp trừu tượng đặt tên là Shape. Trình vẽ ảnh định nghĩa một lớp con của Shape cho mỗi loại đối tượng đồ họa: lớp LineShape cho đường thẳng, lớp PolygonShape cho đa giác.

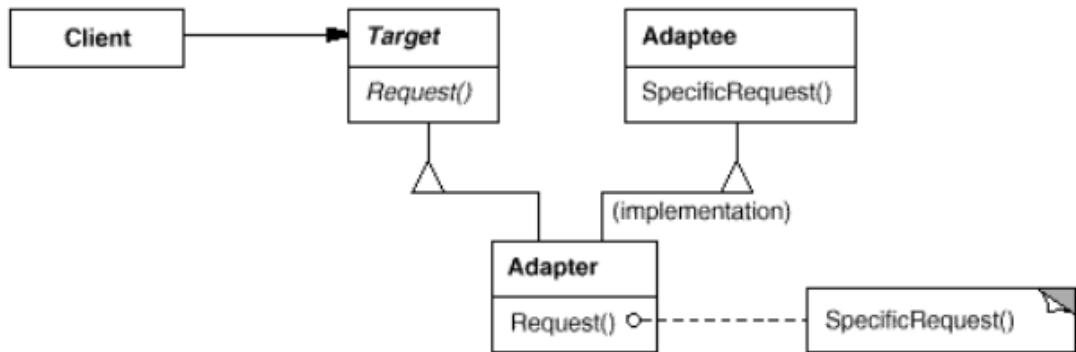
Việc cài đặt các lớp LineShape, PolygonShape khá là dễ dàng bởi vì khả năng chỉnh sửa và vẽ các đối tượng này đã được giới hạn nhưng việc cài đặt một lớp TextShape - dùng để hiển thị và chỉnh sửa các ký tự *text* thì lại khó khăn hơn nhiều bởi vì nó làm ảnh hưởng đến việc quản lý bộ nhớ đệm và cập nhật màn hình trong khi đó bộ giao diện người dùng đang có sẵn đã cung cấp một lớp TextView rất tốt để hiển thị và chỉnh sửa các ký tự rồi. Trường hợp lý tưởng là chúng ta sẽ sử dụng lại lớp TextView này để cài đặt TextShape nhưng mà giao diện người dùng có sẵn này lại không phải là một lớp con của Shape trong chương trình đang xét nên chúng không thể thay thế được cho nhau trong trường hợp này.

Chúng ta có thể lựa chọn giải pháp khắc phục là thay đổi lớp TextView để cho nó phù hợp với Shape nhưng việc có được mã nguồn của TextView là vấn đề khó khăn, kể cả có được thì cũng không nên thực hiện thay đổi TextView của cả một chương trình hệ thống chỉ để cho phù hợp với một chương trình ứng dụng nhỏ. Mà thay vào đó, chúng ta có thể định nghĩa lại TextShape sao cho nó có thể điều chỉnh được giao diện TextView thành giao diện Shape theo một trong 2 cách sau: (1) Kế thừa giao diện của Shape và cách cài đặt của TextView hoặc (2) tạo một thể hiện TextView trong một TextShape và cài đặt TextShape như là giao diện của TextView.

Hai cách tiếp cận này tương ứng là lớp và các đối tượng của mẫu Adapter. Trong trường hợp ví dụ đang xét này chúng ta gọi TextShape chính là một Adapter.

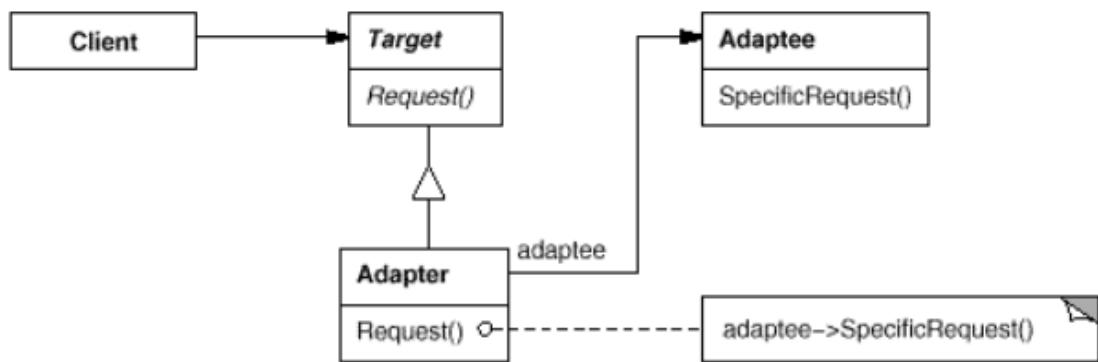
- Mô hình cấu trúc mẫu

+ Mô hình 1: Mẫu điều chỉnh lớp sử dụng đa kế thừa để điều chỉnh một giao diện của lớp này với lớp khác



Hình 6.23. Mô hình cấu trúc mẫu Adapter 1.

+ Mô hình 2: Mẫu điều chỉnh đối tượng phụ thuộc vào thành phần tạo nên đối tượng



Hình 6.24. Mô hình cấu trúc mẫu Adapter 2.

