Bắt đầu với Java

Nội dung

- Lịch sử của Java
- Các đặc trưng cơ bản
- Java applications và Java applets
- Tạo ứng dụng Java đơn giản

Lịch sử hình thành

- 1991: được Sun Microsystems phát triển nhằm mục đích viết phần mềm điều khiển (phần mềm nhúng) cho các sản phẩm gia dụng
  - lúc đầu được đặt tên là Oak
- 1995: được phổ cập với sự phát triển mạnh mẽ của Internet
  - thị trường phần mềm nhúng không phát triển mạnh
  - WWW bùng nổ (1993~)
- Hiện nay, được chấp nhận rộng rãi với tư cách là một ngôn ngữ (công nghệ) đa dụng
  - khả chuyển, an toàn
  - hướng đối tượng, hướng thành phần

Java là một công nghệ

Java bao gồm

- Ngôn ngữ lập trình
- Môi trường phát triển
- Môi trường thực thi và triển khai

Mục tiêu của Java

- Ngôn ngữ dễ dùng
  - Khắc phục nhiều nhược điểm của các ngôn ngữ trước đó
  - Hướng đối tượng
  - Sáng sủa
- Môi trường thông dịch
  - Tăng tính khả chuyển
  - An toàn

Mục tiêu của Java

- Cho phép chạy nhiều tiến trình (threads)
- Nạp các lớp (classes) động vào thời điểm cần thiết từ nhiều nguồn khác nhau
  - Cho phép thay đổi động phần mềm trong khi hoạt động
- Tăng độ an toàn

Biên dịch và thông dịch

- Chương trình nguồn được biên dịch sang mã đích (bytecode)
- Mã đích (bytecode) được thực thi trong môi trường thông dịch (máy ảo)

Các dạng ứng dụng của Java

- Desktop applications - J2SE
  - Java Applications: ứng dụng Java thông thường trên desktop
  - Java Applets: ứng dụng nhúng hoạt động trong trình duyệt web
- Server applications - J2EE
  - JSP và Servlets
- Mobile (embedded) applications – J2ME

Đặc trưng của Java

- JVM – máy ảo Java
- Cơ chế giải phóng bộ nhớ tự động
- Bảo mật chương trình

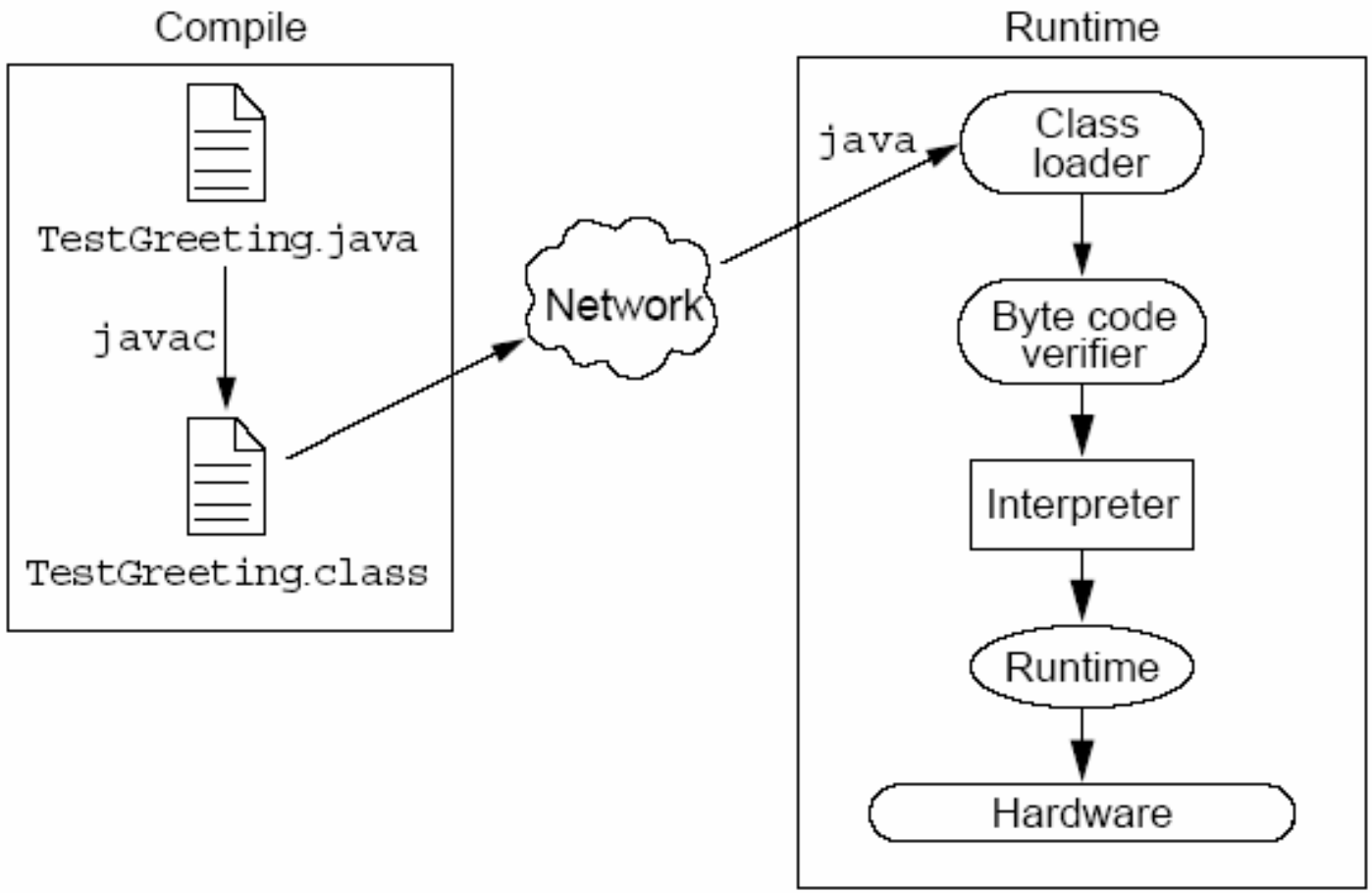
JVM - Máy ảo Java

- Máy ảo phụ thuộc vào platform (phần cứng, OS)
- Cung cấp môi trường thực thi cho chương trình Java (độc lập với platform)
- Máy ảo đảm bảo an toàn cho hệ thống
- Máy ảo thông thường được cung cấp dưới dạng phần mềm
  - JRE - Java Runtime Environment
- Java platform: JVM + APIs

Giải phóng bộ nhớ
(Garbage Collection)

- Java cung cấp một tiến trình mức hệ thống để theo dõi việc cấp phát bộ nhớ
- Garbage Collection
  - Đánh dấu và giải phóng các vùng nhớ không còn được sử dụng
  - Được tiến hành tự động
  - Cơ chế hoạt động phụ thuộc vào các phiên bản máy ảo

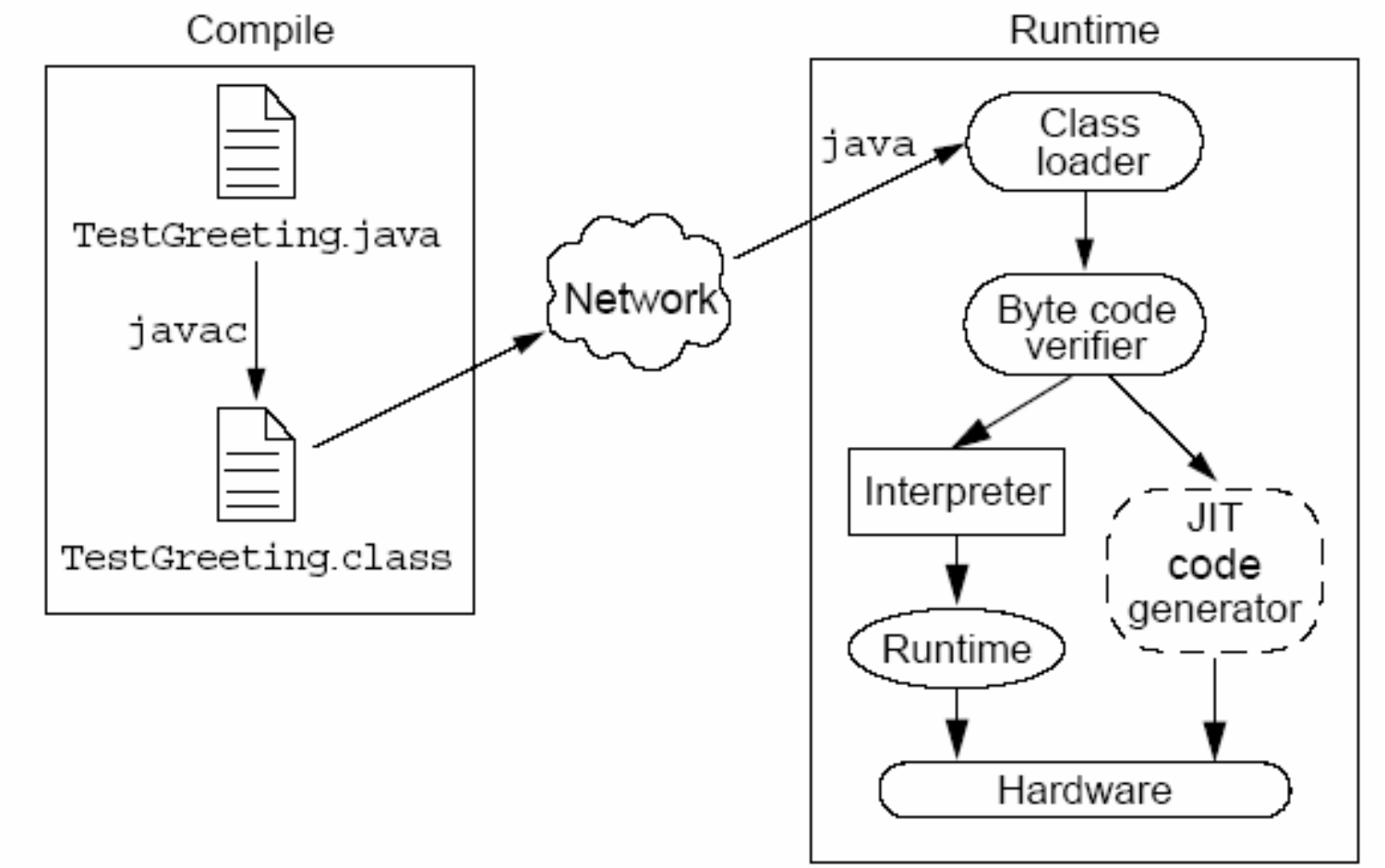
Chống sao chép



JDK

- Môi trường phát triển và thực thi do Sun Microsystems cung cấp (http://java.sun.com)
  - Phiên bản hiện tại J2SDK 5.0 (1.5)
- Bao gồm
  - *javac* Chương trình dịch chuyển mã nguồn sang *bytecode*
  - *java* Bộ thông dịch: Thực thi java application
  - *appletviewer* Bộ thông dịch: Thực thi java applet mà không cần sử dụng trình duyệt như *Nestcape*, hay *IE, v.v.*
  - *javadoc* Bộ tạo tài liệu dạng HTML từ mã nguồn và chú thích
  - *jdb* Bộ gỡ lỗi (*java debuger*)
  - *javap* Trình dịch ngược bytecode

Công nghệ JIT
Just-In-Time Code Generator

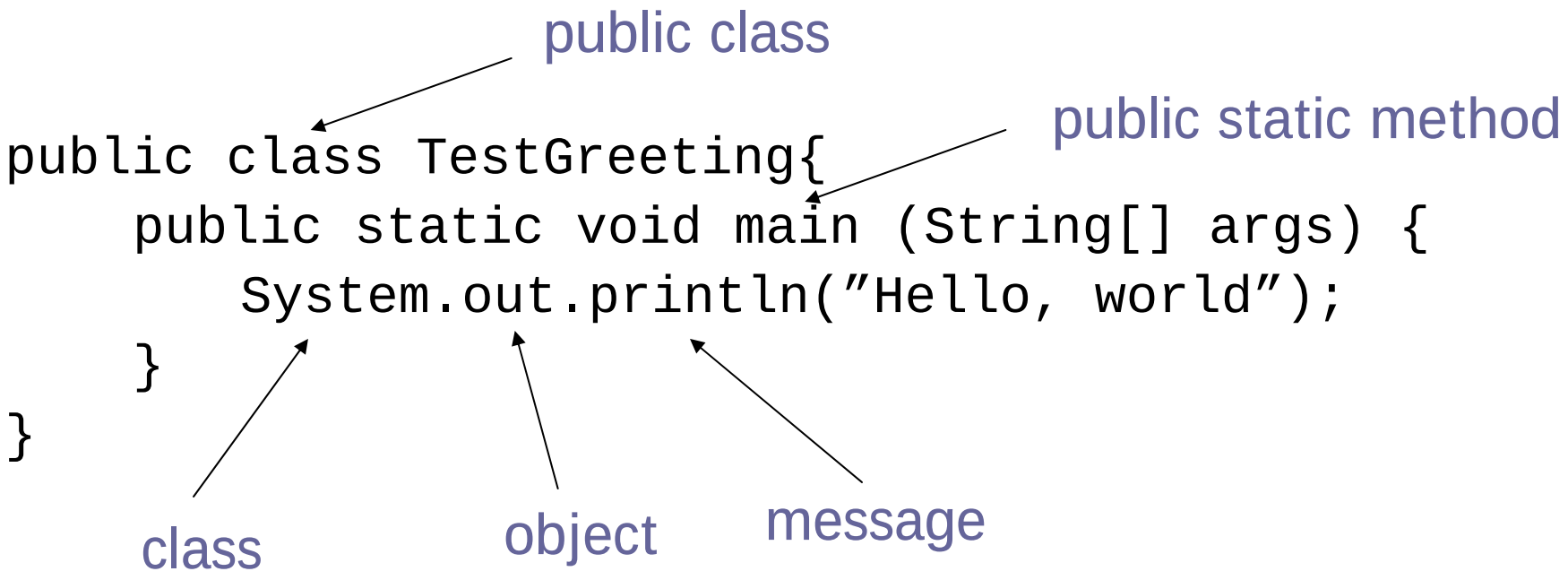


Java Applications

- Chương trình ứng dụng hoàn chỉnh
- Giao diện dòng lệnh hoặc đồ họa
- Được bắt đầu bởi phương thức (hàm) `main()` là phương thức `public static`

Chương trình Java đơn giản

TestGreeting.java:



Biên dịch và thực hiện

- Biên dịch `TestGreeting.java`

```
javac TestGreeting.java
```

- Thực hiện

```
java TestGreeting
```

- Kết quả

```
Hello, world
```

Một chút cải tiến

TestGreeting.java:

```
public class TestGreeting {
    public static void main(String[] args) {
        Greeting gr = new Greeting();
        gr.greet();
    }
}
```

Greeting.java:

```
public class Greeting {
    public void greet() {
        System.out.print(”Hello, world”);
    }
}
```

Biên dịch và thực hiện

- Biên dịch `TestGreeting.java`

```
javac TestGreeting.java
```

- `Greeting.java` được biên dịch tự động
- Thực hiện

```
java TestGreeting
```

- Kết quả

```
Hello, world
```

Java Applets

- Được nhúng trong một ứng dụng khác (web browser)
- Có giao diện hạn chế (đồ họa)
- Không truy cập được tài nguyên của client (không thực hiện được các hành vi *xấu*)

Applet đơn giản

Welcome.java:

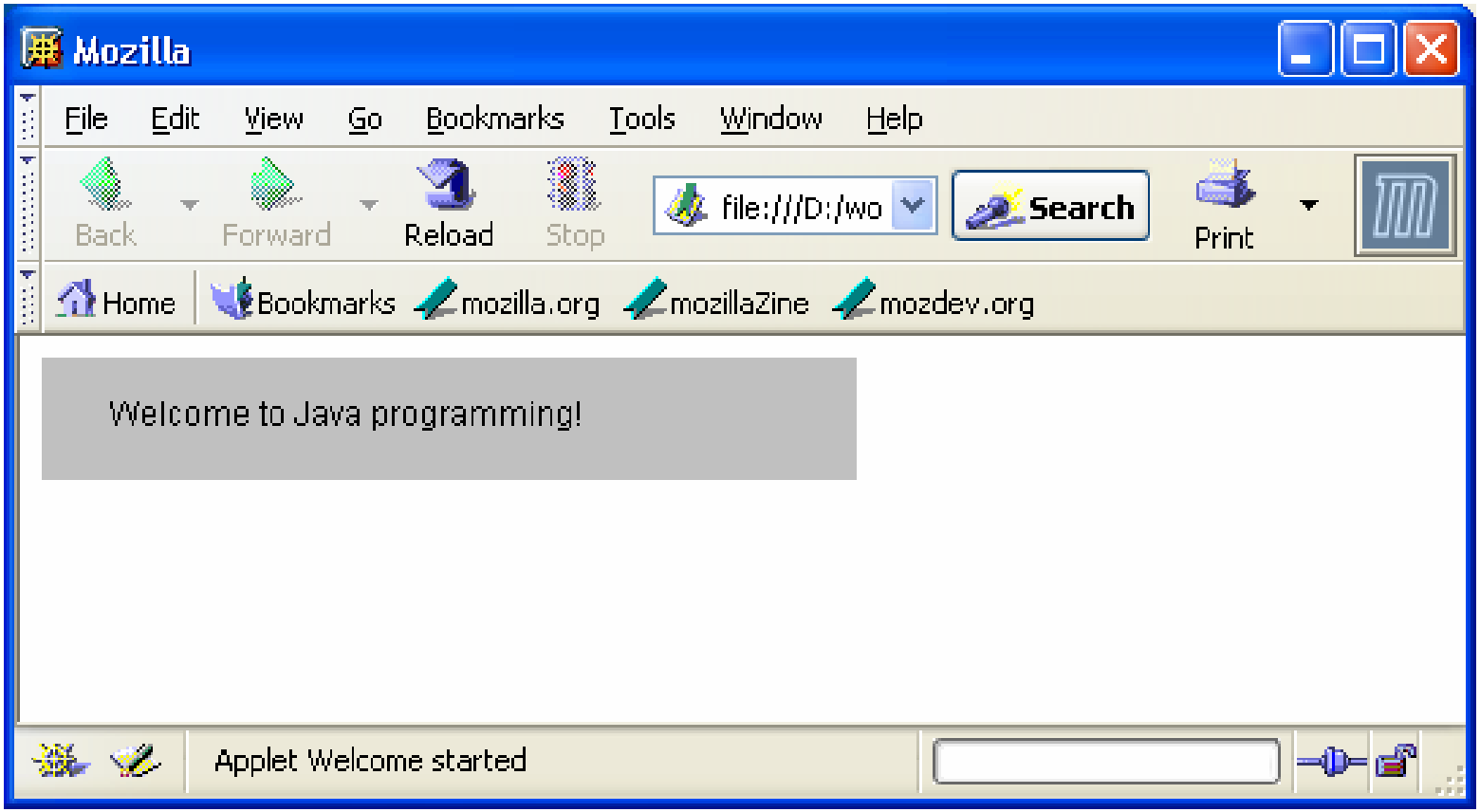
```
// Java packages
import java.awt.Graphics;
import java.applet.Applet;

public class Welcome extends Applet {

    public void paint(Graphics g)
    {
        // call superclass version of method paint
        super.paint(g);

        // draw a String
        g.drawString(”Welcome to Java programming!”, 25, 25);
    }
}
```

Nhúng vào trang Web

Welcome.html:

```
<html>
<applet code = ”Welcome.class”
  width = ”300” height = ”45”>
</applet>
</html>
```

Thực hiện (trong webbrowser)



Thực hiện

```
appletviewer Welcome.html
```

Các phương thức của Applet

- `init()`: khởi tạo applet
- `start()`: khởi động applet
  - mặc định sẽ gọi `paint()`
- `stop()`: dừng applet
- `destroy()`: giải phóng (hủy) applet

Thực hành

- Đăng nhập vào website môn học
- Làm quen với môi trường phát triển Java trên Linux và Windows
- Tập viết các ứng dụng nhỏ
  - các ví dụ trong bài giảng
  - chuyển các bài thực hành cơ bản của môn C/C++ sang Java

Bài tập: Tìm hiểu về Java

- Các kiểu dữ liệu cơ bản
  - các kiểu số nguyên, kiểu ký tự, kiểu logic
- Từ khóa, cách đặt tên (lớp, phương thức, biến
- Các cấu trúc điều khiển cơ bản
  - điều kiện
  - vòng lặp
  - switch

Lập trình hướng đối tượng

Khái niệm

Nội dung

- Lịch sử phát triển của kỹ thuật lập trình
- Hạn chế của kỹ thuật lập trình truyền thống
- Khái niệm lập trình hướng đối tượng
  - Đóng gói / Che dấu thông tin

Tài liệu tham khảo

- *Thinking in Java*, chapter 1, 2
- *Java how to program*, chapter 8

Mục tiêu của kỹ sư phần mềm

- Tạo ra sản phẩm tốt một cách có hiệu quả
- Nắm bắt được công nghệ
-

Phần mềm ngày càng lớn

- Một số hệ Unix chứa khoảng 4M dòng lệnh
- MS Windows chứa hàng chục triệu dòng lệnh
- Người dùng ngày càng đòi hỏi nhiều chức năng, đặc biệt là chức năng *thông minh*
- Phần mềm luôn cần được sửa đổi

Vì vậy

- Cần kiểm soát chi phí
  - Chi phí phát triển
  - Chi phí bảo trì
- Giải pháp chính là ***sử dụng lại***
  - Giảm chi phí và thời gian phát triển
  - Nâng cao chất lượng

Để sử dụng lại (mã nguồn)

- Cần dễ hiểu
- Được coi là chính xác
- Có giao diện rõ ràng
- *Không yêu cầu thay đổi khi sử dụng trong chương trình mới*

Các phương pháp lập trình

- Lập trình không có cấu trúc
- Lập trình có cấu trúc (lập trình thủ tục)
- Lập trình chức năng
- Lập trình logic
- Lập trình hướng đối tượng

Lập trình không có cấu trúc
(non-structured programming)

- Là phương pháp xuất hiện đầu tiên
  - các ngôn ngữ như Assembly, Basic
  - sử dụng các biến tổng thể
  - lạm dụng lệnh GOTO
- Các nhược điểm
  - khó hiểu, khó bảo trì, hầu như không thể sử dụng lại
  - chất lượng kém
  - chi phí cao
  - không thể phát triển các ứng dụng lớn

Ví dụ

```
10   k =1
20   gosub 100
30   if y > 120 goto 60
40   k = k+1
50   goto 20
60   print k, y
70   stop
100  y = 3*k*k + 7*k-3
110  return
```

Lập trình có cấu trúc/lập trình thủ tục (structured/procedural programming)

- sử dụng các lệnh có cấu trúc: for, do while, if then else...
- các ngôn ngữ: Pascal, C, ...
- chương trình là tập các hàm/thủ tục
- Ưu điểm
  - chương trình được cục bộ hóa, do đó dễ hiểu, dễ bảo trì hơn
  - dễ dàng tạo ra các thư viện phần mềm

Ví dụ

```
struct Date {
    int year, mon, day;
};
...
print_date(Date d) {
    printf(”%d / %d / %d\n”, d.day,
           d.mon, d.year);
}
```

Lập trình có cấu trúc/lập trình thủ tục

- Nhược điểm
  - dữ liệu và mã xử lý là tách rời
  - người lập trình phải biết cấu trúc dữ liệu (vấn đề này một thời gian dài được coi là hiển nhiên)
  - khi thay đổi cấu trúc dữ liệu thì mã xử lý (thuật toán) phải thay đổi theo
  - khó đảm bảo tính đúng đắn của dữ liệu
  - không tự động khởi tạo hay giải phóng dữ liệu động

Tại sao phải thay đổi cấu trúc dữ liệu?