Các thành phần tham gia mô hình

- + Target (Shape): định nghĩa giao diện miền cụ thể mà Client sử dụng
- + Client (DrawingEditor): hợp tác với các đối tượng phù hợp với giao diện của Target

- + Adaptee (TexView): định nghĩa một giao diện đang tồn tại mà cần phải làm biến đổi cho phù hợp
- + Adapter (TextShape): làm tương thích giao diện của Adaptee với giao diện Target
 - Các trường hợp áp dụng mẫu Adapter
 - + Muốn sử dụng một lớp có sẵn nhưng mà giao diện của nó không tương thích với lớp đang xét
 - + Muốn tạo một lớp có khả năng sử dụng lại, có thể kết hợp được với các lớp phát sinh mà chúng ta không biết trước được các giao diện của các lớp phát sinh này
 - + Khi cần sử dụng một số lớp con đang có, nhưng không thể thay đổi các giao diện của từng lớp và lúc này cần điều chỉnh các giao diện này của lớp cha.

d) Mẫu Façade

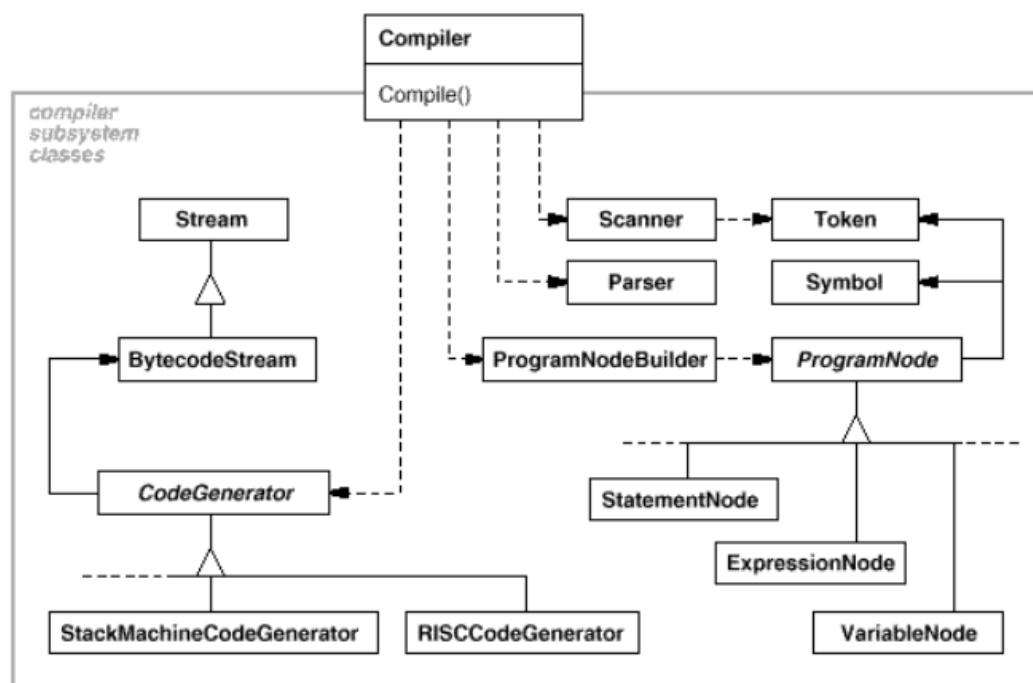
- Ý nghĩa

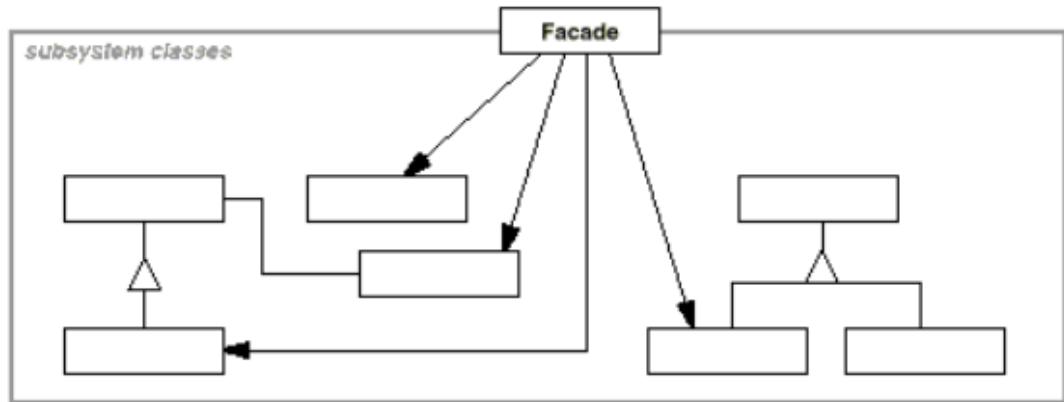
Mẫu *Façade* cung cấp một giao diện thống nhất cho một tập các giao diện trong hệ thống con, nó định nghĩa một giao diện ở mức độ cao hơn, làm cho các hệ thống con dễ sử dụng hơn. Việc tạo nên một hệ thống từ các hệ thống con sẽ làm giảm độ phức tạp xuống. Mục tiêu thiết kế chung là tối thiểu hóa sự phụ thuộc và sự liên kết giữa các hệ thống con. Có một cách để đạt được mục tiêu này đó là sử dụng đối tượng *Façade*, nó cung cấp một giao diện đơn giản, đã được rút gọn bớt. Ví dụ: Xem xét một môi trường lập trình cho phép các ứng dụng truy cập đến các hệ thống con của trình biên dịch, hệ thống con này chứa các lớp như: scanner, parser, ProgramNode, BytecodeStream, và programNodeBuilder. Một vài ứng dụng đặc biệt có thể cần truy cập trực tiếp đến các lớp này. Nhưng hầu hết các ứng dụng của trình biên dịch không quan tâm quá chi tiết về việc một đoạn chương trình được phân rã và sinh ra như thế nào mà chỉ quan tâm là đoạn mã đã được

biên dịch. Bởi vậy, các lớp đối tượng của trình biên dịch tuy rằng nó rất quan trọng nhưng lại có các giao diện ở mức thấp, và chỉ làm cho các nhiệm vụ trở nên phức tạp hơn

Để cung cấp một giao diện ở mức cao hơn, ngăn cản việc truy cập của các ứng dụng đến các lớp của trình biên dịch, thì phải khai báo thêm một lớp Compiler. Lớp này định nghĩa một giao diện thống nhất đến các hàm chức năng của trình biên dịch và đóng vai trò như một vỏ bọc. Compiler sẽ yêu cầu các ứng dụng đơn với những giao diện đơn giản đến các hệ thống con của trình biên dịch, nó sẽ kết nối các lớp này với nhau để cài đặt chức năng trình biên dịch mà không phải giấu chúng hoàn toàn.