- Cấu trúc dữ liệu là mô hình của bài toán cần giải quyết
  - Do thiếu kiến thức về bài toán, về miền ứng dụng..., không phải lúc nào cũng tạo được cấu trúc dữ liệu hoàn thiện ngay từ đầu.
  - Tạo ra một cấu trúc dữ liệu hợp lý luôn là vấn đề đau đầu của người lập trình.
- Bản thân bài toán cũng không bất biến
  - Cần phải thay đổi cấu trúc dữ liệu để phù hợp với các yêu cầu thay đổi.

Các vấn đề

- Thay đổi cấu trúc
  - dẫn đến việc sửa lại mã chương trình (thuật toán) tương ứng và làm chi phí phát triển tăng cao.
  - không tái sử dụng được các mã xử lý ứng với cấu trúc dữ liệu cũ.
- Đảm bảo tính đúng đắn của dữ liệu
  - một trong những nguyên nhân chính gây ra lỗi phần mềm là gán các dữ liệu không hợp lệ
  - cần phải kiểm tra tính đúng đắn của dữ liệu mỗi khi thay đổi giá trị

Ví dụ: MyDate

MyDate.java:

```
class MyDate {
    public int year, month, day;
}
```

MyCalendar.java:

```
MyDate d = new MyDate();
d.day = 32;                  // invalid day
d.day = 31; d.month = 2; // how to check
d.day = d.day + 1;           //
```

Ví dụ: MyDate (2)

Thay đổi cấu trúc dữ liệu:

MyDate.java:

```
class MyDate {
    public short year;
    public short mon_n_day;
}
```

Giải pháp

- Che dấu dữ liệu (che dấu cấu trúc)
- Truy cập dữ liệu thông qua giao diện xác định

```
class MyDate {
    private int year, mon, day;
    public int getDay() {...}
    public boolean setDay(int) {...}
    ...
}
```

Sử dụng giao diện

MyCalendar.java:

```
MyDate d = new MyDate();
...
d.day = 32;     // compile error
d.setDay(31);
d.setMonth(2); // should return False
```

Đóng gói/che dấu thông tin

- Đóng gói dữ liệu và các thao tác tác động lên dữ liệu thành một thể thống nhất (lớp đối tượng) thuận tiện cho sử dụng lại
- Che dấu thông tin
  - thao tác với dữ liệu thông qua các giao diện xác định
  - che dấu người lập trình khách (*client programmer*) cái có khả năng thay đổi (tách cái bất biến ra khỏi cái khả biến)

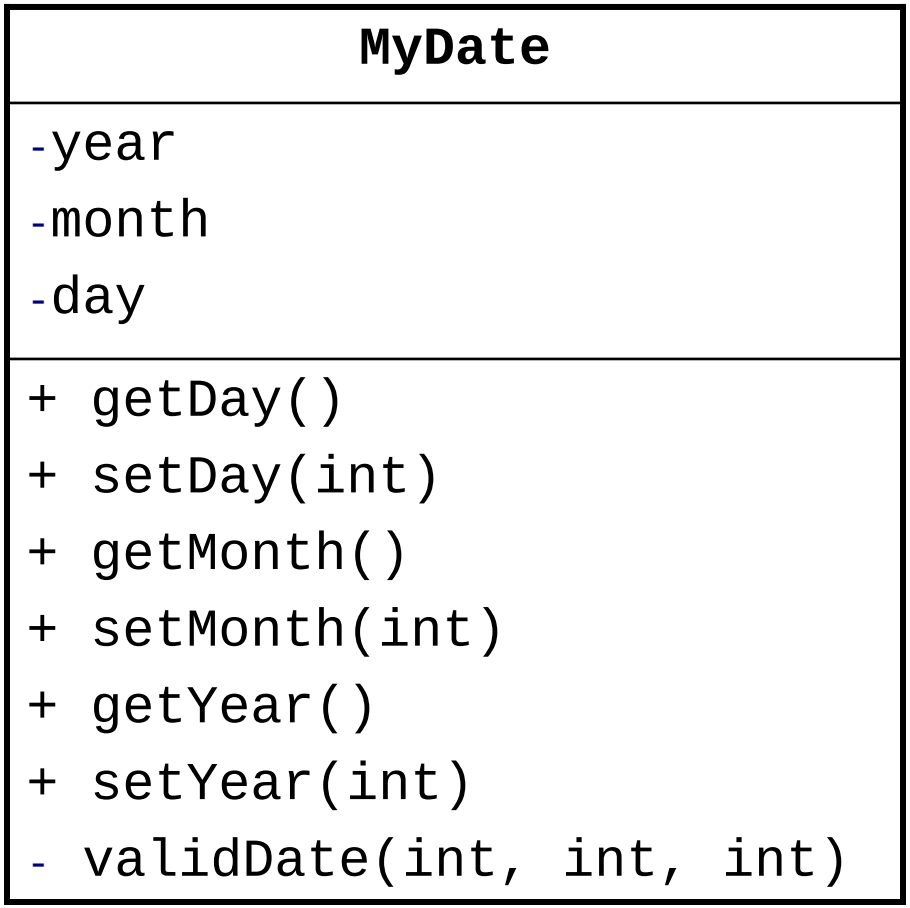
Lớp và đối tượng

- Lớp đối tượng (class) là khuôn mẫu để sinh ra đối tượng
- Đối tượng là thể hiện (instance) của một lớp. Đối tượng có
  - định danh
  - thuộc tính (dữ liệu)
  - hành vi (phương thức)

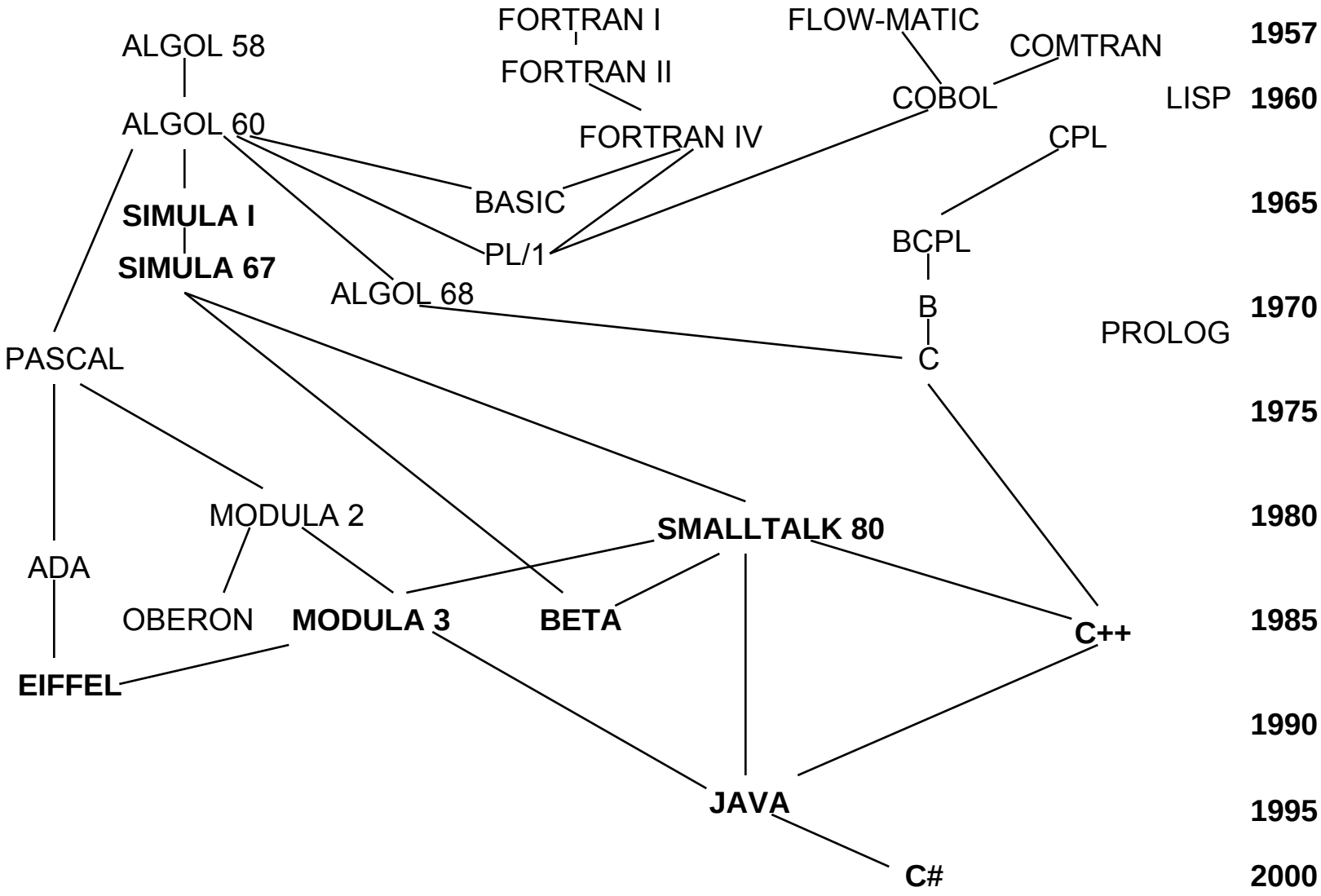
Hệ thống hướng đối tượng

- Bao gồm một tập các đối tượng
  - mỗi đối tượng chịu trách nhiệm một công việc
- Các đối tượng tương tác thông qua trao đổi thông điệp (message)
- Các đối tượng có thể tồn tại phân tán/có thể hoạt động song song

Mô hình hóa đối tượng



Lịch sử ngôn ngữ lập trình



Lập trình hướng đối tượng làm tăng

- năng suất lập trình (năng suất phát triển)
- chất lượng phần mềm
- tính hiểu được của phần mềm
- vòng đời của phần mềm

OOP và OOL

- Có thể thể hiện phần nào tư tưởng đóng gói/che dấu thông tin trên ngôn ngữ thủ tục
  - không triệt để, khó kiểm soát
- Ngôn ngữ hướng đối tượng cung cấp khả năng kiểm soát truy cập; ngoài ra
  - kế thừa
  - đa hình

Lớp và đối tượng trong Java

Nội dung

- Định nghĩa lớp
- Thuộc tính
- Phương thức
- Kiểm soát truy cập
- Phương thức khởi tạo
- Thao tác với đối tượng

Tài liệu tham khảo

- *Thinking in Java*, chapter 1, 2
- *Java how to program*, chapter 8

Định nghĩa lớp

Lớp được định nghĩa bởi

```
class class_name {
...
}
```

Ví dụ:

```
class MyDate {
}
```

Đối tượng

- Đối tượng được thao tác thông qua *tham chiếu*
  - Tham chiếu đóng vai trò gần giống như một con trỏ
- Đối tượng phải được tạo ra một cách tường minh bằng toán tử new

```
MyDate d;
d = new MyDate();
MyDate myBirthday = d;
```

Thuộc tính, phương thức và kiểm soát truy cập

```
class MyDate {
    private int year, mon, day;
    public int getYear() {
        return year;
    }
    public boolean setYear(int y) {
        ...
    }
    ...
}
```

```
MyDate d = new MyDate();
...
d.year = 2005;    // compile error
d.setYear(2005);
System.out.println(”Year=” + d.getYear());
```

Phương thức trùng tên (overload)

- Có thể định nghĩa các phương thức trùng tên, tuy nhiên phải phân biệt bởi danh sách tham số

```
class MyDate {
    …
    public boolean setMonth(int m) { …}
    public boolean setMonth(String s) { …}
}

d.setMonth(9);
d.setMonth(”September”);
```

Phương thức khởi tạo (constructor)

- Dữ liệu nên được khởi tạo trước khi sử dụng
  - lỗi khởi tạo là một trong các lỗi phổ biến
- Phương thức khởi tạo
  - là phương thức đặc biệt được gọi tự động sau khi tạo ra đối tượng
  - nhằm mục đích chính là khởi tạo cho các thuộc tính của đối tượng

Phương thức khởi tạo

- Có tên trùng với tên lớp
- Không nhận giá trị trả lại
- Mỗi khi đối tượng được tạo ra bởi toán tử new, hệ thống sẽ tự động gọi phương thức khởi tạo.
  - nếu không khai báo, hệ thống sẽ gọi constructor mặc định là một phương thức rỗng

Ví dụ: Constructor rỗng

```
class SayMsg {
}
…

SayMsg msg = new SayMsg();
```

Ví dụ: Constructor mặc định

```
class SayMsg {
    SayMsg() {
        System.out.println(”Hello”);
    }
}
...
SayMsg msg = new SayMsg();
```

Ví dụ:

```
class SayMsg {
    SayMsg() {
        System.out.println(”Hello”);
    }
    SayMsg(String s) {
        System.out.println(s);
    }
}
...
SayMsg msg1 = new SayMsg();
SayMsg msg2 = new SayMsg(”Java”);
```

Ví dụ:

```
class SayMsg {
    SayMsg(String s) {
        System.out.println(s);
    }
}
...
SayMsg msg1 = new SayMsg();      // error
SayMsg msg2 = new SayMsg("");
```

Copy constructor

- Khởi tạo đối tượng bằng một đối tượng khác

```
public class MyDate {
    private int year, month, day;
    public MyDate() {...}
    public MyDate(MyDate d) {
        year = d.year;
        month = d.month;
        day = d.day;
    }
...
}
```

```
MyDate d = new MyDate();
d.setYear(2005);
d.setMonth(9);
d.setDay(12);
MyDate openDay = new MyDate(d);
MyDate dd = d;
```

Kiểm soát truy cập

```
public class MyDate {
    private int year, month, day;
    public MyDate() {...}
    public MyDate(MyDate d) {
        year = d.year; // year = d.getYear();
        month = d.month;
        day = d.day;
    }
...
}
```

Hiểu thêm về Java

Nội dung

- Dữ liệu kiểu nguyên thủy và đối tượng
- Tham chiếu
- Giải phóng bộ nhớ
- Gói và kiểm soát truy cập
- Kiểu hợp thành (composition)
- Vào ra với luồng dữ liệu chuẩn

Tài liệu tham khảo

- *Thinking in Java*, chapter 2, 4, 5
- *Java how to program*, chapter 4,5,6,7,8

Kiểu dữ liệu nguyên thủy

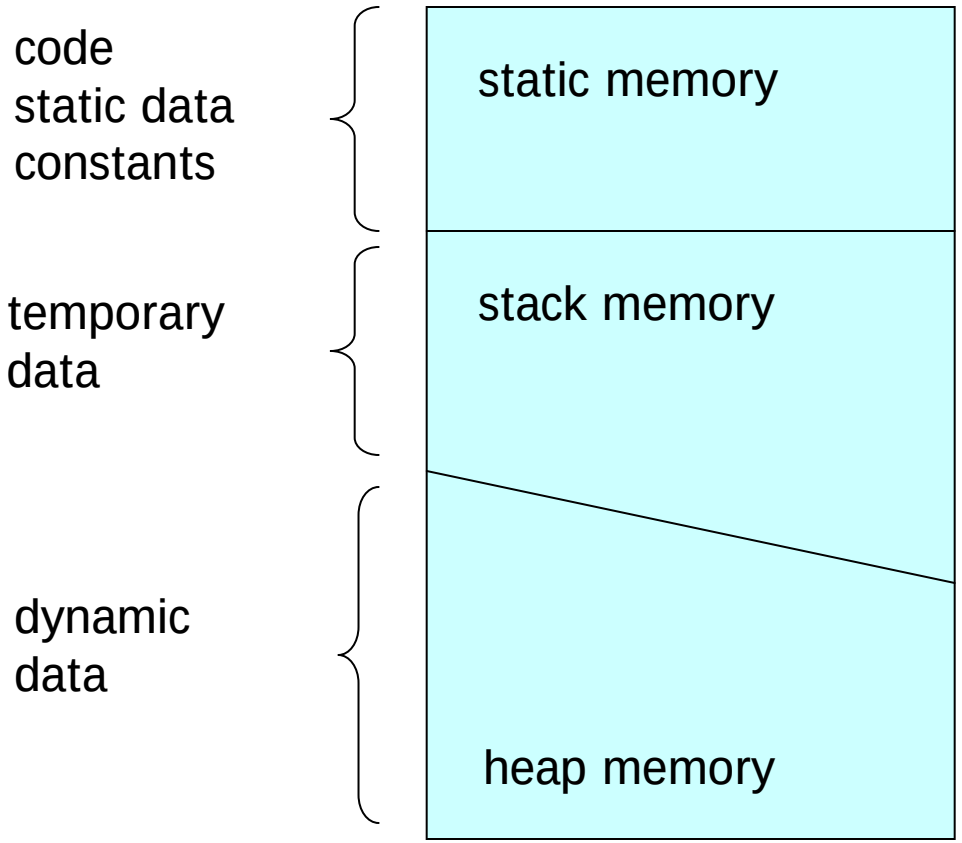
- Java cung cấp các kiểu nguyên thủy
  - số: byte, short, int, long, float, double
    - không có khái niệm unsigned
    - kích thước cố định trên mọi platform
  - logic: boolean
  - ký tự: char
- Dữ liệu kiểu nguyên thủy không phải là đối tượng
  - int a = 5;
  - if (a==b)…
- Tồn tại lớp đối tượng tương ứng: Interger, Float,..
  - Interger count = new Interger(0);

Kiểu dữ liệu	Độ rộng (bits)	Giá trị cực tiểu	Giá trị cực đại
char	16	0x0	0xffff
byte	8	-128 (-2^7)	+127 (2^7-1)
short	16	-32768 (-2^{15})	32767 (2^{15}-1)
int	32	- 2^{31}, 0x80000000	+ 2^{31} - 1, 0x7fffffff
long	64	- 2^{63}	+ 2^{63} - 1
float	32	1.4012984632481707e-45	3.4028234663852860e+38
double	64	4.9406564584124654e-324	1.7976931348623157e+308
boolean			

Dữ liệu được lưu trữ ở đâu

- Dữ liệu kiểu nguyên thủy
  - thao tác thông qua *tên biến*
- Dữ liệu là thuộc tính của đối tượng
  - Đối tượng được thao tác thông qua tham chiếu
- Vậy biến kiểu nguyên thủy, tham chiếu và đối tượng được lưu trữ ở đâu?