- Mô hình cấu trúc mẫu





Hình 6.25. Mô hình cấu trúc mẫu Façade.

Các thành phần tham gia mô hình gồm có:

- + Façade (Compiler): biết được các lớp con nào chịu trách nhiệm trả lời cho một yêu cầu đến; đại diện cho các yêu cầu của máy khách đến các đối tượng hệ thống con thích hợp

- + Subsystem classes (Scanner, Parser, ProgramNode,...): cài đặt các chức năng của hệ thống con; xử lý các công việc do Façade phân công; không biết và không tham chiếu đến Façade.

- Các trường hợp áp dụng mẫu Façade:

- + Muốn cung cấp một giao diện đơn giản đến một hệ thống con phức tạp
- + Có nhiều sự ràng buộc giữa các ứng dụng và các lớp của các ứng dụng này là trừu tượng

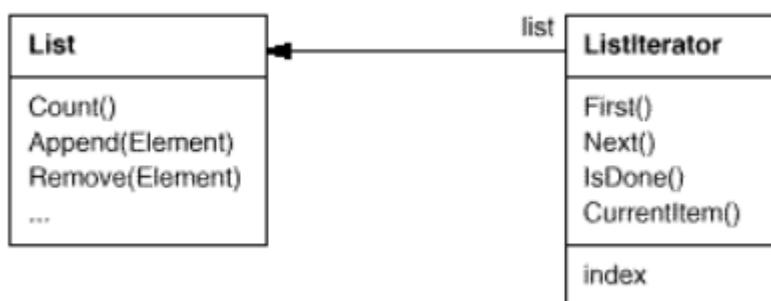
Khi các hệ thống con sắp xếp thành các lớp thì việc sử dụng façade sẽ định nghĩa một con trỏ đến từng từng hệ thống con; nếu các hệ thống con này là phụ thuộc nhau thì có thể đơn giản hóa sự phụ thuộc này bằng cách cho các hệ thống con giao diện với nhau thông qua các façade.

e) Mẫu Iterator

- Ý nghĩa

Mẫu Iterator cung cấp cách truy cập đến các thành phần của một tập hợp các

đối tượng liên kết nhau và duyệt ác thành phần trong tập hợp này mà không cần quan tâm đến cách thức biểu diễn bên trong. Một đối tượng liên kết như là một danh sách sẽ cho chúng ta cách để truy cập đến từng phần tử của nó mà không cần phải thể hiện ra bên ngoài cấu trúc của nó. Mẫu Iterator có trách nhiệm truy cập và vượt qua đối tượng danh sách và đặt nó vào đối tượng iterator. Lớp Iterator định nghĩa một giao diện để truy cập vào các thành phần của một danh sách; một đối tượng iterator dõi theo thành phần hiện tại và nó sẽ biết những thành phần nào đã được đi qua. Ví dụ: một lớp List sẽ gọi tới một lớp ListIterator theo mối quan hệ như Hình 6.31.



Hình 6.26. Mối quan hệ giữa các danh sách.

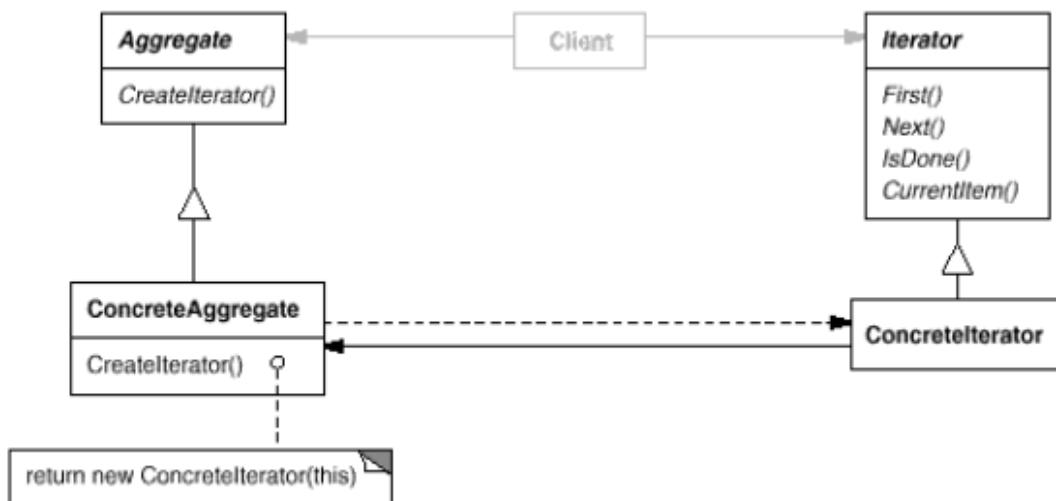
Mỗi lần có thể hiện *ListIterator*, chúng ta có thể truy nhập vào các thành phần của danh sách liên tục. Phương thức *CurrentItem* trả về giá trị hiện tại trong danh sách. Hàm *First()* là để khởi tạo giá trị đầu tiên trong danh sách; *Next()* là phát triển từ giá trị hiện tại sang giá trị kế tiếp, *isDone()* là kiểm tra rằng danh sách đã được duyệt đến giá trị cuối cùng chưa và hoàn thành việc duyệt danh sách.