3 vùng bộ nhớ cho ứng dụng



Tham chiếu

- Đối tượng được thao tác thông qua tham chiếu
  - là *con trỏ* tới đối tượng
  - thao tác trực tiếp tới thuộc tính và phương thức
  - không có các toán tử con trỏ
  - phép gán (=) không phải là phép toán copy nội dung đối tượng
- tham chiếu được lưu trữ trong vùng nhớ static/stack như các con trỏ trong C/C++

Toán tử New

- Phải tạo mọi đối tượng một cách tường minh bằng toán tử new
  - cấp phát vùng nhớ động
  - được tạo trong bộ nhớ Heap
- Ví dụ:

```
MyDate d;
MyDate birthday;
d = new MyDate();
```

Phép gán “=”

- Phép gán không phải là copy thông thường
  - copy nội dung của tham chiếu
  - hai tham chiếu sẽ tham chiếu đến cùng đối tượng

```
Integer m = new Integer(10);
Integer n  = new Integer(20);
m = n;
n.setValue(50);
System.out.print(m);
```

“New” và “=”

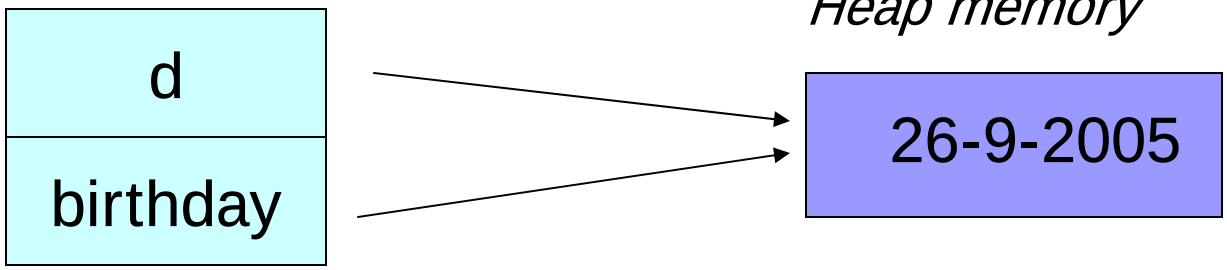
```
MyDate d;
MyDate birthday;
d = new MyDate(26,9,2005);
birthday = d;
```

new operation

assign operation

Static/Stack memory

Heap memory



Toán tử quan hệ “==”

- So sánh nội dung của các dữ liệu kiểu nguyên thủy (int, long, float, …)
- So sánh nội dung của tham chiếu chứ không so sánh nội dung của đối tượng do tham chiếu trỏ đến

```
Integer n1 = new Integer(47);
Integer n2 = new Integer(47);
System.out.println(n1 == n2);
System.out.println(n1 != n2);
--
false
true
```

So sánh nội dung đối tượng

```
class MyDate {
...
    boolean equalTo(MyDate d) {
    ...
    }
}
...
MyDate d1 = new MyDate(10,10,1954);
MyDate d2 = new MyDate(d1);
System.out.println(d1.equalTo(d2));
```

Giải phóng bộ nhớ động (Garbage collection)

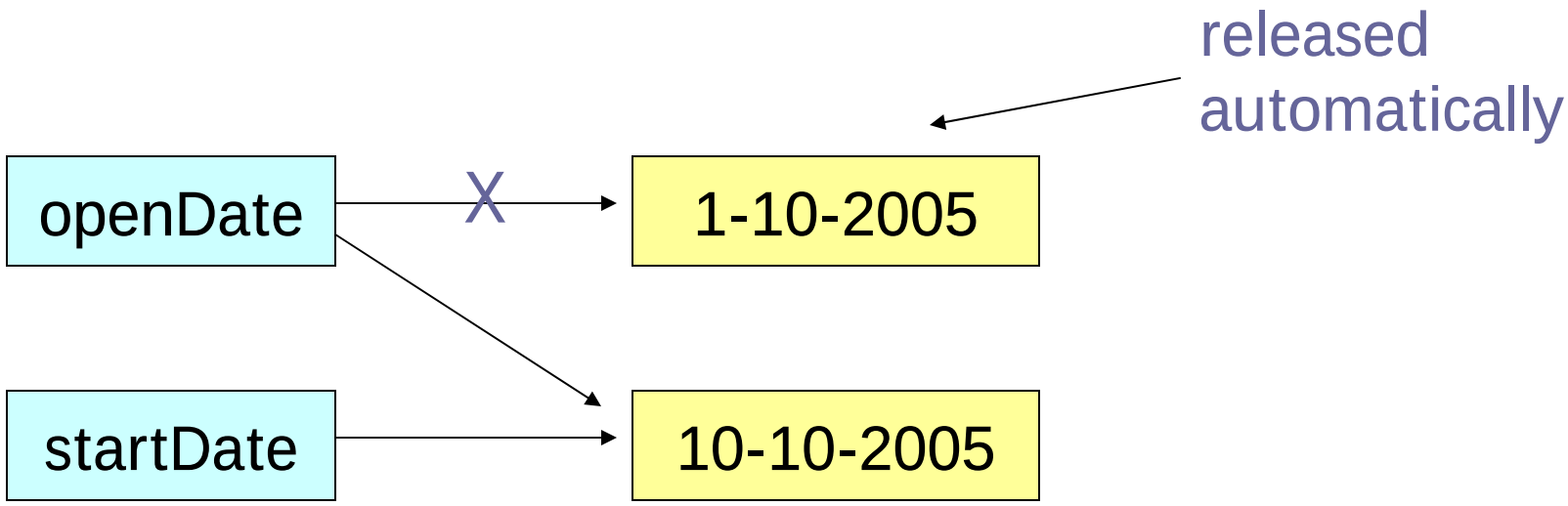
- Lập trình viên không cần phải giải phóng đối tượng
- JVM cài đặt cơ chế "Garbage collection" để giải phóng tự động các đối tượng không còn cần thiết
  - *tuy nhiên, GC không nhất thiết hoạt động với mọi đối tượng*
- GC tăng tốc độ phát triển và tăng tính ổn định của ứng dụng
  - Không phải viết mã giải phóng đối tượng
  - *Do đó, không bao giờ quên giải phóng đối tượng*

GC hoạt động như thế nào

- Sử dụng cơ chế đếm?
  - mỗi đối tượng có một số đếm các tham chiếu trỏ tới
  - giải phóng đối tượng khi số đếm = 0
- Giải phóng các đối tượng *chết*
  - kiểm tra tất cả các tham chiếu
  - đánh dấu các đối tượng còn được tham chiếu
  - giải phóng các đối tượng không được tham chiếu

Garbage Collection

```
MyDate openDate = new MyDate(1,10,2005);
MyDate startDate = new MyDate(10,10,2005);
openDate = startDate;
```



Truyền tham số và nhận giá trị trả lại

- Truyền giá trị
  - đối với dữ liệu kiểu nguyên thủy
  - giá trị của tham số (RValue) được copy lên stack
  - có thể truyền hằng số (vd: 10, 0.5, …)
- Truyền tham chiếu
  - đối với đối tượng
  - nội dung của tham chiếu (LValue) được copy lên stack

Truyền tham số trị

```
class MyDate {
...
    public boolean setYear(int y) {
        ...
    }
    public int getYear() {
        return year;
    }
...
}
...
MyDate d = new MyDate();
d.setYear(1975);
int y = d.getYear();
```

Truyền tham chiếu

```
class MyDate {
    int year, month, day;
    public MyDate(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copy(MyDate d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public MyDate copy() {
        return new MyDate(day, month, year);
    }
...
}
```

Truyền tham chiếu

```
MyDate d1 = MyDate(2005, 9, 26);
MyDate d2 = MyDate(2000, 1, 1);
d1.copy(d2);
MyDate d3;
d3 = d1.copy();
```

Tham chiếu `this`

- Java cung cấp tham chiếu `this` để trỏ tới chính đối tượng đang hoạt động
- `this` được sử dụng vào các mục đích như
  - tham chiếu tường minh đến thuộc tính và phương thức của đối tượng
  - truyền tham số và trả lại giá trị
  - dùng để gọi constructor

`this` làm giá trị trả lại

```
class Counter {
    private int c = 0;
    public Counter increase() {
        c++;
        return this;
    }
    public int getValue() {
        return c;
    }
}
...
Counter c = new Counter();
System.out.println(c.increase().increase().getValue());
```

this làm tham số

```
class Document {
    Viewer vi;
    ...
    Document(Viewer v) {
        vi = v;
        ...
    }
    void display() {
        v.display(this);
    }
...
}
```

Gọi constructor bằng `this`

```
class MyDate {
    private int year, month, day;

    public MyDate(int y, int m, int d) {
        ...
    }
    // copy constructor
    MyDate(MyDate d) {
        this(d.year, d.month, d.day);
        System.out.println("copy constructor called");
    }
...
}
```

- Constructor chỉ được gọi bên trong một constuctor khác và chỉ được gọi một lần ở thời điểm (vị trí) đầu tiên.

Phương thức và thuộc tính static

- Có thể khai báo phương thức và thuộc tính là tĩnh (static)
  - độc lập với đối tượng
  - có thể sử dụng mà không cần có đối tượng
- Phương thức tĩnh
  - không sử dụng được thuộc tính thông thường (non-static)
  - không gọi được các phương thức thông thường

Gói các lớp đối tượng (package)

- Các lớp đối tượng được chia thành các gói
  - nếu không khai báo thì các lớp thuộc gói default
  - các lớp trong cùng một tệp mã nguồn luôn thuộc cùng một gói
- Tồn tại mức truy cập package
  - mức package là mặc định (nếu không khai báo tường minh là public hay private)
  - các đối tượng của các lớp thuộc cùng gói có thể truy cập đến non-private members của nhau
  - chỉ có thể tạo (new) đối tượng của lớp được khai báo là public của gói khác

Hello.java:

```
class HelloMsg {
    void sayHello() {
        System.out.println(”Hello, world!”);
    }
}

public class Hello {
    public static void main(String[] args) {
        HelloMsg msg = new HelloMsg();
        msg.sayHello();
    }
}
```

Khai báo và sử dụng package

- Khai báo gói bằng lệnh `package`
  - các gói được lưu trữ theo cấu trúc cây thư mục
  - sử dụng tham số -d để tạo thư mục khi biên dịch
- Dùng lệnh `import` để khai báo việc sử dụng một gói đã có

Đối tượng hợp thành (Composition)

- Đối tượng có thể chứa các đối tượng khác (các thuộc tính không thuộc kiểu nguyên thủy)
- Thuộc tính là tham chiếu phải được tạo ra bằng new hoặc được gán cho một đối tượng đã tồn tại

```
class Person {
    private String name;
    private MyDate birthday;
...
}
```

Get và Set thuộc tính không thuộc kiểu nguyên thủy

```
class Person {
…
    public MyDate getBirthday() {
        return birthday;
    }
}

Person p = new Person(...);
MyDate d = p.getBirthday();
d.setYear(1900);
```

Sử dụng copy constructor

```
class Person {
    private String name;
    private MyDate birthday;
    public Person(String s, MyDate d) {
        name = s;
        birthday = new MyDate(d);
    }
    public MyDate getBirthday() {
        return new MyDate(birthday);
    }
    public void setBirthday(MyDate d) {
        birthday = new MyDate(d);
    }
...
}
```

Vào ra từ luồng dữ liệu chuẩn

- Luồng ra chuẩn: System.out
  - xuất ra luồng ra chuẩn (standard output)
  - có thể tái định hướng
- Luồng thông báo lỗi: System.err
  - xuất ra Console (thiết bị output chuẩn)
  - không thể tái định hướng
- Luồng dữ liệu vào chuẩn: System.in
  - chưa sẵn sàng cho sử dụng

Nhập dữ liệu từ luồng vào chuẩn

- InputStream: lớp đối tượng ứng với luồng vào chuẩn
  - System.in: đối tượng tương ứng
  - chưa có phương thức nhập dữ liệu
- InputStreamReader: nhập dữ liệu không thông qua buffer
  - đọc từng ký tự (kể cả ký tự đặc biệt)
- BufferedReader: sử dụng buffer
  - đọc từng dòng

Ví dụ

```
import java.io.*;

public class Echo {
    public static void main(String[] args)
    throws IOException {
        InputStreamReader reader;
        BufferedReader bufReader;
        reader = new InputStreamReader(System.in);
        bufReader = new BufferedReader(reader);
        String s;
        while( null != (s = bufReader.readLine())
            System.out.println(s);
    }
}
```

Nhập một số

```
import java.io.*;

class SimpleIO {
   public static void main(String args[])
   throws IOException {
        int n;
        String str;
        ...
        str = bufReader.readLine();
        Integer num = Integer.valueOf(str);
        n = num.intValue();
        System.out.println(n);
   }
}
```

Nhập một số

```
import java.io.*;

class SimpleIO {
   public static void main(String args[])
   throws IOException {
        int n;
        String str;
       ...
       str = bufReader.readLine();
       n = Integer.valueOf(str).intValue();
       System.out.println(n);
   }
}
```

Tham số dòng lệnh

CmdLineParas.java:

```
public class CmdLineParas {
public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
        System.out.println(args[i]);
    }
}
```

Ví dụ:

```
#java CmdLineParas hello world
hello
world
```

Kế thừa

Nội dung

- Vấn đề sử dụng lại
- Sử dụng lại bằng kế thừa
- Kế thừa trong Java
  - định nghĩa lớp kế thừa
  - thêm phương thức, thuộc tính
  - kiểm soát truy cập
  - constructor
- Lớp Object

Tài liệu tham khảo

- *Thinking in Java*, chapter 6
- *Java how to program*, chapter 9

Sử dụng lại

- Tồn tại nhiều loại đối tượng có các thuộc tính và hành vi tương tự hoặc liên quan đến nhau
  - Person, Student, Manager,…
- Xuất hiện nhu cầu sử dụng lại các mã nguồn đã viết
  - Sử dụng lại thông qua copy
  - Sử dụng lại thông qua quan hệ *has_a*
  - Sử dụng lại thông qua cơ chế “kế thừa”

Sử dụng lại

- Copy mã nguồn
  - Tốn công, dễ nhầm
  - Khó sửa lỗi do tồn tại nhiều phiên bản
- Quan hệ *has_a*
  - Sử dụng lớp cũ như là thành phần của lớp mới
  - Sử dụng lại cài đặt với giao diện mới
    - Phải viết lại giao diện
    - Chưa đủ mềm dẻo

Ví dụ: *has_a*

```
class Person {
    private String name;
    private Date bithday;
    public String getName() { return name; }
...
}
class Employee {
    private Person me;
    private double salary;
    public String getName() { return me.getName(); }
...
}
```

```
class Manager {
   private Employee me;
   private Employee assistant;
   public setAssistant(Employee e) {...}
   ...
}
...
Manager junior = new Manager();
Manager senior = new Manager();
senior.setAssistant(junior);  // error
```

Kế thừa

- Dựa trên quan hệ *is_a*
- Thừa hưởng lại các thuộc tính và phương thức đã có
- Chi tiết hóa cho phù hợp với mục đích sử dụng mới
  - Thêm các thuộc tính mới
  - Thêm hoặc hiệu chỉnh các phương thức

Thuật ngữ

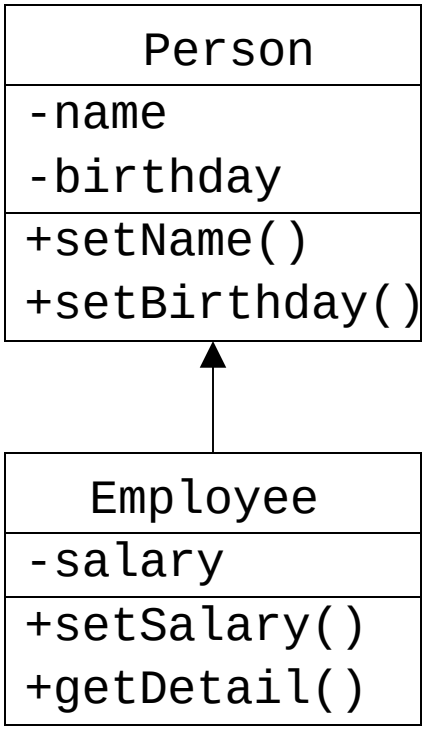
- Kế thừa
- Lớp cơ sở, lớp cha
- Lớp dẫn xuất, lớp con

Kế thừa trong Java

```
[public] class DerivedClass extends BaseClass {
   /* new features goes here */
}
```

Ví dụ:

```
class Employee extends Person {
    private double salary;
    public boolean setSalary(double sal) {
        ...
        salary = sal;
        return true;
    }
}
```




```
Employee e = new Employee();

e.setName("John");
e.setSalary(3.0);
```

private members

```
class Employee extends Person {
...
    public String getDetail() {
        String s;
        //  s = name + "," + birthday;
        s = getName() + "," + getBirthday();
        s += "," + salary;
        return s;
    }
}
```

Mức truy cập protected

- Để đảm bảo che dấu thông tin, thông thường các thuộc tính được khai báo là private
  - Đối tượng thuộc lớp dẫn xuất phải truy cập tới chúng thông qua các phương thức get và set.
- Mức truy cập protected giải quyết vấn đề này
  - Đối tượng của lớp dẫn xuất truy cập được các *protected members* của lớp cơ sở
  - Các đối tượng khác không truy cập được

```
public class Person {
    protected Date birthday;
    protected String name;
...
}

public class Employee extends Person {
...
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        s += "," + salary;
        return s;
    }
}
```

Các mức kiểm soát truy cập

Modifier	Same class	Same package	Subclass	Universe
private	Yes			
package (*default*)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Trong cùng gói

```
public class Person {
    Date birthday;
    String name;
...
}
```

```
public class Employee extends Person {
...
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        s += "," + salary;
        return s;
    }
}
```

Khác gói

```
package abc;
public class Person {
    protected Date birthday;
    protected String name;
...
}
```

```
import abc.Person;
public class Employee extends Person {
...
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        s += "," + salary;
        return s;
    }
}
```

Định nghĩa lại các phương thức

- Chúng ta có thể định nghĩa lại các phương thức của lớp cơ sở
- 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

```
package abc;
public class Person {
    protected Date birthday;
    protected String name;
    public  String getDetail() {...}
...
}
```

```
import abc;
public class Employee extends Person {
...
    public String getDetail() {
        String s;
        s = super.getDetail() + "," + salary;
        return s;
    }
}
```

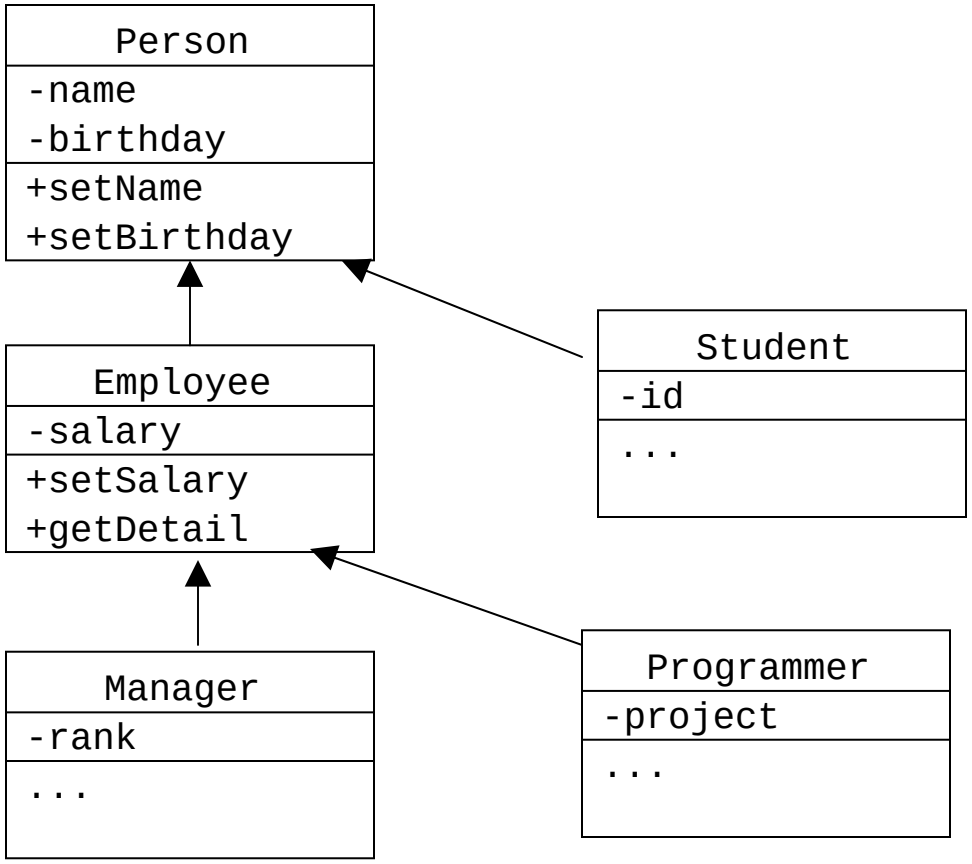
Định nghĩa lại phương thức

- Phải có quyền truy cập không *chặt* hơn phương thức được định nghĩa lại
- Phải có kiểu giá trị trả lại như nhau

```
class Parent {
    public void doSomething() {}
    public int doSomething2() {
        return 0;
    }
}

class Child extends Parent {
    protected void doSomething() {}
    public void doSomething2() {}
}
```

Thừa kế nhiều tầng

Mọi đối tượng đều
thừa kế từ lớp gốc Object



Constructor

- Lớp dẫn xuất kế thừa mọi thuộc tính và phương thức của lớp cơ sở
- *Không kế thừa phương thức khởi tạo*
- Có hai giải pháp gọi constructor của lớp cơ sở
  - sử dụng constructor mặc định
  - gọi constructor của lớp cơ sở một cách tường minh

```
class Point {
   protected int x, y;
   public Point() {}
   public Point(int xx, int yy) {
       x = xx;
       y = yy;
   }
}

class Circle extends Point {
   protected int radius;
  public Circle() {}
}

Point p = new Point(10, 10);
Circle c1 = new Circle();
Circle c2 = new Circle(10, 10); // erorr
```

Gọi constructor của lớp cơ sở

- Việc khởi tạo thuộc tính của lớp cơ sở nên giao phó cho constructor của lớp cơ sở
- Sử dụng từ khóa `super` để gọi constructor của lớp cơ sở
  - Constructor của lớp cơ sở bắt buộc phải được thực hiện đầu tiên
  - Nếu lớp cơ sở không có constructor mặc định thì bắt buộc phải gọi constructor tường minh

```
class Point {
   protected int x, y;
   public Point() {}
   public Point(int xx, int yy) {
       x = xx;
       y = yy;
   }
}

class Circle extends Point {
   protected int radius;
   public Circle() {}
   public Circle(int xx, int yy, int r) {
       super(xx, yy);
       radius = r;
  }
}
```

```
class Point {
   protected int x, y;
   public Point(int xx, int yy) {
       x = xx;
       y = yy;
   }
}

class Circle extends Point {
   protected int radius;
   public Circle() { super(0, 0); }
   public Circle(int xx, int yy, int r) {
       super(xx, yy);
       radius = r;
  }
}
```

```
class Point {
   protected int x, y;
   public Point() {}
   public Point(int xx, int yy) {
       x = xx;
       y = yy;
   }
}

class Circle extends Point {
   protected int radius;
   public Circle() { }
   public Circle(int xx, int yy, int r) {
       // super(xx, yy);
       radius = r;
  }
}
```

Thứ tự khởi tạo

```
class Point {
   protected int x, y;
   public Point() {
       System.out.println("Point constructor");
   }
}
class Circle extends Point {
   protected int radius;
   public Circle() {
       System.out.println("Circle constructor");
   }
}
...
Circle c = new Circle();
```

Từ khóa `final`

- Thuộc tính final
  - hằng số, chỉ được gán giá trị khởi tạo một lần, không thay đổi được giá trị
- Phương thức final
  - không cho phép định nghĩa lại ở lớp dẫn xuất
- Tham số final
  - không thay đổi được giá trị của tham chiếu
- Lớp final
  - không định nghĩa được lớp dẫn xuất

Kế thừa và đa hình

Nội dung

- Đa hình
  - upcasting
  - liên kết động
- Lớp và phương thức trừu tượng
- Đa kế thừa và giao diện
- Một cách lập trình tổng quát

Tài liệu tham khảo

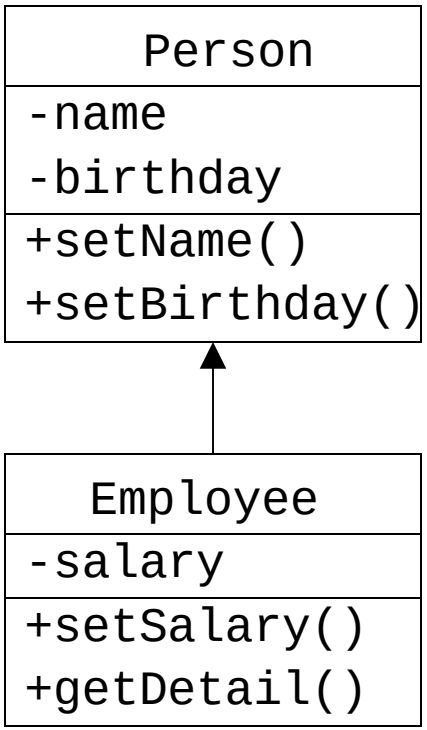
- *Thinking in Java*, chapter 7, 8
- *Java how to program*, chapter 9

Polymorphism (đa hình) là gì

- Polymorphism: nhiều hình thức, nhiều kiểu tồn tại
- Đa hình trong lập trình
  - đa hình hàm: hàm trùng tên, phân biệt bởi danh sách tham số
  - đa hình đối tượng
    - nhìn nhận đối tượng theo nhiều kiểu khác nhau
    - các đối tượng khác nhau giải nghĩa thông điệp theo cách thức khác nhau

Up casting

- Up casting là khả năng nhìn nhận đối tượng thuộc lớp dẫn xuất như là một đối tượng thuộc lớp cơ sở
  - dùng đối tượng của lớp dẫn xuất để truyền tham số
  - dùng đối tượng của lớp dẫn xuất làm thuộc tính




```
Person p;
Employee e = new Employee();
p = (Person) e;
p.setName(...);
p.setSalary(...);  // compile error
```

```
String teamInfo(Person p1, Person p2) {
    return "Leader: " + p1.getName() +
           "; member: " + p2.getName();
}
...
Employee e1, e2;
Manager  m1, m2;
…
System.out.println(teamInfo(e1, e2));
teamInfo(m1, m2); teamInfo(m1,e2);
```

```
class Manager extends Employee {
    Employee assistant;
...
    public void setAssistant(Employee e) {
        assistant = e;
    }
...
}
...
Manager junior, senior;
...
senior.setAssistant(junior);
```

Đa hình và liên kết động

- Khả năng giải nghĩa các thông điệp theo các cách thức khác nhau

```
Person p1 = new Person();
Person p2 = new Employee();
Person p3 = new Manager();
...
System.out.println(p1.getDetail());
System.out.println(p2.getDetail());
System.out.println(p3.getDetail());
```

```
class EmployeeList {
    Employee list[];
...
    public void add(Employee e) {...}
    public void print() {
        for (int i=0; i<list.length; i++) {
            System.out.println(list[i].getDetail());
...
}
...
EmployeeList list = new EmployeeList();
Employee e1; Manager m1;
...
list.add(e1); list.add(m1);
list.print();
```

Liên kết tĩnh và liên kết động

Static and dynamic binding

- Liên kết tĩnh: lời gọi hàm (phương thức) được quyết định khi biên dịch, do đó chỉ có một phiên bản của chương trình con được thực hiện
  - ưu điểm về tốc độ
- Liên kết động: lời gọi phương thức được quyết định khi thực hiện, phiên bản của phương thức phù hợp với đối tượng được gọi
  - Java mặc định sử dụng liên kết động

Down casting

```
Employee e = new Employee();
Person p = e;  // up casting
Employee ee = (Employee)p;  // down casting
Manager m = (Manager)ee;  //  run-time error

Person p2 = new Manager();
Employee e2 = (Employee) p2;
```

Toán tử instanceof

```
public class Employee extends Person {}
public class Student extends Person {}
---
public doSomthing(Person e) {
    if (e instanceof Employee) {...
    } else if (e instanceof Student) {...
    } else {...}
}
```

Private method

```
class Base {
  private void f() { System.out.println(”base f()”); }
  public void show() { f() }
}

public class Derived extends Base {
  private void f() {
    System.out.println(”derived f()”);
  }
  public static void main(String args[]) {
    Derived d = new Derived();
    Base b = d;
    b.show();
  }
}
```

Copy constructor(?)

```
class Employee extends Person {
    double salary;
    Employee(Employee e) {
        super(e);
        salary = e.salary;
    }
...
}
```

Gọi phương thức trong constructor

```
class Shape {
   public Shape() {
        draw();
   }
   public void draw() {}
}

class Point extends Shape {
    protected int x, y;
    public Point(int xx, int yy) {
        x = xx;  y = yy;
    }
    public void draw() {
        System.out.println("(" + x + "," + y + ")");
    }
}
--
Point p = new Point(10, 10);
```

Lớp trừu tượng

- Chúng ta có thể tạo ra các lớp cơ sở để tái sử dụng mà không muốn tạo ra đối tượng thực của lớp
  - các lớp Point, Circle, Rectangle chung nhau khái niệm cùng là hình vẽ *Shape*
- Giải pháp là khái báo lớp trừu tượng
  - không thể tạo đối tượng

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

...
Shape s1 = new Circle();
Shape s = new Shape(10, 10) //  compile error
```

```
class Circle extends Shape {
  int r;

  public Circle(int _x, int _y, int _r) {
     super(_x, _y);
     r = _r;
  }
...
}
```

Phương thức trừu tượng

- Để thống nhất giao diện, có thể khai báo các phương thức tại lớp cơ sở nhưng được cài đặt thực tế tại lớp dẫn xuất
  - các lớp dẫn xuất khác nhau có cách cài đặt khác nhau
- Phương thức trừu tượng
  - bắt buộc phải định nghĩa lại tại lớp dẫn xuất

```
abstract class Shape {
   protected int x, y;

   public void moveTo(int x1, int y1) {
       erase();
       x = x1;
       y = y1;
       draw();
   }

   abstract public void erase();
   abstract public void 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("Erase at (" + x + "," + y + ")");
   }

   public void draw() {
       System.out.println("Draw at (" + x + "," + y + ")");
   }
}
```

Giao diện (Interface)

- Interface là mức trừu tượng cao hơn lớp trừu tượng
- Bao gồm
  - phương thức trừu tượng
  - hằng số (static final)
  - mặc định là public
- Cú pháp:
  - từ khóa `interface` và `implements`

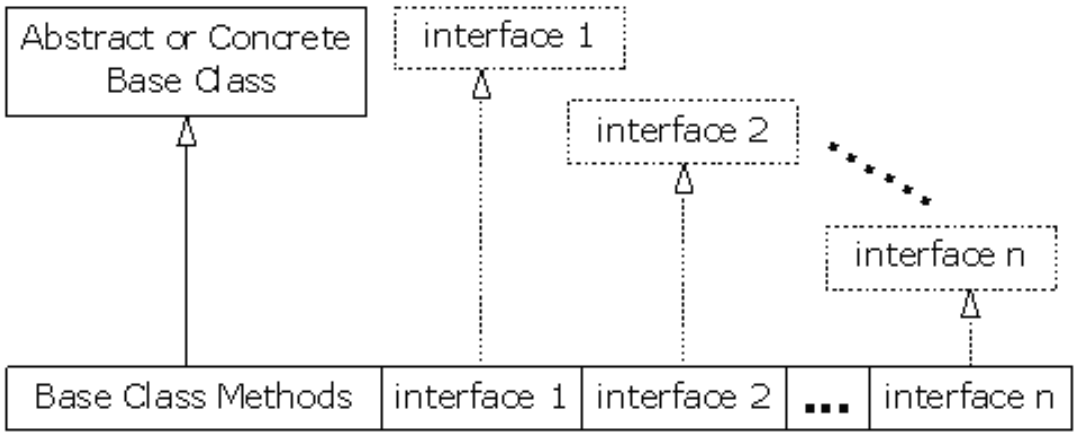
```
interface Action {
   void moveTo(int x, int y);
   void erase();
   void draw();
}

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) {...}
}
```

Lớp trừu tượng cài đặt giao diện

```
abstract class Shape implements Action {
   protected int x, y;

   public Shape() {...}
   public Shape(int _x, int _y) {...}

   public void moveTo(int x1, int y1) {
       erase();
       x = x1;
       y = y1;
       draw();
   }
}
```

Đa kế thừa

- Java không cho phép đa kế thừa từ nhiều lớp cơ sở
  - đảm bảo tính dễ hiểu
  - hạn chế xung đột
- Có thể cài đặt đồng thời nhiều giao diện



```
class ImageBuffer {
...
}

class Animation extends ImageBuffer
  implements Action {
...
    public void erase() {...}
    public void draw() {...}
    public void moveTo() {...}
}
```

```
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 {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
}
```

Xung đột (1)

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
    public int f() { return 1; }
}
class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // overloaded
}
class C3 extends C implements I2 {
    public int f(int i) { return 1; } // overloaded
}
```

Xung đột (2)

```
class C4 extends C implements I3 {
    // Identical, no problem:
    public int f() { return 1; }
}

class C5 extends C implements I1 {}

interface I4 extends I1, I3 {}
```

Mở rộng lớp trừu tượng và giao diện

```
interface I1 {}

interface I2 {}

interface I3 extends I1, I2 {}

abstract class A1 {}

abstract class A2 extends A1 implements I1, I2 {}
```

Abstract class vs. Interface

- Lớp trừu tượng có thể có phương thức thực và thuộc tính
- Interface hỗ trợ đa kế thừa
- Cái gì là bất biến ?

Hướng tới lập trình tổng quát

Generic programming

- Tổng quát hóa chương trình để có thể hoạt động với các kiểu dữ liệu khác nhau, kể cả các kiểu dữ liệu trong tương lai
  - thuật toán đã xác định
- Ví dụ, kiểu ngăn xếp
  - C: dùng con trỏ *void*
  - C++: dùng template
  - Java: lợi dụng upcasting và lớp gốc Object
  - Java 1.5: template

```
class MyStack {
...
    public void push(Object obj) {...}
    public Object pop() {...}
}
...

MyStack s = new MyStack();
Point p = new Point();
Circle c = new Circle();
s.push(p);
s.push(c);
Circle c1 = (Circle) s.pop();
Point p1 = (Point) s.pop();
```

Local copy

- Có nhu cầu sao chép các đối tượng
  - Sao chép khi chuyền tham số để tránh sửa đổi đối tượng gốc
- Làm thế nào để sao chép đối tượng mà không biết rõ kiểu (lớp) thực sự của nó?
  - Sử dụng copy constructor?
  - Sử dụng phương thức copy?
    - `Interface Cloneable` và phương thức `clone()`

Copy constructor

```
class Base {
   public Base() {}
   public Base(Base b) {...}
   public String print() { return "base class"; };
}

class Derived extends Base {
   public Derived() {}
   public Derived(Derived d) {
       super(d);
   }
   public String print() { return "derived class"; };
}
```

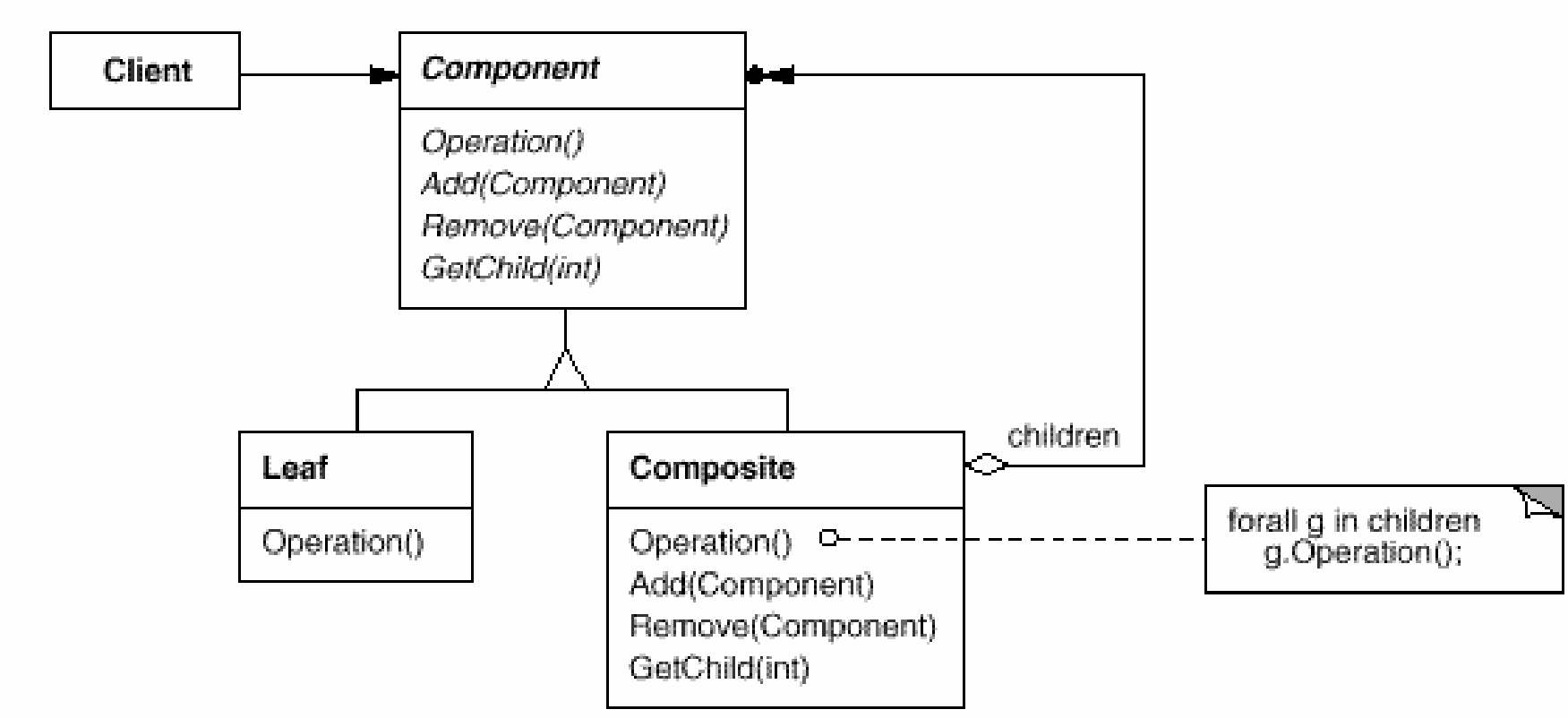
Local copy sử dụng copy constructor

```
public class TestCopy {
   static void copy(Derived d) {
        Derived d1 = new Derived(d);
        System.out.println(d1.print());
   }
   static void copy2(Base b) {
        Base b1 = new Base(b);
        System.out.println(b1.print());
   }
   public static void main(String args[]) {

        Derived d = new Derived();
        copy(d);
        copy2(d);
   }
}
```

Bài tập

- Sử dụng kiến thức về kế thừa và đa hình để thiết kế các lớp sau
  - Các lớp đối tượng hình học cơ sở Point, Circle, Rectange, ...
  - Lớp Graphics là một hình phức hợp do người dùng định nghĩa (là một tập các hình cơ sở hoặc phức hợp khác)
- Yêu cầu: các lớp phải sử dụng giao diện như nhau: move(), draw(),…

Composite design pattern



Xử lý ngoại lệ

Nội dung

- Khái niệm về xử lý ngoại lệ (exception handling)
- Ném và bắt ngoại lệ
- Khai báo ngoại lệ
- Ném lại ngoại lệ
- Định nghĩa ngoại lệ mới
- Xử lý ngoại lệ trong constructor

Tài liệu tham khảo

- *Thinking in Java*, chapter 9
- *Java how to program*, chapter 15

Lỗi và ngoại lệ

- Mọi đoạn chương trình đều tiềm ẩn khả năng sinh lỗi
  - lỗi chủ quan: do lập trình sai
  - lỗi khách quan: do dữ liệu, do trạng thái của hệ thống
- Ngoại lệ: các trường hợp hoạt động không bình thường
- Xử lý ngoại lệ như thế nào
  - làm thế nào để có thể tiếp tục (tái) thực hiện

Cách xử lý lỗi truyền thống

- Cài đặt mã xử lý tại nơi phát sinh ra lỗi
  - làm cho chương trình trở nên khó hiểu
  - không phải lúc nào cũng đầy đủ thông tin để xử lý
  - không nhất thiết phải xử lý
- Truyền trạng thái lên mức trên
  - thông qua tham số, giá trị trả lại hoặc biến tổng thể (flag)
  - dễ nhầm
  - vẫn còn khó hiểu
- Khó kiểm soát được hết các trường hợp
  - lỗi số học, lỗi bộ nhớ,…
- Lập trình viên thường quên không xử lý lỗi
  - bản chất con người
  - thiếu kinh nghiệm, cố tình bỏ qua