Chúng ta định nghĩa một lớp *AbstractList* cung cấp một giao diện chung cho các danh sách liên quan nhau. Cũng giống như vậy, chúng ta cần một lớp Iterator trừu tượng định nghĩa một giao diện lặp lại chung. Như vậy, kỹ thuật lặp trở nên độc lập với các lớp liên kết.

Vấn đề tồn tại ở đây là làm thế nào để tạo ra vòng lặp. Chúng ta muốn viết mã chương trình độc lập với các lớp con của List thì công việc không thể đơn giản là khởi tạo một lớp đặc biệt mà thay vào đó chúng ta còn phải tạo ra các đối tượng

danh sách chịu trách nhiệm cho việc tạo ra vòng lặp tương ứng cho chúng. Điều này cần có một hàm như *CreateIterator* xuyên suốt khi có yêu cầu một đối tượng lặp. *CreateIterator* là một ví dụ của mẫu *Factory Method*, chúng ta sử dụng ở đây để cho phép một máy khách yêu cầu một đối tượng danh sách với vòng lặp thích hợp.

- Mô hình cấu trúc mẫu như hình 6.32 dưới đây:



Hình 6.27. Mô hình cấu trúc mẫu Iterator.

Các thành phần tham gia mô hình gồm có:

- + Iterator: định nghĩa một giao diện để truy cập và duyệt các phần tử
- + ConcreteIterator: cài đặt giao diện iterator, theo dõi vị trí hiện tại trong quá trình duyệt danh sách
- + Aggregate: định nghĩa một giao diện để tạo một đối tượng Iterator
- + ConcreteAggregate: cài đặt giao diện khởi tạo Iterator để trả về một thẻ hiện cho ConcreteIterator.

- Các trường hợp áp dụng mẫu Iterator:

- + Truy cập đến các nội dung của một đối tượng liên kết mà không thể hiện những hành vi có bên trong đối tượng
- + Hỗ trợ việc duyệt nhiều lần cho các đối tượng liên kết

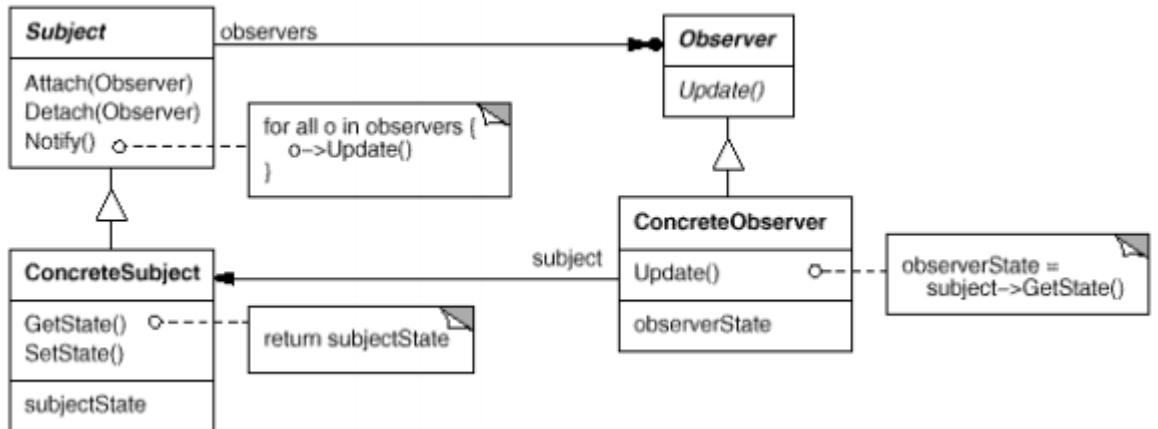
- + Cung cấp một giao diện đồng bộ cho việc duyệt các cấu trúc liên kết khác nhau

g) Mẫu Observer

- Ý nghĩa

Mẫu *Observer* định nghĩa một phụ thuộc một – nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái, tất cả những đối tượng phụ thuộc nó sẽ được cảnh báo và cập nhật tự động. Một tác dụng phụ phổ biến của việc phân chia một hệ thống thành một tập các lớp là cần phải duy trì sự nhất quán giữa các đối tượng với nhau. Nếu như tạo ra sự nhất quán này từ việc kết hợp chặt chẽ các lớp lại với nhau thì chúng ta sẽ mất đi tính tái sử dụng nó. Ví dụ: nhiều bộ công cụ giao diện người dùng chia các khía cạnh hiển thị giao diện người dùng từ dữ liệu ứng dụng lớp dưới; các lớp định nghĩa dữ liệu ứng dụng và các thể hiện có thể sử dụng lại một cách độc lập và cũng có thể kết hợp với nhau. Cả một bảng tính và một biểu đồ dạng cột cùng miêu tả thông tin được lấy từ cùng một dữ liệu ứng dụng giống nhau nhưng cách thể hiện lại khác nhau. Bảng tính và biểu đồ là các đối tượng độc lập nên người dùng có thể chỉ sử dụng 1 trong 2 để biểu diễn dữ liệu. Nhưng nếu người dùng thay đổi thông tin trong bảng tính thì biểu đồ cột cũng phản ánh sự thay đổi đó ngay lập tức. Hành vi này chỉ ra rằng bảng tính và biểu đồ cùng phụ thuộc vào đối tượng dữ liệu, bởi vậy nên có sự cảnh báo nếu như có bất kỳ sự thay đổi nào về trạng thái của đối tượng. Không có lý do gì để hạn chế số lượng các đối tượng phụ thuộc nhau ở con số hai. Và mẫu *Observer* mô tả cách thiết lập nên mối quan hệ này. Các đối tượng chính trong mẫu *Observer* là chủ thể (subject) và quan sát (observer). Một chủ thể có thể có nhiều quan sát phụ thuộc nó. Tất cả các quan sát được thông báo bất kỳ khi nào chủ thể của chúng chuẩn bị có sự thay đổi về trạng thái. Cùng với đó, mỗi một quan sát sẽ truy vấn chủ thể để đồng bộ hóa trạng thái của nó với trạng thái của chủ thể.

- Mô hình cấu trúc mẫu



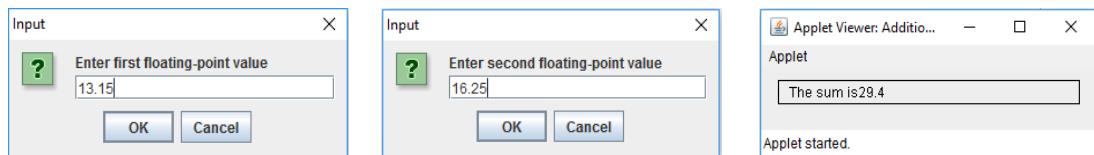
Hình 6.28. Mô hình cấu trúc mẫu Observer.

Các thành phần tham gia mô hình gồm có:

- + Subject: biết được các quan sát của nó; cung cấp một giao diện cho việc gắn và tách các đối tượng quan sát
- + Observer: định nghĩa một giao diện cập nhật cho các đối tượng sẽ được cảnh báo về sự thay đổi của một chủ thể (subject)
- + ConcreteSubject: lưu trữ trạng thái của đối tượng quan tâm; gửi thông báo đến các quan sát khi trạng thái thay đổi
- + ConcreteObserver: duy trì một tham chiếu đến đối tượng ConcreteSubject; lưu trữ trạng thái nhất quán với chủ thể; cài đặt giao diện cập nhật cho các quan sát để duy trì sự nhất quán giữa quan sát với chủ thể của nó.
- Các trường hợp áp dụng mẫu Observer
 - + Khi một lớp trùu tượng có hai thành phần, thành phần này phụ thuộc vào thành phần kia.
 - + Khi một sự thay đổi đối với 1 đối tượng yêu cầu đối tượng còn lại cũng phải thay đổi theo, và người dùng không biết có bao nhiêu đối tượng bị thay đổi.

Câu hỏi và bài tập cuối chương

Câu 1. Viết chương trình ứng dụng applet tính tổng cho hai số thực. Có thể tham khảo kết quả đầu ra như hình dưới đây:

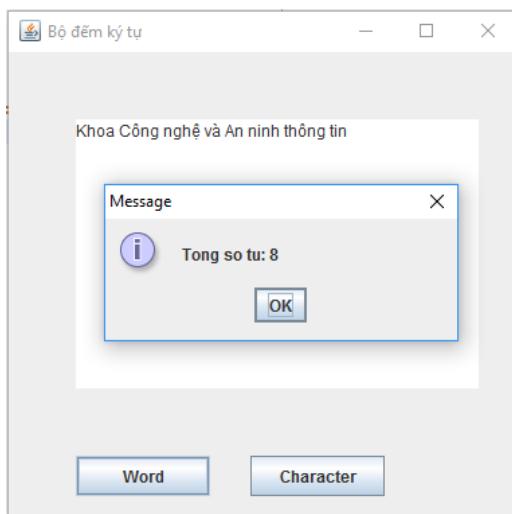


Câu 2. Viết một applet có dạng sau:



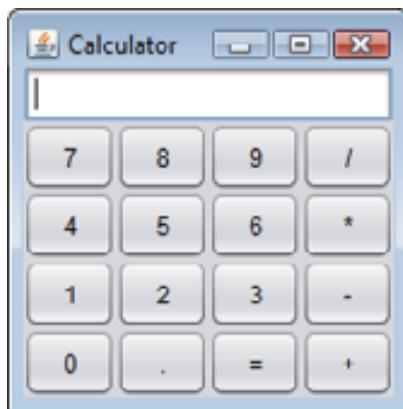
Khi người dùng nhập mật khẩu vào hộp thoại và bấm nút "Kiem tra mat khau", nếu mật khẩu nhập vào là "khoaToanTin" thì sẽ hiện lên thông báo "Da nhap dung mat khau", ngược lại sẽ đưa ra thông báo: "Mật khẩu nhập sai, hãy nhập lại".

Câu 3. Xây dựng bộ đếm ký tự WordCount có giao diện kết quả như sau:



Câu 4. Thực hiện các yêu cầu dưới đây:

- a. Tạo giao diện máy tính đơn giản như sau



- b. Hãy viết các phép tính: +, - , *, / cho giao diện ở ý a để thành một ứng dụng hoàn chỉnh.

Câu 5. Thiết kế giao diện cho nhập mã sinh viên, họ tên, chọn mã lớp, nhập điểm lý thuyết, nhập điểm thực hành.

Yêu cầu xử lý:

- Nút “Kết quả”, ghi điểm trung bình và kết quả của sinh viên đăng nhập vào các JTextFields. Chú ý: ô nhập điểm phải là số