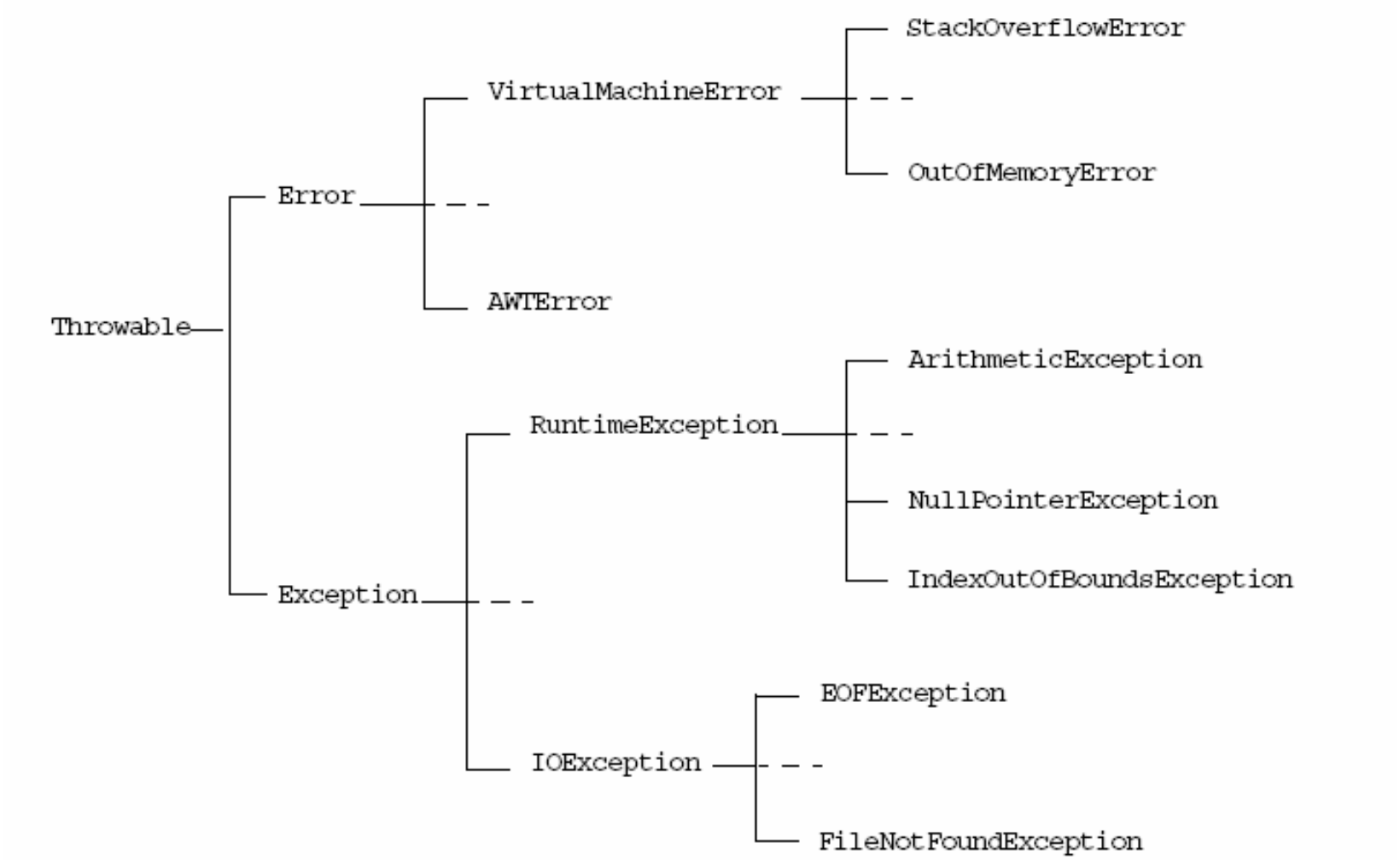
Ví dụ

```
int devide(int num, int denom, int& error)
{
    if (0 != denom) {
        error = 0;
        return num/denom;
    } else {
        error = 1;
        return 0;
    }
}
```

Xử lý ngoại lệ (Exception handling) trong Java

- Xử lý ngoại lệ trong Java được kế thừa từ C++
- Dựa trên cơ chế ném và bắt ngoại lệ
  - ném ngoại lệ: dừng chương trình và chuyển điều khiển lên mức trên (nơi bắt ngoại lệ)
  - bắt ngoại lệ: xử lý với ngoại lệ
- Ngoại lệ: là đối tượng mang thông tin về lỗi đã xảy ra
  - ngoại lệ được ném tự động
  - ngoại lệ được ném tường minh

Phả hệ ngoại lệ trong Java



Ưu điểm của ném bắt ngoại lệ

- Dễ sử dụng
  - dàng chuyển điều khiển đến nơi có khả năng xử lý ngoại lệ
  - có thể ném nhiều loại ngoại lệ
- Tách xử lý ngoại lệ khỏi thuật toán
  - tách mã xử lý
  - sử dụng cú pháp khác
- Không bỏ sót ngoại lệ (ném tự động)
- Làm chương trình dễ đọc hơn, an toàn hơn

Ném ngoại lệ (tường minh)

- Ném ngoại lệ bằng câu lệnh `throw`

```
if (0==denominator) {
    throw new Exception();
} else res = nominator / denominator;
```

Cú pháp `try - catch`

- Việc phân tách đoạn chương trình thông thường và phần xử lý ngoại lệ được thể hiện thông qua cú pháp `try – catch`
  - Khối lệnh `try {…}`: khối lệnh có khả năng ném ngoại lệ
  - Khối lệnh `catch() {…}`: bắt và xử lý với ngoại lệ

```
try {
// throw an exception
}
catch (TypeOfException e) {
    exception-handling statements
}
```

Ví dụ

```
try {
    if (0 == denom) {
        throw new Exception(”denom = 0”);
    } else res = num/denom;
} catch(Exception e) {
    System.out.println(e.getMessage());
}
```

Cú pháp `try catch finally`

- Có thể bắt nhiều loại ngoại lệ khác nhau bằng cách sử dụng nhiều khối lệnh `catch` đặt kế tiếp
  - khối lệnh `catch` sau không thể bắt ngoại lệ là lớp dẫn xuất của ngoại lệ được bắt trong khối lệnh `catch` trước
- Khối lệnh `finally` có thể được đặt cuối cùng để thực hiện các công việc "dọn dẹp" cần thiết
  - `finally` luôn được thực hiện dù ngoại lệ có được bắt hay không
  - `finally` được thực hiện cả khi không có ngoại lệ được ném ra

Cú pháp try catch finally

```
try {
…
}
catch(Exception1 e1) {
…
}
catch(Exception2 e2) {
…
}
finally {
…
}
```

Ví dụ

```
...
try {
   str = buf.readLine();
   num = Integer.valueOf(str).intValue();
}
catch (IOException e) {
   System.err.println("IO Exception");
}
catch (NumberFormatException e) {
   System.err.println("NumberFormatException");
}
catch(Exception e) {
   System.err.println(e.getMessage());
}
finally {
   buf.close();
}
```

Ném ngoại lệ khỏi phương thức

- Không nhất thiết phải xử lý ngoại lệ trong phương thức
  - không đủ thông tin để xử lý
  - không đủ thẩm quyền
- Một phương thức muốn ném ngoại lệ ra ngoài phải khai báo việc ném ngoại lệ bằng từ khóa `throws`
  - có thể ném ngoại lệ thuộc lớp dẫn xuất của ngoại lệ được khai báo

Ví dụ

```
int readInt() throws IOException,
  NumberFormatException {
...
     str = buf.readLine();
     return Integer.valueOf(str).intValue();
}
```

Ví dụ

```
try {
    int n = readInt();
}
catch (IOException e) {
  System.err.println("IO Exception");
}
catch (NumberFormatException e) {
  System.err.println("NumberFormatException");
}
```

Ngoại lệ và phương thức được định nghĩa lại

- Phương thức được định nghĩa lại tại lớp dẫn xuất có thể không ném ngoại lệ
- Nếu ném ngoại lệ, chỉ có thể ném ngoại lệ giống như tại phương thức của lớp cơ sở hoặc ngoại lệ là lớp dẫn xuất của ngoại lệ được ném tại phương thức của lớp cơ sở
  - đảm bảo bắt được ngoại lệ khi sử dụng cơ chế đa hình

Ví dụ

```
class A {
   public void methodA() throws RuntimeException {
   }
}

class B extends A {
   public void methodA() throws ArithmeticException {
   }
}

class C extends A {
   public void methodA() throws Exception {
   }
}

class D extends A {
    public void methodA() {
   }
}
```

Ví dụ:

```
A a = new B();
try {
    a.methodA();
}
catch (RuntimeException e) {
...
}
```

Ném lại ngoại lệ

- Sau khi bắt ngoại lệ, nếu thấy cần thiết chúng ta có thể ném lại chính ngoại lệ vừa bắt được để cho chương trình mức trên tiếp tục xử lý

```
try {...
}
catch (Exception e) {
    System.out.println(e.getMessage());
    throw e;
}
```

Lần vết ngoại lệ `StackTrace`

- Có thể sử dụng phương thức `printStackTrace()` để lần vết vị trí phát sinh ngoại lệ
  - debug chương trình

```
public class Test4 {
   void methodA() throws Exception {
        methodB();
        throw new Exception();
   }
   void methodB() throws Exception {
        methodC();
        throw new Exception();
   }
   void methodC() throws Exception {
        throw new Exception();
   }
   public static void main(String[] args) {
        Test4 t = new Test4();
        try {
               t.methodA();
        }
        catch(Exception e) {
               e.printStackTrace();
        }
   }
}
```

Ném ngoại lệ từ `main()`

- Nếu không có phương thức nào bắt ngoại lệ, ngoại lệ sẽ được truyền lên phương thức main() và được cần được xử lý tại đây.
- Nếu vẫn không muốn xử lý ngoại lệ, chúng ta có thể để ngoại lệ truyền lên mức điều khiển của máy ảo bằng cách khai báo main() ném ngoại lệ
  - chương trình sẽ bị dừng và hệ thống sẽ in thông tin về ngoại lệ trên Console (`printStackTrace()`)

Ví dụ

```
import java.io.*;

public class Echo {
    public static void main(String[] args)
    throws IOException {
        InputStreamReader reader;
        BufferedReader bufReader;
        reader = new InputStreamReader(System.in);
        bufReader = new BufferedReader(reader);
        String s;
        while( null != (s = bufReader.readLine())
            System.out.println(s);
    }
}
```

Hai loại ngoại lệ

- Java phân biệt hai loại ngoại lệ là ngoại lệ cần kiểm tra và ngoại lệ không cần kiểm tra
- Ngoại lệ cần kiểm tra: chương trình dịch luôn kiểm tra xem chúng ta có viết code xử lý với các ngoại lệ này không (`try catch/ throws`)
  - IOException
- Ngoại lệ không cần kiểm tra: các ngoại lệ có thể loại trừ nếu viết chương trình tốt hơn
  - RuntimeException

Ví dụ: Checked Exception

```
InputStreamReader reader;
BufferedReader bufReader;
reader = new InputStreamReader(System.in);
bufReader = new BufferedReader(reader);

try {
        String s = bufReader.readLine();
}
catch (IOException e) {
...
}
```

Ví dụ: Unchecked Exception

```
int num1 = Integer.ValueOf(str1).intValue();
int num2 = Integer.ValueOf(str2).intValue();
int num3 = num1 / num2;
```

- Hầu hết các ngoại lệ thuộc lớp RuntimeException được hệ thống ném tự động
  - lỗi số học
  - lỗi chỉ số

Hoán đổi ngoại lệ

- Có thể đổi ngoại lệ cần kiểm tra thành ngoại lệ không cần kiểm tra
  - *chưa biết nên làm gì*

```
void wrapException() {
    try {
        throw new IOException();
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

```
try {
    wrapException();
} catch (RuntimeException e) {
    try {
        throw e.getCause();
    }
    catch (IOException e1) {
    …
    }
}
```

Tự định nghĩa ngoại lệ

- Chúng ta có thể tạo lớp ngoại lệ để phục vụ các mục đích riêng
- Lớp ngoại lệ mới phải kế thừa từ lớp Exception hoặc lớp dẫn xuất của lớp này
- Có thể cung cấp hai constructor
  - constructor mặc định (không tham số)
  - constructor nhận một tham số String và truyền tham số này cho phương thức khởi tạo của lớp cơ sở

Ví dụ

```
class SimpleException extends Exception {
}

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}
```

Khởi tạo đối tượng và xử lý ngoại lệ

- Làm thế nào để thông báo khi hàm khởi tạo đối tượng gặp lỗi
  - không có giá trị trả lại
- Một cách là khởi tạo với một trạng thái đặc biệt và hi vọng sẽ có mã chương trình kiểm tra trạng thái này
- Cách hợp lý hơn là ném ngoại lệ

```
class InputFile {
    public InputFile(String fname) throws IOException {
    ...
    }
...
}

---
try {
    InputFile fin = new InputFile(”data.txt”);
}
catch (IOException e) {
    System.err.println(e.getMessage);
}
```

Bài tập và thực hành

- Tìm hiểu về phả hệ ngoại lệ của Java
- Thực hành
  - ném và bắt ngoại lệ
  - khai báo phương thức ném ngoại lệ
  - constructor ném ngoại lệ
  - tự định nghĩa ngoại lệ

Các luồng vào / ra

Nội dung

- Khái niệm về luồng dữ liệu
- Luồng và tệp
- Lớp File
- Truy cập tệp tuần tự
- Truy cập tệp ngẫu nhiên

Tài liệu tham khảo

- *Thinking in Java*, chapter 12
- *Java how to program*, chapter 17

Luồng dữ liệu (data streams)

- Chương trình Java nhận và gửi dữ liệu thông qua các đối tượng là các thực thể thuộc một kiểu luồng dữ liệu nào đó
- Luồng (stream) là một dòng dữ liệu đến từ một nguồn (source) hoặc đi đến một đích (sink)
- Nguồn và đích có thể là tệp (file), bộ nhớ, một tiến trình (process), hay thiết bị (bàn phím, màn hình, …)

Luồng byte và char

- Luồng byte: thao tác theo đơn vị byte
  - InputStream
  - OutputStream
- Luồng char: thao tác với ký tự
  - Reader
  - Writer

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()`

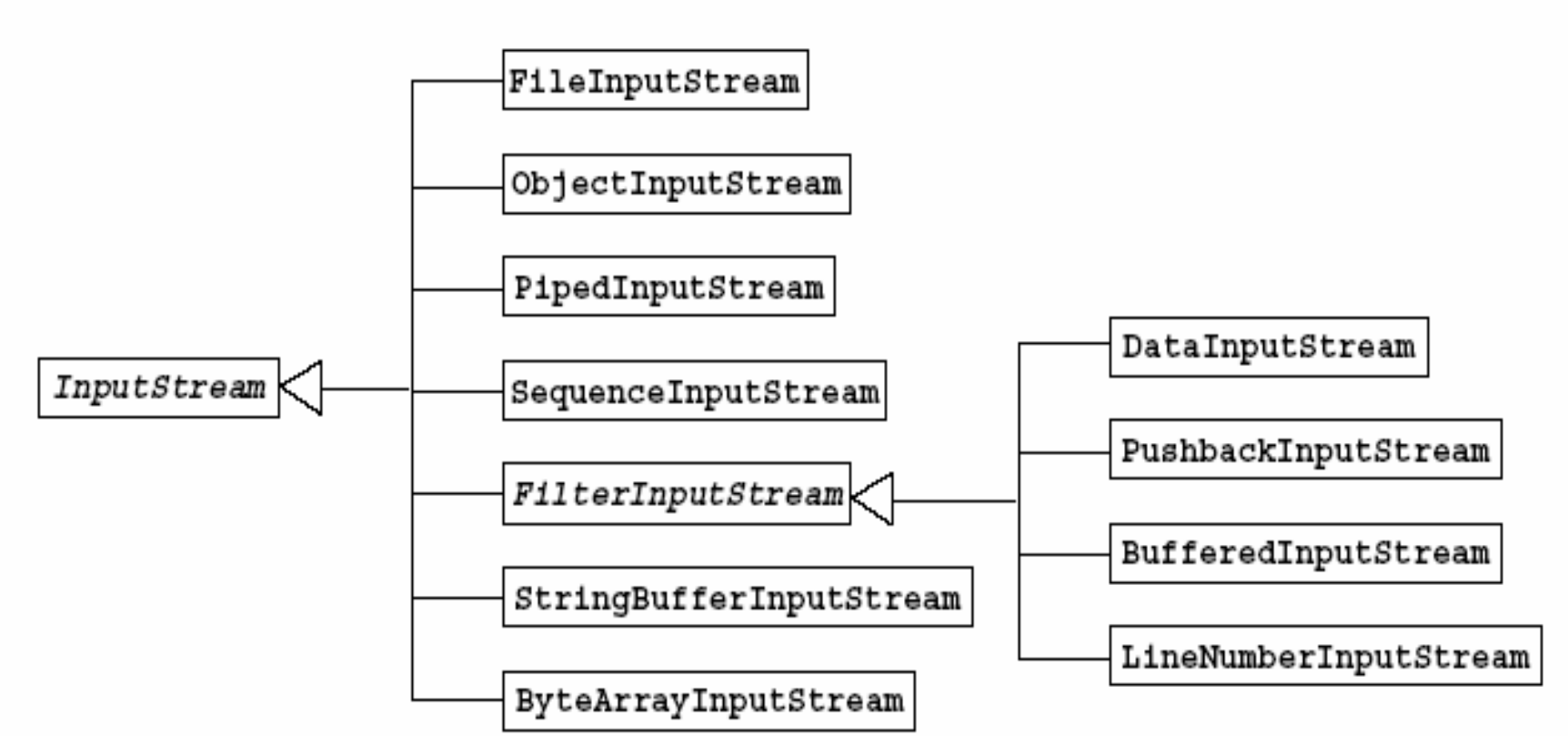
Reader

- `int read()`
- `int read(char buf[])`
- `int read(char buf[], int offset, int length)`
- `void close()`

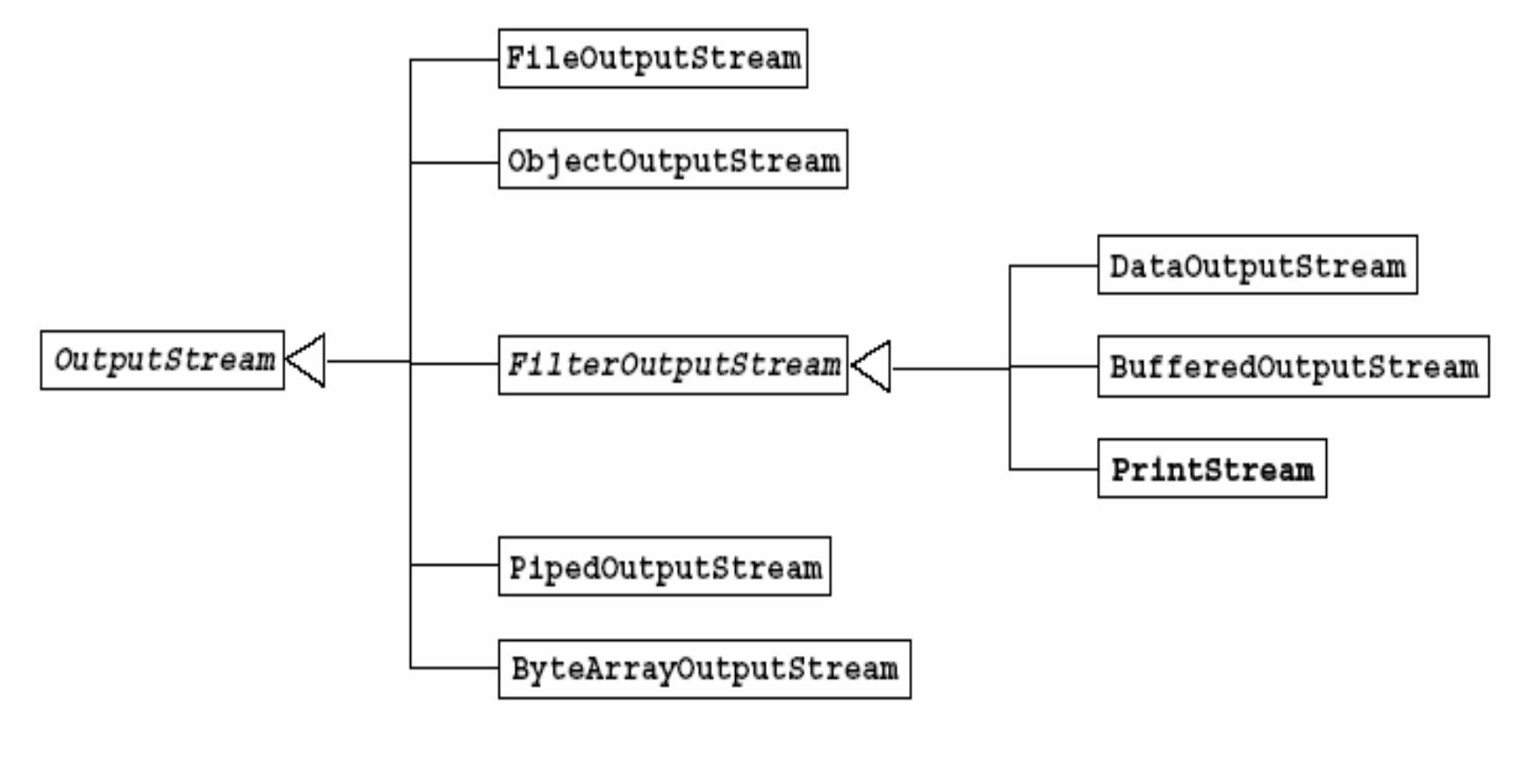
Writer

- `int write(int c)`
- `int write(char buf[])`
- `int write(char buf[], int offset, int length)`
- `void close()`
- `void flush()`

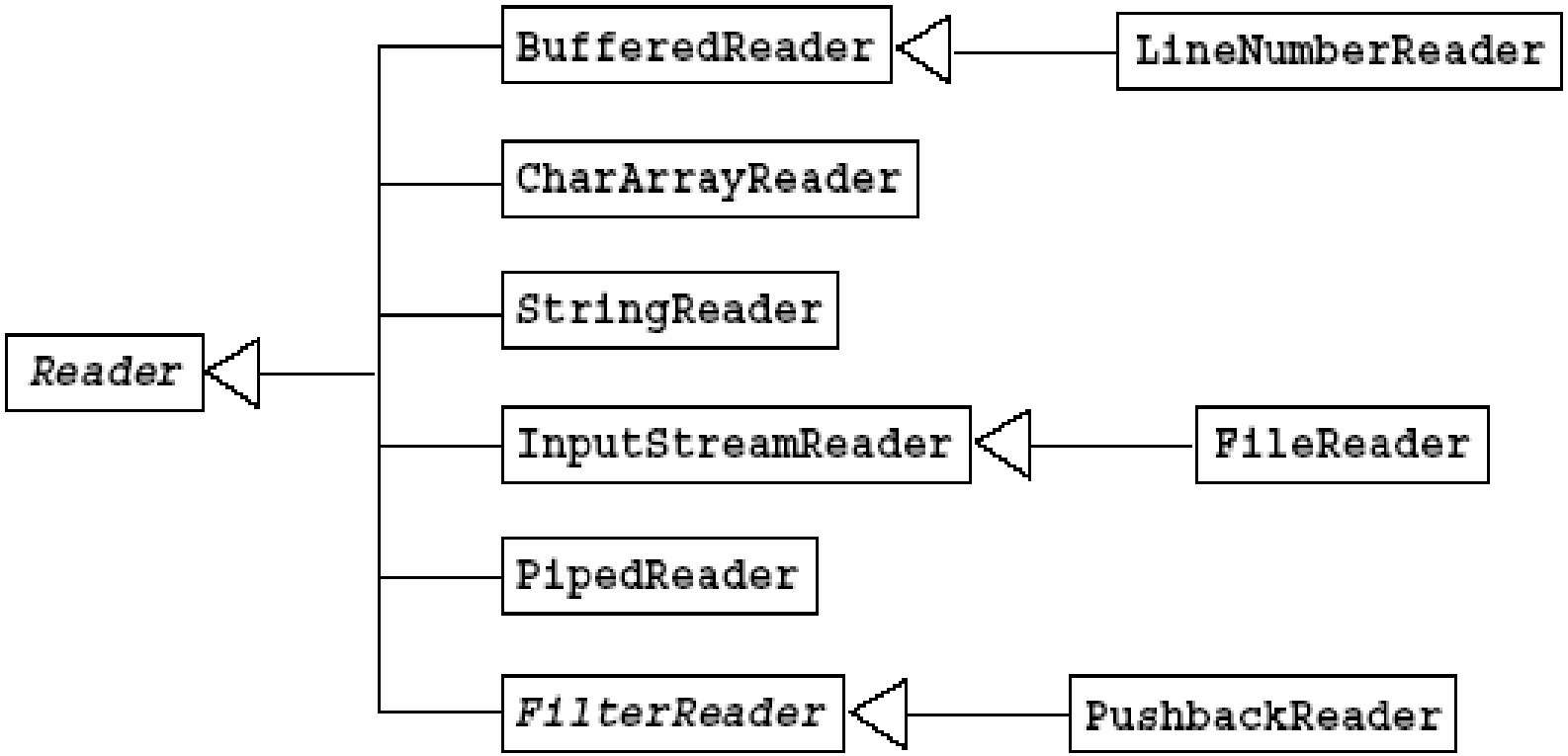
Phả hệ của InputStream



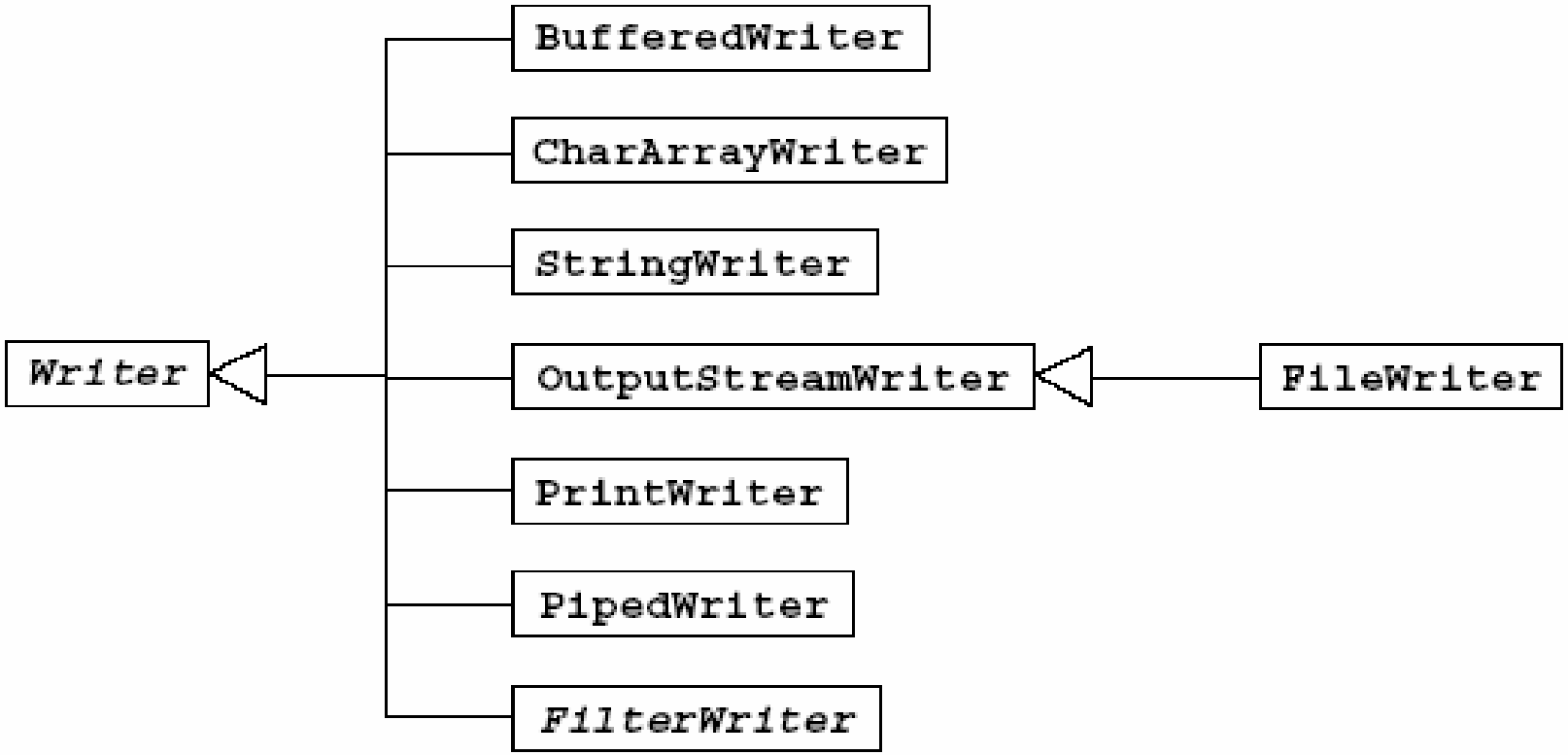
Phả hệ của OutputStream



Phả hệ của Reader



Phả hệ của Writer



Đối tượng vào / ra

- 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 khối
  - 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 khối
  - 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 khối từ thiết bị

Nhập xuất qua thiết bị chuẩn
Console I/O

- System.out cho phép in ra luồng ra chuẩn
  - là đối tượng của lớp `PrintStream`
- System.err cho phép in ra luồng thông báo lỗi chuẩn
  - là đối tượng của lớp `PrintStream`
- System.in cho phép đọc vào từ thiết bị vào chuẩn
  - là đối tượng của lớp `InputStream`

Đọc dữ liệu từ luồng vào chuẩn

- System.in không sử dụng được trực tiếp
- Chúng ta muốn đọc một dòng ký tự
  1. tạo đối tượng luồng ký tự (InputStreamReader)
  2. tạo đối tượng luồng có bộ đệm (BufferedReader)

Ví dụ:

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(reader);
---

String s;
try {
    s = in.readLine();
}
catch (IOException e) {...}
```

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
  - nằm trong gói java.io
  - tạo tệp, mở tệp, các thông tin về tệp và thư mục

Tạo đối tượng File

- File myFile;
- myFile = new File(“data.txt”);
- myFile = new File(“myDocs”, “data.txt”);
- Thư mục cũng được coi như là một tệp
  - File myDir = new File(“myDocs”);
  - File myFile = new File(myDir, “data.txt”);
  - có phương thức riêng để thao tác với thư mục

Các phương thức

- Tên tệp
  - `String getName()`
  - `String getPath()`
  - `String getAbsolutePath()`
  - `String getParent()`
  - `boolean renameTo(File newName)`
- Kiểm tra tệp
  - `boolean exists()`
  - `boolean canWrite()`
  - `boolean canRead()`
  - `boolean isFile()`
  - `boolean isDirectory()`
  - `boolean isAbsolute()`

Các phương thức (2)

- Nhận thông tin
  - **`long lastModified()`**
  - **`long length()`**
  - **`boolean delete()`**
- Thư mục
  - **`boolean mkdir()`**
  - **`String[] list()`**

Thao tác với tệp ký tự

- Đọc từ tệp
  - FileReader: đọc ký tự từ tệp
  - BufferedReader: đọc có bộ đệm (đọc từng dòng readLine())
- Ghi ra tệp
  - FileWriter: ghi ký tự ra tệp
  - PrintWriter: ghi theo dòng (print() và println())

Ví dụ: Đọc vào từ tệp

```
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) {
}
```

Ví dụ: Đọc vào (cont.)

```
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();
```

Ví dụ: Đọc vào (cont.)

```
class Abc {
  public void read(BufferedReader in) {
      String s;
      try {
            s = in.readLine();
            ...
      }
      catch (IOException e) {…}
  }
  public void doSomething() {...}
...
}
```

Ví dụ: Ghi ra tệp

```
File file = new File(”data.out”);
FileWriter writer = new FileWriter(file);
PrintWriter out = new PrintWriter(writer);
String s = ”Hello”;
try {
    out.println(s);
    out.close();
}
catch (IOException e) {
}
```

Ví dụ: Ghi ra (cont.)

```
class Abc {
...
    public void write(PrintStream out) {
        ...
        try {
            out.println(s);
            out.close();
        }
        catch (IOException e) {...}
    }
}
```

Ví dụ: Ghi ra (cont.)

```
class Abc {
...
    public String write() {
        String buf;
        buf += ...
        return buf;
    }
}
```

Ví dụ: File copy

```
import java.io.*;

public class CopyFile {
    public static void main(String args[]) {
          try {
                    FileReader src = new FileReader(args[0]);
                    BufferedReader in = new BufferedReader(src);
                    FileWriter des = new FileWriter(args[1]);
                    PrintWriter out = new PrintWriter(des);
                    String s;

                    s = in.readLine();
                    while (s != null) {
                              out.println(s);
                              s = in.readLine();
                    }

                    in.close();
                    out.close();
          }
          catch (IOException e) {
                    e.printStackTrace();
          }
    }
}
```

Ví dụ: File copy (2)

```
import java.io.*;

public class CopyFile2 {
   public static void main(String args[]) {
        try {
                FileReader src = new FileReader(args[0]);
                FileWriter des = new FileWriter(args[1]);
                char buf[] = new char[128];
                int charsRead;
                charsRead = src.read(buf);
                while (charsRead != -1) {
                        des.write(buf, 0, charsRead);
                        charsRead = src.read(buf);
                }
                src.close();
                des.close();
        }
        catch (IOException e) {
                e.printStackTrace();
        }
   }
}
```

Thao tác với tệp dữ liệu (tuần tự)

- Đọc dữ liệu
  - FileInputStream: đọc dữ liệu từ tệp
  - DataInputStream: đọc dữ liệu kiểu nguyên thủy
  - ObjectInputStream: đọc đối tượng
- Ghi dữ liệu
  - FileOutputStream: ghi dữ liệu ra tệp
  - DataOutputStream: ghi dữ liệu kiểu nguyên thủy
  - ObjectOutputStream: ghi đối tượng

DataInputStream/DataOutputStream

- DataInputStream: đọc các dữ liệu nguyên thủy
  - readBoolean, readByte, readChar, readShort, readInt, readLong, readFloat, readDouble
- DataOutputStream: ghi các dữ liệu nguyên thủy
  - writeBoolean, writeByte, writeChar, writeShort, writeInt, writeLong, writeFloat, writeDouble

Ghi dữ liệu nguyên thủy (tuần tự)

```
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();
        }
   }
}
```

Đọc dữ liệu nguyên thủy (tuần tự)

```
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();
         }
   }
}
```

Đối tượng *"tuần tự"*

- Một đối tượng có thể được lưu trong bộ nhớ tại nhiều vùng nhớ khác nhau
  - các thuộc tính không phải là kiểu nguyên thủy
- Đối tượng muốn ghi / đọc được phải thuộc lớp có cài đặt giao diện Serializable
  - đây là giao diện *nhãn*, không có phương thức

```
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;
   }
}
```

```
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();
        }
   }
}
```

```
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();
        }
    }
}
```

Lớp RandomAccessFile

- Là một lớp độc lập (kế thừa trực tiếp từ Object)
- Đảm nhận việc đọc và ghi dữ liệu ngẫu nhiên
  - cài đặt các giao diện DataInput và DataOutput
- Kích thước bản ghi phải cố định

```
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();
        }
   }
}
```

```
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 (từ Java 1.5)
- Khởi tạo với đối số là đối tượng vào (luồng, tệp, xâu ký tự)
- Có các phương thức hỗ trợ nhập trực tiếp
  - next*Type*, hasNext*Type*

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();

---
Scanner sc;
sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
   long aLong = sc.nextLong();
}
--
String str = ”2 3 5 7”;
Scanner sc = new Scanner(str);
while (sc.hasNextInt()) {
   System.out.println(sc.nextInt());
}
```

```
class Abc {
  public void read(Scanner sc) {
  ...
  }
  public void doSomething() {...}
...
}
```

Nguyên lý thiết kế và mẫu thiết kế

Nội dung

- Thiết kế module
- Chất lượng thiết kế
- Độ đo thiết kế tốt
- Khái niệm về mẫu thiết kế

Tài liệu tham khảo

- Bruce Eckel, *Thinking in Patterns*
- Erich Gamma, *Design Patterns – Elements of Reusable Object-Oriented Software*

Thiết kế module

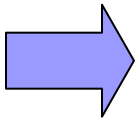
Dựa trên quan điểm "chia để trị"

C: độ phức tạp

E: nỗ lực thực hiện

$C(p1 + p2) > C(p1) + C(p2)$

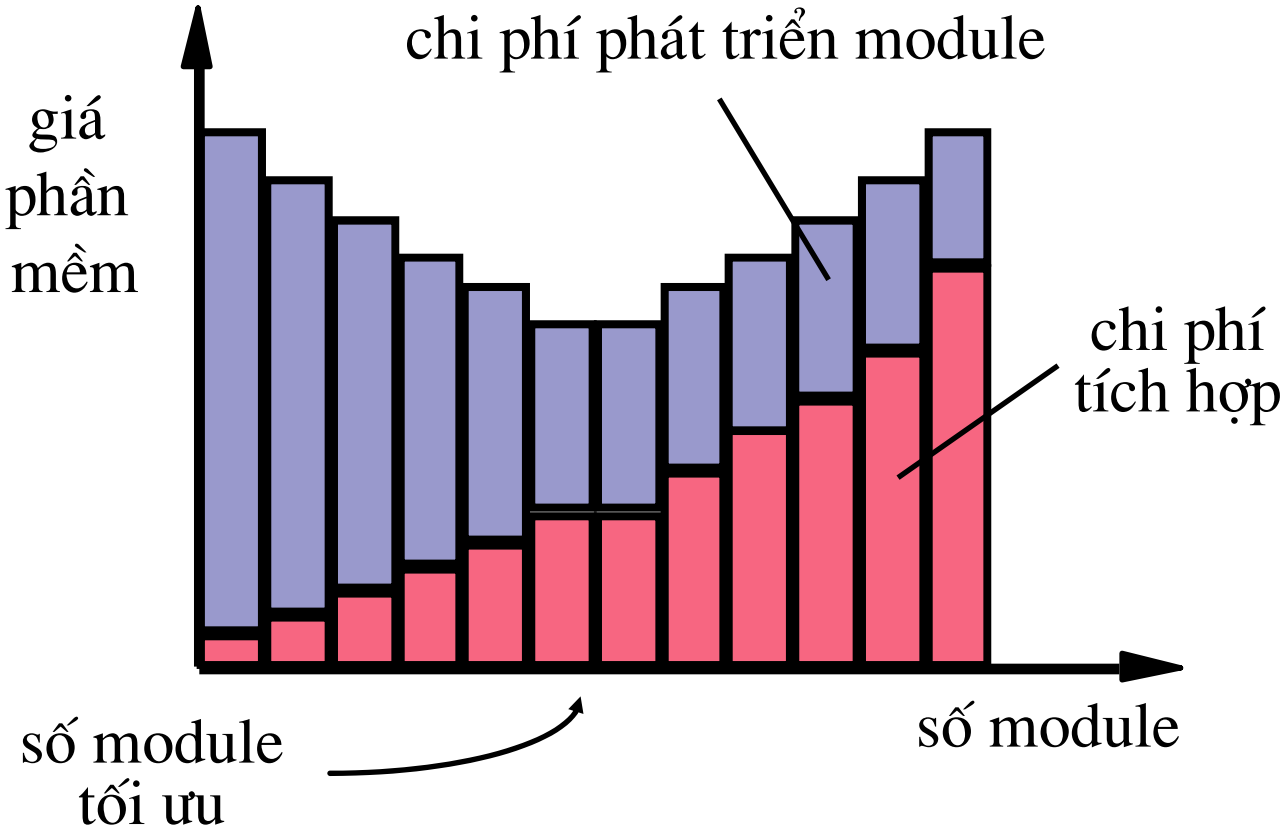
$E(p1 + p2) > E(p1) + E(p2)$



- giảm độ phức tạp
- cục bộ, dễ sửa đổi
- có khả năng phát triển song song
- dễ sửa đổi, dễ hiểu nên dễ tái sử dụng

Số lượng module

Cần xác định số môđun tối ưu



Chất lượng = Che giấu thông tin

- Sử dụng module thông qua các *giao diện*
  - tham số và giá trị trả lại
- Không cần biết cách thức cài đặt thực tế
  - thuật toán
  - cấu trúc dữ liệu
  - giao diện ngoại lai (các mô đun thứ cấp, thiết bị vào/ra)
  - tài nguyên hệ thống

Che giấu thông tin: lý do

- Giảm hiệu ứng phụ khi sửa đổi module
- Giảm sự tác động của thiết kế tổng thể lên thiết kế cục bộ
- Nhấn mạnh việc trao đổi thông tin thông qua giao diện
- Loại bỏ việc sử dụng dữ liệu dùng chung
- Hướng tới sự đóng gói chức năng - thuộc tính của thiết kế tốt

Tạo ra các sản phẩm phần mềm tốt hơn

Chất lượng thiết kế

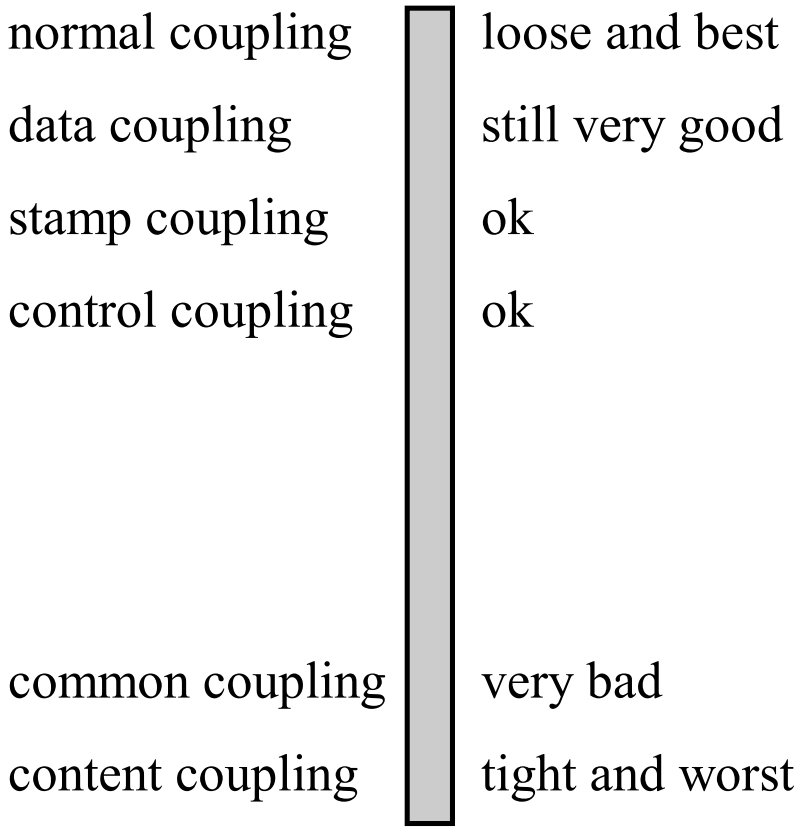
- Phụ thuộc bài toán, không có phương pháp tổng quát
- Một số độ đo
  - Coupling: mức độ ghép nối giữa các module
  - Cohesion: mức độ liên quan lẫn nhau của các thành phần bên trong một module
  - Understandability: tính hiểu được
  - Adaptability: tính thích nghi được

Coupling and Cohesion

- Coupling (ghép nối)
  - độ đo sự liên kết (trao đổi dữ liệu) giữa các mô đun
  - ghép nối chặt chẽ thì khó hiểu, khó sửa đổi (thiết kết tồi)
- Cohesion (kết dính)
  - độ đo sự phụ thuộc lẫn nhau của các thành phần trong một module
  - kết dính cao thì tính cục bộ cao (độc lập chức năng); dễ hiểu, dễ sửa đổi