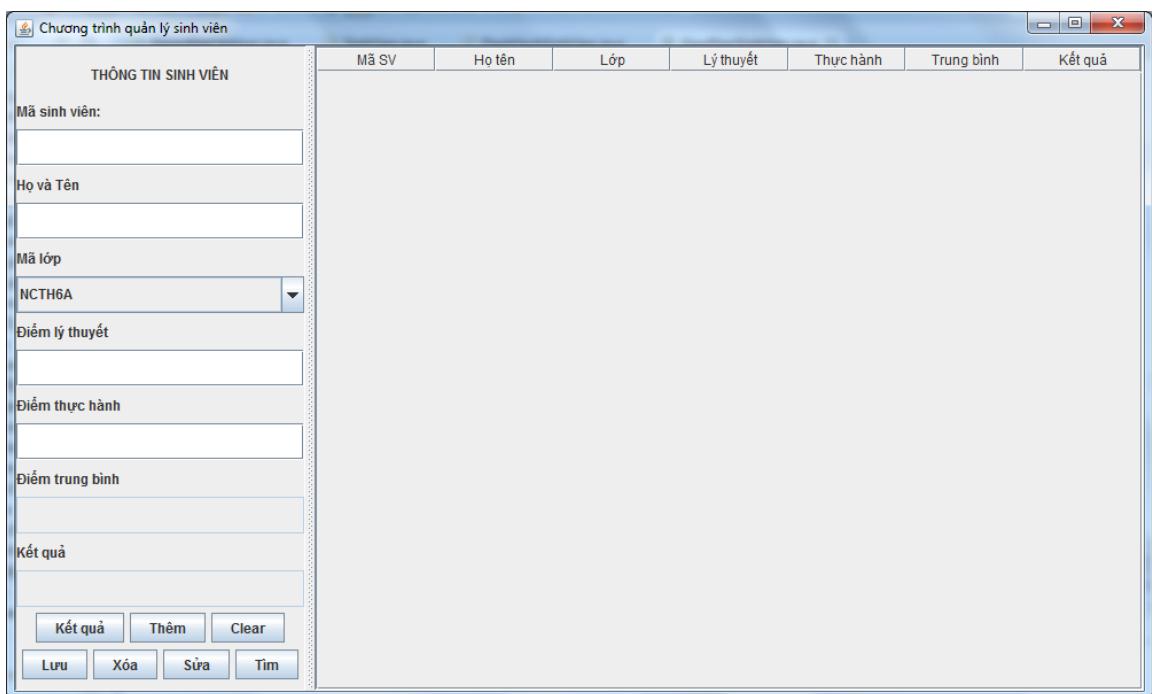
- Nút “Thêm”, thêm một sinh viên vào bảng với các cột như hình. Chú ý không được thêm khi: không nhập đủ dữ liệu, điểm không phải là số, trùng mã sinh viên

- Nút “Clear”: xóa rỗng nội dung trong các JTextFields và JComboBox, mã lớp được thiết lập lại như ban đầu

- Khi chọn 1 dòng trên bảng thì hiện thông tin sinh viên đó lên các ô nhập liệu

- Nút “Lưu”, lưu danh sách sinh viên vào file. Lưu ý: khi chương trình bắt đầu thực hiện, nếu tồn tại file dữ liệu đã lưu trước đó, thì nạp danh sách sinh viên lên trên bảng

- Nút “Tìm”: cho phép người sử dụng nhập mã sinh viên cần tìm vào một JOptionPane. Nếu tìm thấy, hỏi người sử dụng trước khi xóa. Ngược lại, thông báo không tồn tại trong một JOptionPane
- Nút “Xóa”: cho phép người sử dụng nhập mã sinh viên cần xóa vào một JOptionPane. Nếu tìm thấy, hỏi người sử dụng trước khi xóa. Ngược lại, thông báo không tồn tại trong một JOptionPane
- Nút “Sửa”: cho phép người sử dụng sửa thông tin sinh viên. Lưu ý: không được sửa mã số sinh viên



Câu 6. Phòng Quản lý học viên cần quản lý các thông tin 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 khóa học.

Câu 7. Viết chương trình thực hiện độc lập có các menu sau:

- | |
|--|
| a. Nhập thông tin của sinh viên mới |
| b. Tìm theo số báo danh |
| c. Hiển thị tất cả các danh sách các sinh viên |
| d. Thoát khỏi hệ thống |

Người dùng có thể chọn một trong các mục nêu trên. Nếu chọn a. thì người

sử dụng có thể nhập vào một sinh viên mới, chọn b. để tìm theo số báo danh,...

Câu 8. Viết chương trình ứng dụng applet thực hiện các nội dung của câu 6.

Câu 9. 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*

BÀI TẬP TỔNG HỢP

Câu 1. Xây dựng chương trình QLCM dùng để Quản lý, Tra cứu, Tổng hợp toàn bộ hồ sơ công dân đăng ký cấp CMND dành cho các cơ quan công an cấp huyện, tỉnh đăng ký và cấp giấy CMND cho công dân. Các chức năng chính như sau:

- Cho phép các cơ quan quản lý thực hiện Quản lý chứng minh và hoạt động đăng ký...theo các tiêu thức; Thống kê hồ sơ cấp CMND theo thời gian, theo địa giới hành chính; Thống kê hồ sơ theo các hình thức: cấp mới, đổi, cấp lại hoặc thống kê kết hợp tất cả các tiêu thức trên;

- Cho phép thực hiện các yêu cầu tra cứu. Cán bộ quản lý có thể Tra cứu hồ sơ theo số CMND hoặc tra cứu theo từng chỉ tiêu thông tin như: họ tên, ngày sinh, quê quán, nơi thường trú, hoặc kết hợp tất cả các tiêu thức trên.

- Khi muốn in các sổ sách, báo cáo về CMND phần Tổng hợp kết xuất ra các dữ liệu cần thiết như danh sách cấp CMND, báo cáo tình hình hoạt động cấp CMND.

Câu 2. Xây dựng chương trình quản lý các án phẩm cho thư viện gồm sách, tạp chí và kỷ yếu;

- Sách cần quản lý: tên sách, tác giả, số xuất bản, số lượng, thời gian xuất bản, đơn giá.

- Tạp chí gồm: tên tạp chí, loại tạp chí (ví dụ tuần san, quý..), số xuất bản, số lượng, thời gian xuất bản, đơn giá.

- Kỷ yếu gồm: tên kỷ yếu, loại Hội nghị, năm Hội nghị, danh sách tên bài đăng và tác giả, số xuất bản, thời gian xuất bản.