Coupling

- mức độ quan hệ của các module
- module nên ghép nối lỏng lẻo
- càng lỏng lẻo càng dễ sửa đổi thiết kế

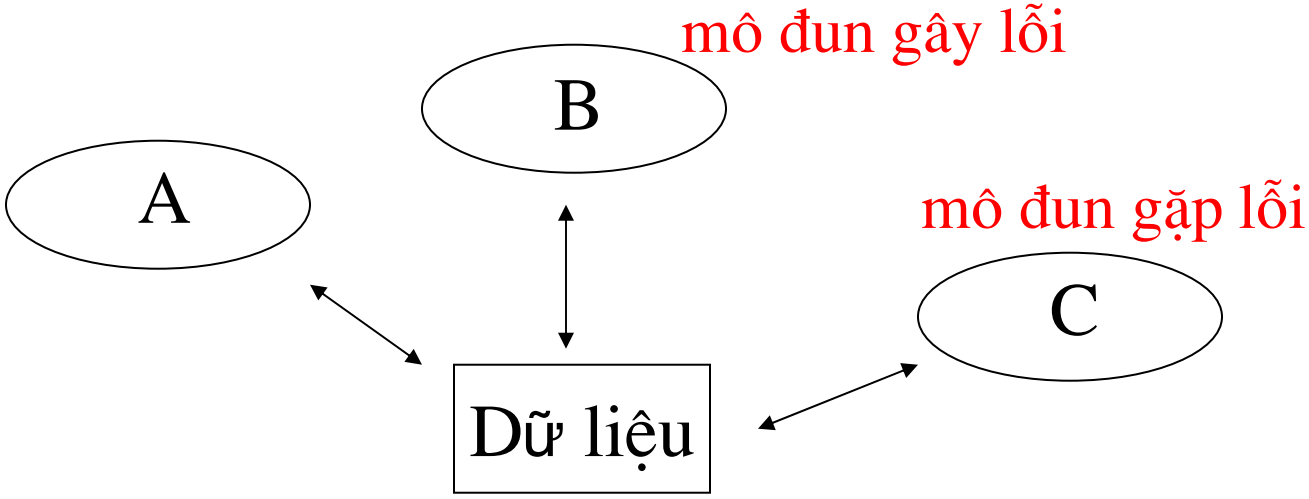


Ghép nối nội dung (content coupling)

- Là trường hợp xấu nhất
- Các module dùng lẫn dữ liệu của nhau
  - các ngôn ngữ bậc thấp không có biến cục bộ
  - lạm dụng lệnh Goto

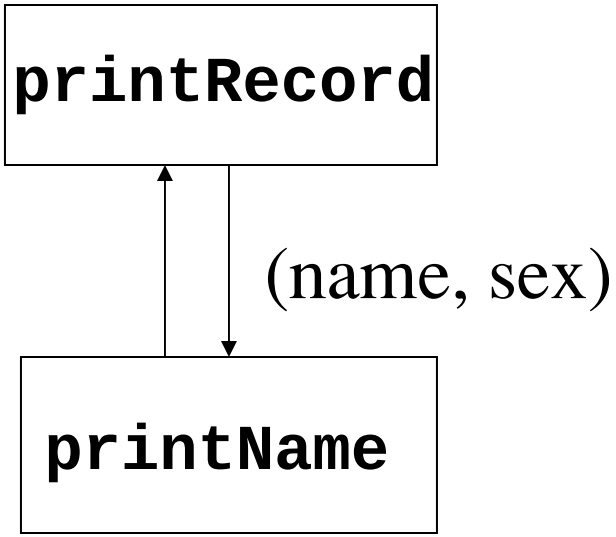
Ghép nối chung (common coupling)

- Các module trao đổi dữ liệu thông qua biến tổng thể
- Lỗi của module này có thể ảnh hưởng đến hoạt động của module khác
- Khó sử dụng lại các module



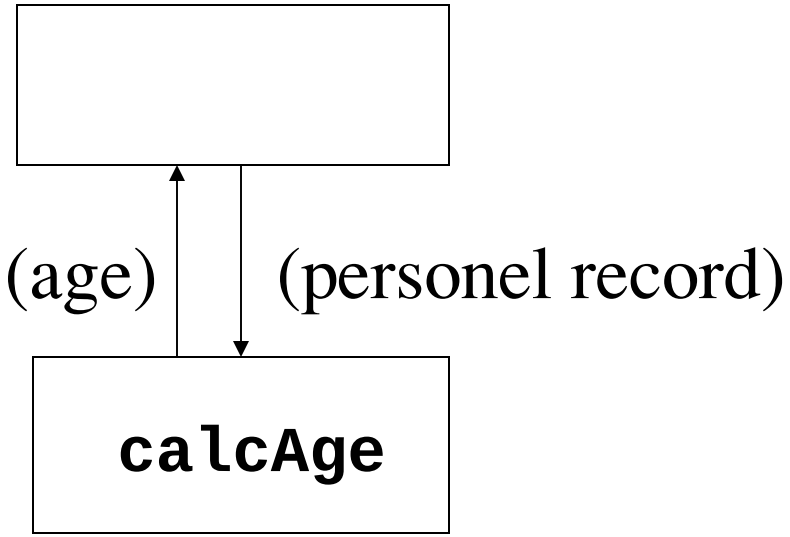
Ghép nối điều khiển (control coupling)

- Các module trao đổi thông tin điều khiển
- Làm cho thiết kế khó hiểu, khó sửa đổi, dễ nhầm



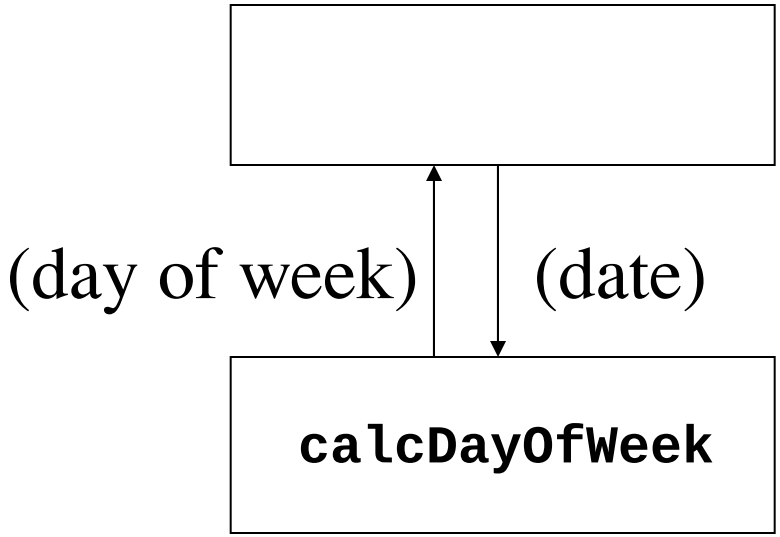
Ghép nối nhãn (stamp coupling)

- Các module trao đổi thừa thông tin
- Module có thể thực hiện chức năng ngoài ý muốn
- Làm giảm tính thích nghi



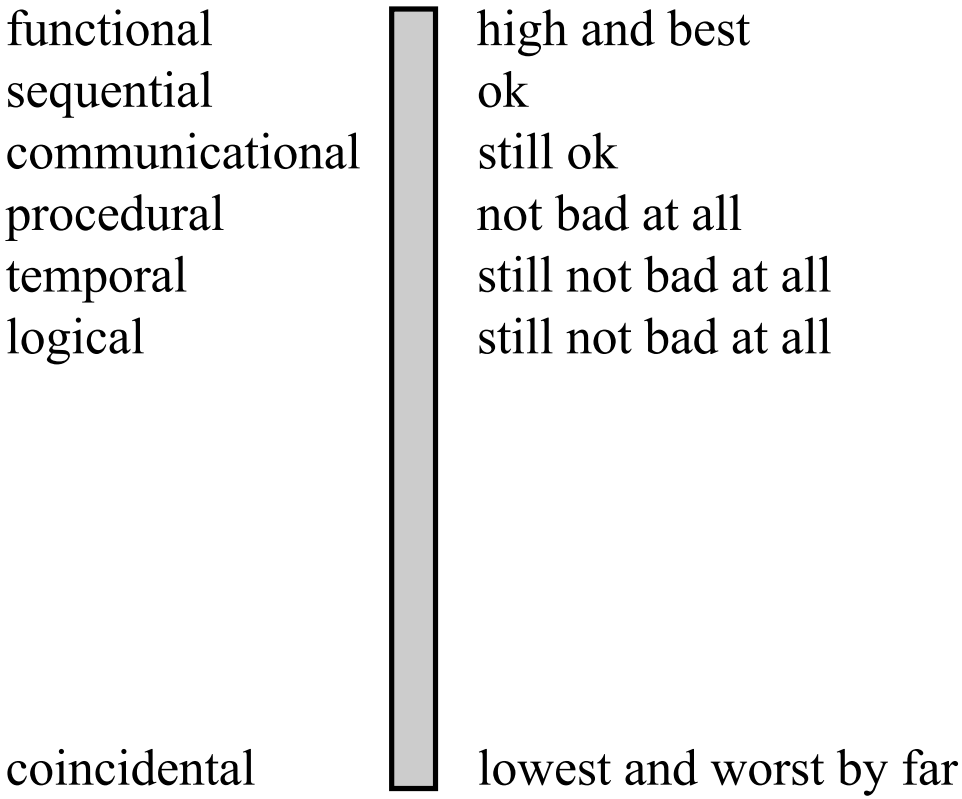
Ghép nối dữ liệu (data coupling)

- Truyền dữ liệu qua tham số
- Nhận kết quả qua tham số và giá trị trả lại



Cohesion

- mỗi module chỉ nên thực hiện một chức năng
- mọi thành phần nên tham gia thực hiện chức năng đó



Các chủng loại kết dính

- Kết dính gom góp (coincidental cohesion)
  - các thành phần không liên quan đến nhau
- Kết dính lô gic (logical cohesion)
  - các thành phần làm chức năng lô gic tương tự
  - vd: hàm xử lý lỗi chung
- Kết dính thời điểm (temporal cohesion)
  - các thành phần hoạt động cùng thời điểm
  - vd: hàm khởi tạo (đọc dữ liệu, cấp phát bộ nhớ...)

Các chủng loại kết dính

- Kết dính thủ tục (procedural cohesion)
  - các thành phần tạo có một thứ tự xác định
  - vd: tính lương cơ bản, tính phụ cấp, tính bảo hiểm

- Kết dính truyền thông (communicational cohesion)
  - các thành phần truy cập cùng dữ liệu
  - vd: thống kê (tính max, min, mean, variation...)

Các chủng loại kết dính

- Kết dính tuần tự (sequential cohesion)
  - output của một thành phần là input của thành phần tiếp theo
  - vd: ảnh mầu -> đen trắng -> ảnh nén
- Kết dính chức năng (functional cohesion)
  - các thành phần cùng góp phần thực hiện một chức năng
  - vd: sắp xếp

Understandability

Tính hiểu được

- Ghép nối lỏng lẻo
- Kết dính cao
- Được lập tài liệu
- Thuật toán, cấu trúc dễ hiểu

Thiết kế hướng đối tượng

- Thiết kế hướng đối tượng hướng tới chất lượng thiết kế tốt
  - đóng gói, che dấu thông tin
  - là các thực thể hoạt động độc lập
  - trao đổi dữ liệu qua thông điệp
  - có khả năng kế thừa
  - cục bộ, dễ hiểu, dễ tái sử dụng

Adaptability
Tính thích nghi được

- Hiểu được
  - sửa đổi được, tái sử dụng được
- Tự chứa
  - không sử dụng thư viện ngoài
  - *mâu thuẫn với xu hướng tái sử dụng*

Adaptability (2)

- Các chức năng cần được thiết kế sao cho dễ dàng mở rộng mà không cần sửa các mã đã có (Open closed principle)
- Trừu tượng hóa là chìa khóa để giải quyết vấn đề này
  - các chức năng trừu tượng hóa thường bất biến
  - các lớp dẫn xuất cài đặt các giải pháp cụ thể
  - sử dụng đa hình
- Mẫu thiết kế: là thiết kế chuẩn cho các bài toán thường gặp

Mẫu thiết kế (Design Patterns)

- **Creational** - Thay thế cho khởi tạo tường minh, ngăn ngừa phụ thuộc môi trường (platform)
- **Structural** - thao tác với các lớp không thay đổi được, giảm độ ghép nối và cung cấp các giải pháp thay thế kế thừa
- **Behavioral** - Che dấu cài đặt, che dấu thuật toán, cho phép thay đổi động cấu hình của đối tượng

Abstract Factory

- Một chương trình cần có khả năng chọn một trong một vài họ các lớp đối tượng
- Ví dụ, giao diện đồ họa nên chạy được trên một vài môi trường
- Mỗi môi trường (platform) cung cấp một tập các lớp đồ họa riêng:

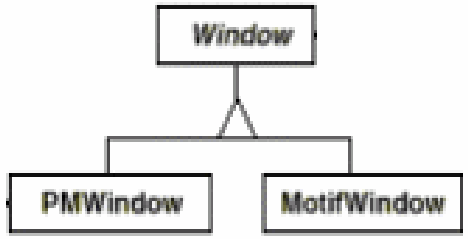
  WinButton, WinScrollBar, WinWindow

  MotifButton, MotifScrollBar, MotifWindow

  pmButton, pmScrollBar, pmWindow

Yêu cầu

- Thống nhất thao tác với mọi đối tượng: button, window,...
  - Dễ dàng - định nghĩa giao diện (interfaces):



- Thống nhất cách thức tạo đối tượng
- Dễ dàng thay đổi các họ lớp đối tượng
- Dễ dàng thêm họ mới

Giải pháp

- Định nghĩa Factory - lớp để tạo đối tượng:

```
class WidgetFactory {
   Button makeButton(args) = 0;
   Window makeWindow(args) = 0;
   // other widgets…
}
```

Giải pháp (tt)

- Định nghĩa Factory chi tiết cho từng họ lớp đối tượng:

```
class WinWidgetFactory extends WidgetFactory
{
    public Button makeButton(args) {
      return new WinButton(args);
    }
    public Window makeWindow(args) {
      return new WinWindow(args);
    }
}
```

Giải pháp (tt)

- Chọn họ lớp muốn dùng:

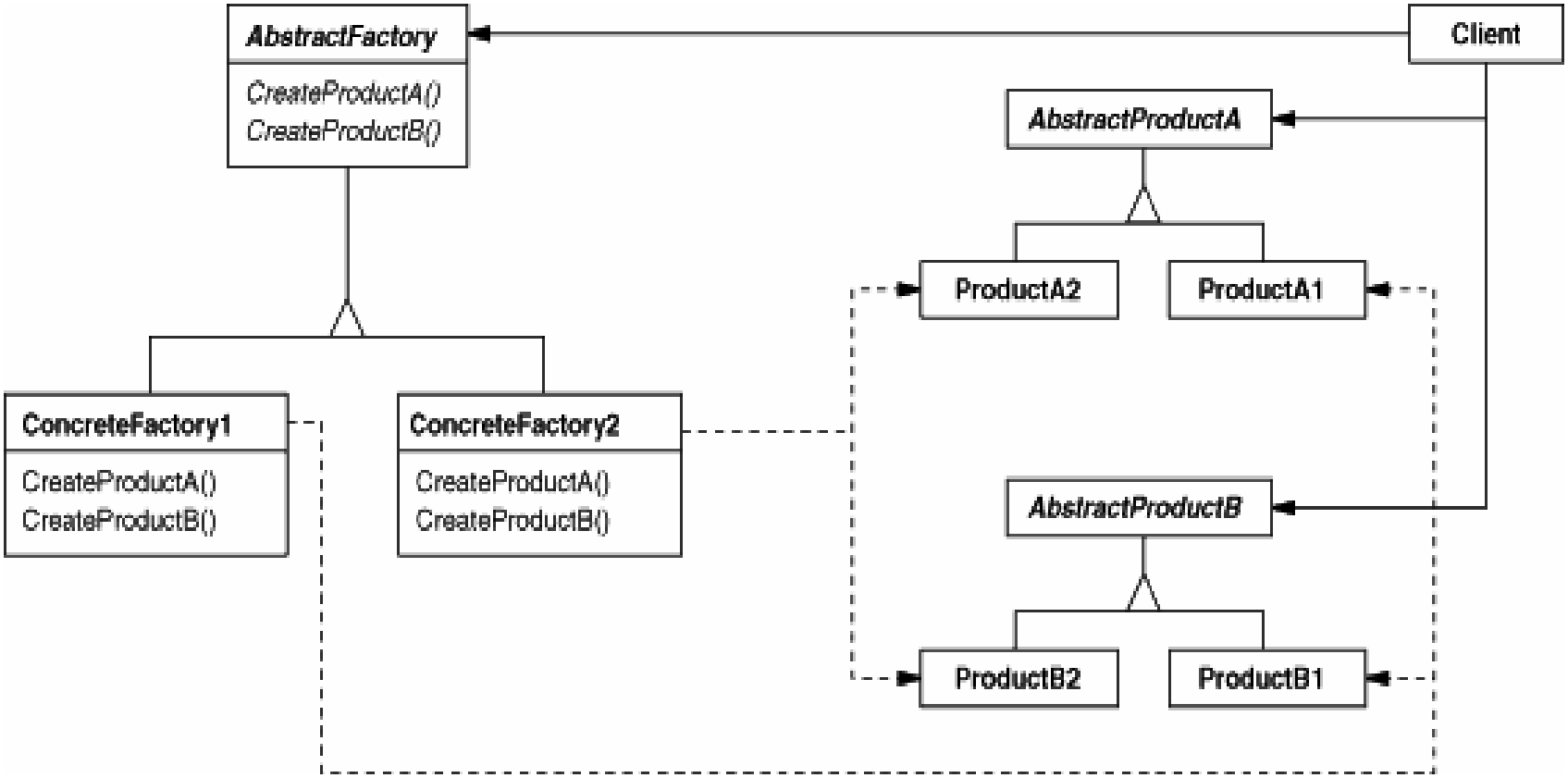
```
WidgetFactory wf =
   new WinWidgetFactory();
```

- Khi khởi tạo đối tượng, không dùng "*new*" mà gọi:

```
Button b = wf.makeButton(args);
```

- Thay đổi họ đối tượng - chỉ một lần trong mã cài đặt!
- Thêm họ - thêm một factory, không ảnh hưởng tới mã đang tồn tại!

Sơ đồ lớp


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

Ứng dụng

- Các hệ điều hành khác nhau
- Các chuẩn look-and-feel khác nhau
- Các giao thức truyền thông khác nhau

Composite

- Một chương trình cần thao tác với các đối tượng dù là đơn giản hay phức tạp một cách thống nhất
- Ví dụ, chương trình vẽ hình chứa đồng thời các đối tượng đơn giản (đoạn thẳng, hình tròn, văn bản) và đối tượng hợp thành (bánh xe = hình tròn + 6 đoạn thẳng).

Yêu cầu

- Thao tác với các đối tượng đơn giản/phức tạp một cách thống nhất - move, erase, rotate, set color
- Một vài đối tượng hợp thành được định nghĩa tĩnh (bánh xe) trong khi một vài đối tượng khác được định nghĩa động (do người dùng lựa chọn...)
- Đối tượng hợp thành có thể tạo ra bằng các đối tượng hợp thành khác
- Chúng ta cần một cấu trúc dữ liệu *thông minh*

Giải pháp

- Mọi đối tượng đơn giản kế thừa từ một giao diện chung, ví dụ *Graphic*:

```
class Graphic {
    abstract void move(int x, int y);
    abstract void setColor(Color c);
    abstract void rotate(double angle);
}
```

- Các lớp như *Line*, *Circle...* kế thừa *Graphic* và thêm các chi tiết (bán kính, độ dài,...)