Chương trình có các chức năng như sau:

1. Nhập thông tin cho từng án phẩm.
2. In danh sách án phẩm tăng dần theo số lượng phát hành.

3. Tìm kiếm theo loại án phảm, tên án phảm.

4. Xóa thông tin án phảm.

Câu 3. Xây dựng chương trình quản lý xe ô tô cho một đơn vị gồm các thông tin xe như sau: biển số xe, tên xe, năm được cấp, trạng thái xe (bận, rỗi, sửa chữa..) tên lái xe, liên hệ; Xe con cần biết thêm thông tin số chỗ ngồi, xe tải cần thông tin tải trọng. Chương trình có các chức năng như sau:

1. Nhập thông tin xe
2. Thống kê danh sách xe
3. Tìm kiếm thông tin xe theo biển số, tên lái xe hoặc trạng thái xe.
4. Xóa thông tin xe.

Câu 4. Xây dựng chương trình quản lý nhân khẩu với các chức năng đăng ký hộ khẩu hoặc đăng ký thường trú cho người dân theo bản khai nhân khẩu. Công việc của quản lý hộ khẩu gồm:

- Quản lý hộ gia đình thông qua sổ hộ khẩu theo mẫu quy định;
- Quản lý thường trú thông qua sổ hộ khẩu, hộ gia đình, người nào thuộc hộ gia đình nào được ghi thông tin trong sổ hộ khẩu của hộ gia đình đó;
- Quản lý tạm trú: Khi có người đến xin tạm trú cần phải xuất trình CMND hoặc giấy tạm vắng của địa phương nơi cư trú;
- Quản lý tạm vắng theo mẫu phiếu khai báo tạm vắng.

DANH MỤC TÀI LIỆU THAM KHẢO

- [1]. Đoàn Văn Ban, *Lập trình hướng đối tượng với JAVA*, NXB Khoa học và kỹ thuật, 2006.
- [2]. H. M. Deitel, P. J. Deitel, *Java How to Program* (sách điện tử), 11th Edition, Prentice Hall, 2017.
- [3]. Cay S. Horstmann, Gary Cornell, *Core Java™ 2 Volume I - Fundamentals* (sách điện tử), 10th edition, Prentice Hall, 2016.
- [4]. Rohit Joshi, *Java Design Pattern*, Exelixis Media P.C., 2015

MỤC LỤC

LỜI NÓI ĐẦU	9
Chương 1. GIỚI THIỆU LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ NGÔN NGỮ JAVA	11
I. GIỚI THIỆU LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG	11
1. Các cách tiếp cận trong lập trình.....	11
a) <i>Lập trình tuyến tính</i>	11
b) <i>Lập trình hướng cấu trúc.....</i>	11
c) <i>Lập trình hướng đối tượng.....</i>	11
2. Các khái niệm và tính chất cơ bản của lập trình hướng đối tượng .	12
a) <i>Các khái niệm cơ bản</i>	12
b) <i>Các tính chất cơ bản của lập trình hướng đối tượng</i>	13
II. NGÔN NGỮ LẬP TRÌNH JAVA	16
1. Lịch sử, kiến trúc và đặc trưng của ngôn ngữ Java	16
a) <i>Lịch sử Java</i>	16
b) <i>Kiến trúc, môi trường và đặc trưng Java</i>	17
2. Các ứng dụng Java và công cụ lập trình Java	21
a) <i>Ứng dụng Java</i>	21
b) <i>Công cụ lập trình Java</i>	21
3. Cấu trúc một chương trình Java	23
4. Các phần tử cơ sở.....	35
a) <i>Định danh</i>	35
b) <i>Các từ khóa.....</i>	37
c) <i>Chú thích.....</i>	40

5. Các kiểu dữ liệu, biến và hằng	43
a) Các kiểu dữ liệu	43
b) Biến	46
c) Hằng.....	49
6. Các toán tử	50
a) Các toán tử cơ bản.....	50
b) Thứ tự ưu tiên của các toán tử và quy tắc kết hợp	55
7. Nhập, xuất dữ liệu và các cú pháp điều khiển cơ bản.....	57
a) Nhập dữ liệu từ bàn phím và xuất ra màn hình.....	57
b) Các cú pháp điều khiển cơ bản.....	58
8. Mảng và kiểu liệt kê enum.....	69
a) Mảng	69
b) Kiểu liệt kê enum	73
Câu hỏi và bài tập cuối chương.....	76
Chương 2. LỚP VÀ ĐỐI TƯỢNG	79
I. TRÙU TƯỢNG HÓA ĐỐI TƯỢNG VÀ ĐÓNG GÓI DỮ LIỆU VÀO LỚP	79
1. Trùu tượng hóa đối tượng	79
2. Đóng gói dữ liệu vào lớp	82
II. KHAI BÁO LỚP, TẠO VÀ SỬ DỤNG ĐỐI TƯỢNG.....	86
1. Khai báo lớp và tạo đối tượng.....	86
a) Khai báo lớp	88
b) Tạo đối tượng	93
2. Sử dụng đối tượng.....	94

<i>a) Tham chiếu</i>	94
<i>b) Kiểm soát truy nhập.....</i>	96
<i>c) Dữ liệu kiểu hằng, kiểu tĩnh.....</i>	116
Câu hỏi và bài tập cuối chương.....	123
Chương 3. THÙA KẾ VÀ ĐA HÌNH	126
I. THÙA KẾ.....	126
1. Giới thiệu về thừa kế.....	126
<i>a) Vấn đề sử dụng lại trong lập trình</i>	126
<i>b) Các thành phần trong quan hệ thừa kế</i>	130
<i>c) Phân tích bài toán theo mối quan hệ thừa kế</i>	130
2. Biểu diễn quan hệ thừa kế.....	132
<i>a) Sử dụng sơ đồ quan hệ đối tượng ORD.....</i>	132
<i>b) Cây thừa kế</i>	137
3. Cài đặt thừa kế	138
<i>a) Cài đặt các lớp con.....</i>	140
<i>b) Hiện tượng ẩn danh</i>	151
<i>c) Sự chuyển kiểu đối tượng với upcasting và downcasting</i>	153
II. ĐA HÌNH	156
1. Các loại đa hình.....	156
2. Liên kết phương thức	158
<i>a) Hoạt động của overloading và overriding</i>	158
<i>b) Vai trò của liên kết phương thức đối với đa hình.....</i>	166
<i>c) Phương thức riêng</i>	168

3. Cài đặt đa hình	168
a) Lớp trừu tượng - abstract class	168
b) Phương thức trừu tượng - abstract method.....	169
III. MỞ RỘNG ĐA THỪA KẾ BẰNG GIAO DIỆN	172
1. Khái niệm và cú pháp khai báo giao diện.....	172
a) Khái niệm.....	172
b) Cú pháp khai báo.....	172
2. Phát triển giao diện	173
a) Lớp trừu tượng cài đặt giao diện.....	173
b) Cài đặt nhiều giao diện thay vì thừa kế.....	175
c) Mở rộng lớp trừu tượng và giao diện	178
d) Các ứng dụng khác của giao diện	179
Câu hỏi và bài tập cuối chương.....	181
Chương 4. CÁC LUỒNG VÀO RA VÀ XỬ LÝ NGOẠI LỆ	184
I. CÁC LUỒNG VÀO/RA.....	184
1. Luồng và phân loại luồng	184
a) Khái niệm luồng.....	184
b) Phân loại luồng.....	185
2. Thao tác với luồng	188
a) Nguyên tắc chung	188
b) Thao tác với một số luồng trong Java API.....	188
3. Thao tác chuỗi hóa đối tượng.....	198
II. XỬ LÝ NGOẠI LỆ	200

1. Ngoại lệ	200
a) <i>Khái niệm</i>	200
b) <i>Phân loại</i>	201
c) <i>Ngoại lệ do người dùng tự định nghĩa (Custom Exception)</i>	203
2. Xử lý ngoại lệ.....	204
a) <i>Mô hình xử lý ngoại lệ</i>	205
b) <i>Xử lý ngoại lệ sử dụng khói try...catch</i>	206
c) <i>Xử lý ngoại lệ sử dụng nhiều khói catch</i>	209
d) <i>Xử lý ngoại lệ sử dụng khói try...catch...finally</i>	210
e) <i>Xử lý ngoại lệ sử dụng try-with-resource</i>	212
f) <i>Xử lý ngoại lệ sử dụng Nested try</i>	215
g) <i>Xử lý ngoại lệ sử dụng từ khóa throws và throw</i>	217
Câu hỏi và bài tập cuối chương.....	220
Chương 5. LỚP CƠ SỞ VÀ LẬP TRÌNH TỔNG QUÁT	224
I. MỘT SỐ LỚP CƠ SỞ	224
1. Giới thiệu chung.....	224
2. Lớp String	225
a) <i>Giới thiệu lớp String</i>	225
b) <i>Sử dụng lớp String</i>	233
3. Lớp Math.....	238
a) <i>Giới thiệu lớp Math</i>	238
b) <i>Sử dụng lớp Math</i>	242
4. Các collection.....	243

<i>a) Giới thiệu các collection.....</i>	243
<i>b) Các giao diện - Interface của collection</i>	246
<i>c) Các lớp collection trong Java.....</i>	250
<i>d) Một số phương thức với collection.....</i>	262
<i>f) Sử dụng đối tượng Iterator với các collection</i>	267
II. LẬP TRÌNH TỔNG QUÁT	270
1. Giới thiệu về lập trình tổng quát	270
<i>a) Khái niệm lập trình tổng quát.....</i>	270
<i>b) Lợi ích của lập trình tổng quát.....</i>	272
2. Lớp tổng quát	273
<i>a) Tổng quát hóa một lớp.....</i>	273
<i>b) Các kiểu dữ liệu chưa xác định – Raw types.....</i>	275
3. Phương thức tổng quát	276
4. Các ký tự đại diện generic – wildcards	278
<i>a) Kí tự đại diện <?>.....</i>	280
<i>b) Kí tự đại diện <? extends type></i>	280
<i>c) Kí tự đại diện <? super type></i>	281
5. Ứng dụng của lập trình tổng quát	281
6. Hạn chế của lập trình tổng quát	283
Câu hỏi và bài tập cuối chương.....	284
Chương 6. GIỚI THIỆU LẬP TRÌNH GIAO DIỆN VÀ MẪU THIẾT KẾ	286
I. APPLET VÀ LẬP TRÌNH GIAO DIỆN GUI	286
1. Applet.....	286

<i>a) Giới thiệu về Applet</i>	286
<i>b) Sử dụng applet</i>	290
2. Lập trình giao diện GUI.....	293
<i>a) Giới thiệu</i>	293
<i>b) Khung chứa trong lập trình GUI</i>	296
<i>c) Các thành phần giao diện GUI cơ bản</i>	299
<i>d) Quản lý bộ cục</i>	310
<i>e) Xử lý các sự kiện</i>	314
II. MẪU THIẾT KẾ (DESIGN PATTERNS)	322
1. Tổng quan về mẫu thiết kế.....	322
<i>a) Khái niệm</i>	322
<i>b) Phân loại mẫu thiết kế</i>	324
2. Các mẫu thiết kế phổ biến.....	328
<i>a) Mẫu Abstract Factory</i>	328
<i>b) Mẫu Factory Method</i>	330
<i>c) Mẫu Adapter</i>	332
<i>d) Mẫu Façade</i>	335
<i>e) Mẫu Iterator</i>	337
<i>g) Mẫu Observer</i>	340
Câu hỏi và bài tập cuối chương.....	342
BÀI TẬP TỔNG HỢP	346