Giải pháp (tt)

- Lớp dưới đây cũng là một lớp dẫn xuất:

```
class CompositeGraphic extends Graphic
{
  Graphics list[];
  ...
  public void rotate(double angle) {
    for (int i=0; i<list.length; i++)
      list[i].rotate();
  }
}
```

Giải pháp (tt)

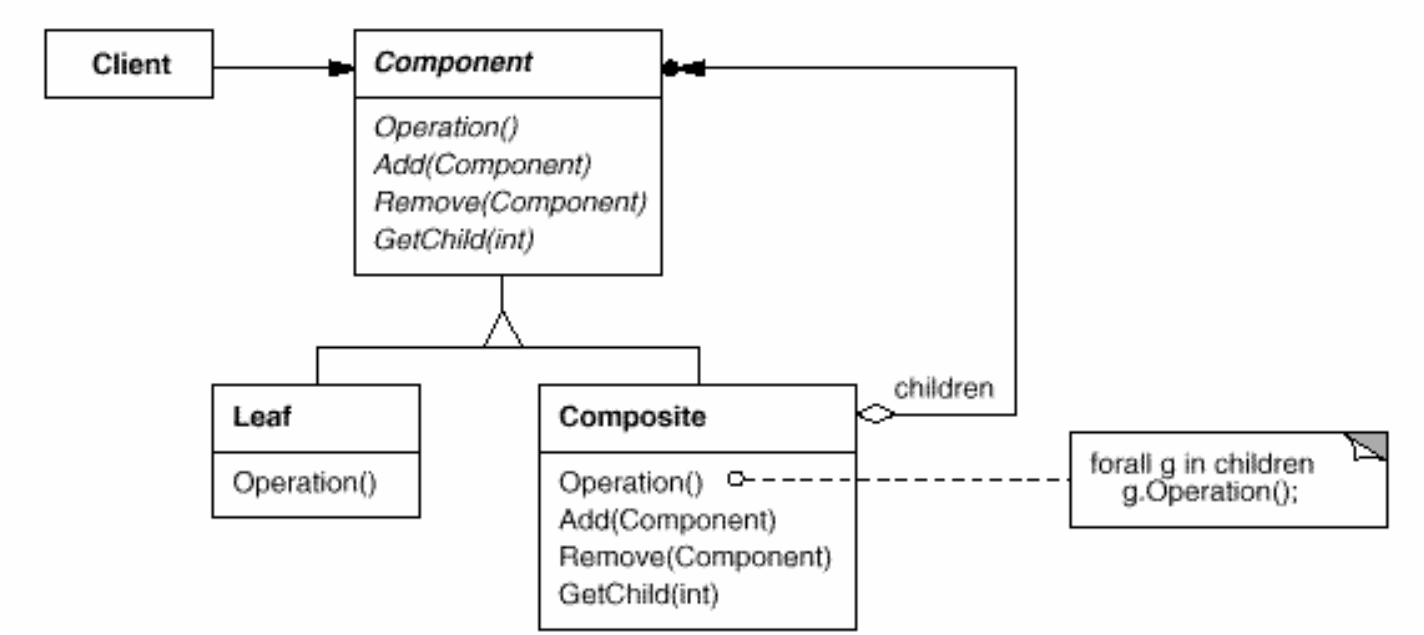
- *CompositeGraphic* là
  - một danh sách nên nó có *add(), remove()* và *count()*
  - *Graphic* nên nó còn có *rotate()*, *move()* và *setColor()*
- Các thao tác đó đối với một đối tượng hợp thành sử dụng một vòng lặp ‘for all’
- Thao tác thực hiện ngay cả với trường hợp thành phần của Composite lại là một Composite khác - cấu trúc dữ liệu dạng cây
- Có khả năng giữ thứ tự của các thành phần

Giải pháp (tt)

- Ví dụ tạo một đối tượng hợp thành:

```
CompositeGraphic cg;
cg = new CompositeGraphic();
cg.add(new Line(0,0,100,100));
cg.add(new Circle(50,50,100));
cg.rotate(90);
```

Sơ đồ lớp



- Kế thừa đơn
- Lớp cơ sở (root) chứa phương thức *add()*, *remove()*

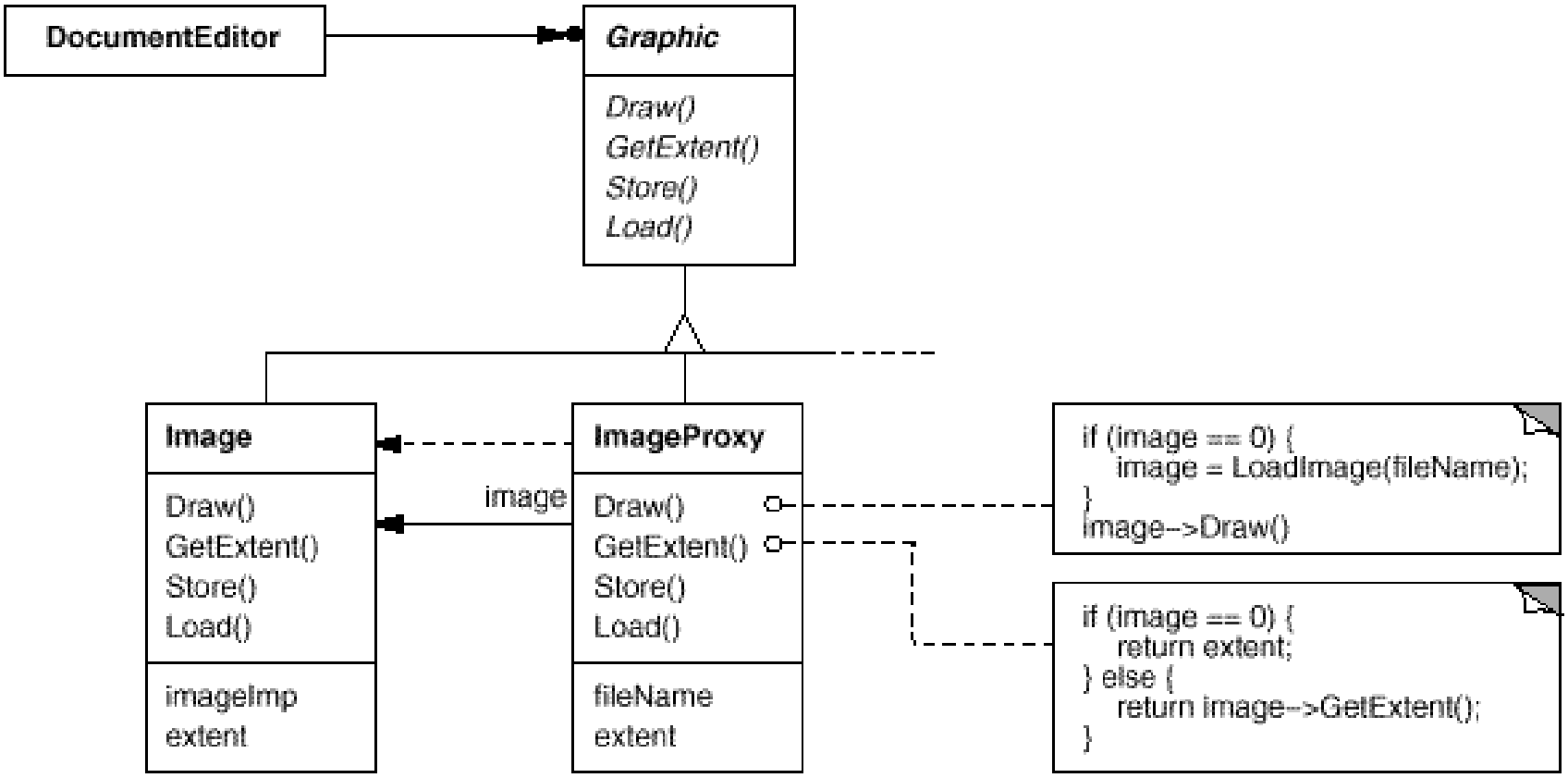
Ứng dụng

- Được dùng trong hầu hết các hệ thống HĐT
- Chương trình soạn thảo
- Giao diện đồ họa
- Cây phân tích cho biên dịch (một khối là một tập các lệnh/lời gọi hàm/các khối khác)

Proxy Pattern

- Các đối tượng có kích thước lớn, chỉ nên nạp vào bộ nhớ khi thực sự cần thiết; hay các đối tượng ở vùng địa chỉ khác (remote objects)
- Ví dụ: Xây dựng một trình soạn thảo văn bản có nhúng các đối tượng Graphic
  - Vấn đề đặt ra: Việc nạp các đối tượng Graphic phức tạp thường rất tốn kém, trong khi văn bản cần được mở nhanh
  - Giải pháp: sử dụng ImageProxy

Sơ đồ lớp


DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent;
} else {
return image->GetExtent();
}

Áp dụng

- Proxy được sử dụng khi nào cần thiết phải có một tham chiếu thông minh đến một đối tượng hơn là chỉ sử dụng một con trỏ đơn giản
  - cung cấp đại diện cho một đối tượng ở một không gian địa chỉ khác (remote proxy).
  - trì hoãn việc tạo ra các đối tượng phức tạp (virtual proxy).
  - quản lý truy cập đến đối tượng có nhiều quyền truy cập khác nhau (protection proxy).
  - smart reference

Strategy

- Chương trình cần chuyển đổi *động* giữa các thuật toán
- Ví dụ, chương trình soạn thảo sử dụng vài thuật toán hiển thị với các hiệu ứng/lợi ích khác nhau

Yêu cầu

- Thuật toán phức tạp và sẽ không có lợi khi cài đặt chúng trực tiếp trong lớp sử dụng chúng
  - ví dụ: việc cài thuật toán hiển thị vào lớp D*ocument* là không thích hợp
- Cần thay đổi động giữa các thuật toán
- Dễ dàng thêm thuật toán mới

Giải pháp

- Định nghĩa lớp trừu tượng để biểu diễn thuật toán:

```
class Renderer {
    abstract void render(Document d);
}
```

- Mỗi thuật toán là một lớp dẫn xuất

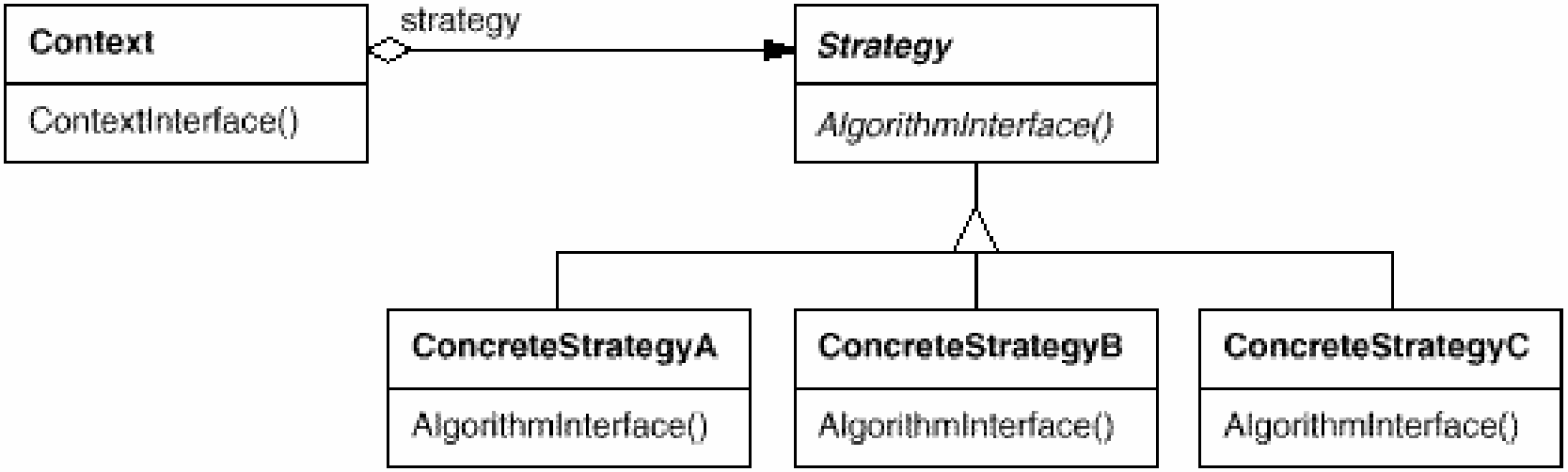
  FastRenderer, TexRenderer, …

Giải pháp (tt)

- Đối tượng "document" tự chọn thuật toán vẽ:

```
class Document {
  render() {
    renderer.render(this);
  }
  setFastRendering() {
    renderer = new FastRenderer();
  }
  private Renderer renderer;
}
```

Sơ đồ lớp


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Ứng dụng

- Chương trình vẽ/soạn thảo
- Tối ưu biên dịch
- Chọn lựa các thuật toán heuristic khác nhau (trò chơi...)
- Lựa chọn các phương thức quản lý bộ nhớ khác nhau

Một số lớp cơ sở

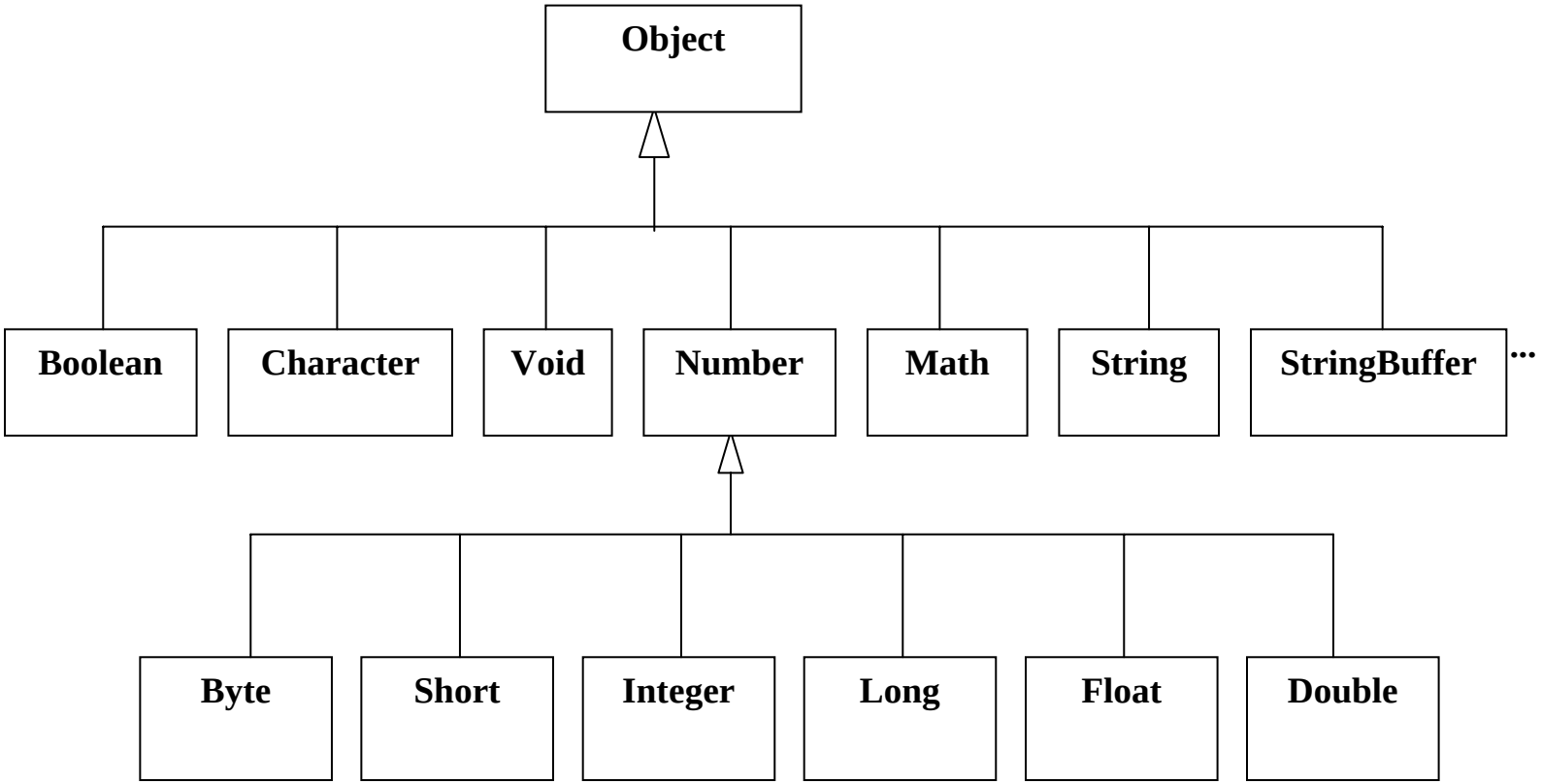
Nội dung

- Các lớp dữ liệu nguyên thủy
- Xâu ký tự
- Lớp Math
- Mảng
- Các lớp Container

Tài liệu tham khảo

- Bruce Eckel, *Thinking in Java, chapter 11*
- Deitel, *Java – How to program, chapter 7, 11, 20*
- Đoàn Văn Ban, *Lập trình hướng đối tượng với Java*, NXB Khoa học kỹ thuật, chương 6.

Một số lớp cơ bản



Lớp Object

- `Class getClass()`: trả lại tên lớp của đối tượng hiện thời.
- `boolean equals(Object)`: so sánh đối tượng, thường được định nghĩa lại.
- `String toString()`: trả lại biểu diễn văn bản của đối tượng, thường được định nghĩa lại.

```
Person p = new Person(”John”);
Class c = p.getClass();
System.out.println(c);
----
class Person
```

Các lớp dữ liệu nguyên thủy

- Các phương thức tiện ích
  - `valueOf(String s)`: trả đối tượng thuộc kiểu tương ứng
  - `typeValue()`: trả giá trị nguyên thủy tương ứng
  - `static parseType(String s)`: trả giá trị nguyên thủy tương ứng
- Hằng số
  - `Type.MAX_VALUE, Type.MIN_VALUE`

Lớp Character

- Các phương thức
  - `static boolean isUppercase(char ch)`
  - `static boolean isLowercase(char ch)`
  - `static boolean isDigit(char ch)`
  - `static boolean isLetter(char ch)`
  - `static boolean isLetterOrDigit(char ch)`
  - `static char toUpperCase(char ch)`
  - `static char toLowerCase(char ch)`

Lớp String

- Xâu ký tự không thay đổi được nội dung
- Khởi tạo
  - **`String(String), String(StringBuffer)`**
  - **`String(byte[]), String(char[])`**
- Phương thức
  - **`int length()`**: kích thước của xâu
  - **`char charAt(int index)`**: ký tự ở vị trí index

Lớp String

- So sánh
  - **`boolean equals(String)`**
  - **`boolean equalsIgnoreCase(String)`**
  - **`boolean startWith(String)`**
  - **`boolean endWith(String)`**
  - **`int compareTo(String)`**

Lớp String

- Chuyển đổi
  - **`String toUpperCase()`**
  - **`String toLowerCase()`**
- Ghép xâu
  - **`String concat(String)`**
  - toán tử “+”

Lớp String

- Tìm kiếm
  - **`int indexOf(char), int indexOf(char ch, int from)`**
  - **`int indexOf(String), int indexOf(String s, int from)`**
  - **`int lastIndexOf(char), lastIndexOf(char, int)`**
  - **`lastIndexOf(String), lastIndexOf(String, int)`**

Lớp String

- Thay thế
  - **`String replace(char ch, char new_ch)`**
- Trích xâu
  - **`String trim()`**: loại bỏ ký tự trắng
  - **`String substring(int startIndex)`**
  - **`String substring(int startIdx, int endIdx)`**

Lớp StringBuffer

- Xâu ký tự thay đổi được nội dung
- Khởi tạo
  - **`StringBuffer(String)`**
  - **`StringBuffer(int length)`**
  - **`StringBuffer()`**: đặt kích thước mặc định 16
- Các phương thức
  - **`int length(), void setLength()`**
  - **`char charAt(int index)`**
  - **`void setCharAt(int index, char ch)`**
  - **`String toString()`**

Lớp StringBuffer

- Thêm, xóa
  - **`append(String), append(type)`**
  - **`insert(int offset, String s), insert(int offset, char[] chs), insert(int offset, type t)`**
  - **`delete(int start, int end)`**: xóa xâu con
  - **`delete(int index)`**: xóa một ký tự
  - **`reverse()`**: đảo ngược

Lớp Math

- Hằng số
  - **`Math.E`**
  - **`Math.PI`**
- Các phương thức static
  - **`type abs(type)`**
  - **`double ceil(double), double floor(double)`**
  - **`int round(float), long round(double)`**
  - **`type max(type, type), type min(type, type)`**
  - **`double random()`**: sinh số ngẫu nhiên trong đoạn **`[0.0,1.0]`**

Lớp Math

- Lũy thừa
  - **`double pow(double, double)`**
  - **`double exp(double)`**
  - **`double log(double)`**
  - **`double sqrt(double)`**
- Lượng giác
  - **`double sin(double)`**
  - **`double cos(double)`**
  - **`double tan(double)`**

Mảng

- Mảng là đối tượng
  - chứa một tập các đối tượng khác
  - cần tạo ra trước khi sử dụng (new)

Ví dụ:

```
int a[];
a = new int[10];
for (int i=0; i<a.length; i++) a[i] = i*i;

int b[] = {2, 3, 5, 7};
a = b;
int m, n[];
double[] arr1, arr2;
```

Truyền tham số và nhận giá trị trả lại

```
int[] myCopy(int[] a) {
    int b[] = new int[a.length];
    for (i=0; i<a.length; i++)
        b[i] = a[i];
    return b;
}
...
int a[] = {0, 1, 1, 2, 3, 5, 8};
int b[] = myCopy(a);
```

Mảng nhiều chiều

```
int a[][];
a = new int[10][20];
a[2][3] = 10;
for (int i=0; i<a[0].length; i++)
    a[0][i] = i;

int b[][] = { {1 , 2}, {3, 4} };
int c[][] = new int[2][];
c[0] = new int[5];
c[1] = new int[10];
```

Copy mảng

- `System.arraycopy(src, s_off, des, d_off, len)`
  - `src`: mảng nguồn, `s_off`: offset của mảng nguồn
  - `des`: mảng đích, `d_off`: offset của mảng đích
  - `len`: số phần tử cần copy
- Copy nội dung của dữ liệu nguyên thủy, copy tham chiếu đối với đối tượng

Lớp Arrays

- Nằm trong gói `java.util`
- Cung cấp 4 phương thức static để làm việc với mảng
  - `fill()`: khởi tạo các phần tử của mảng với một giá trị như nhau
  - `sort()`: xắp xếp mảng
  - `equals()`: so sánh hai mảng
  - `binarySearch()`: tìm kiếm nhị phân trên mảng đã sắp xếp

So sánh mảng equals()

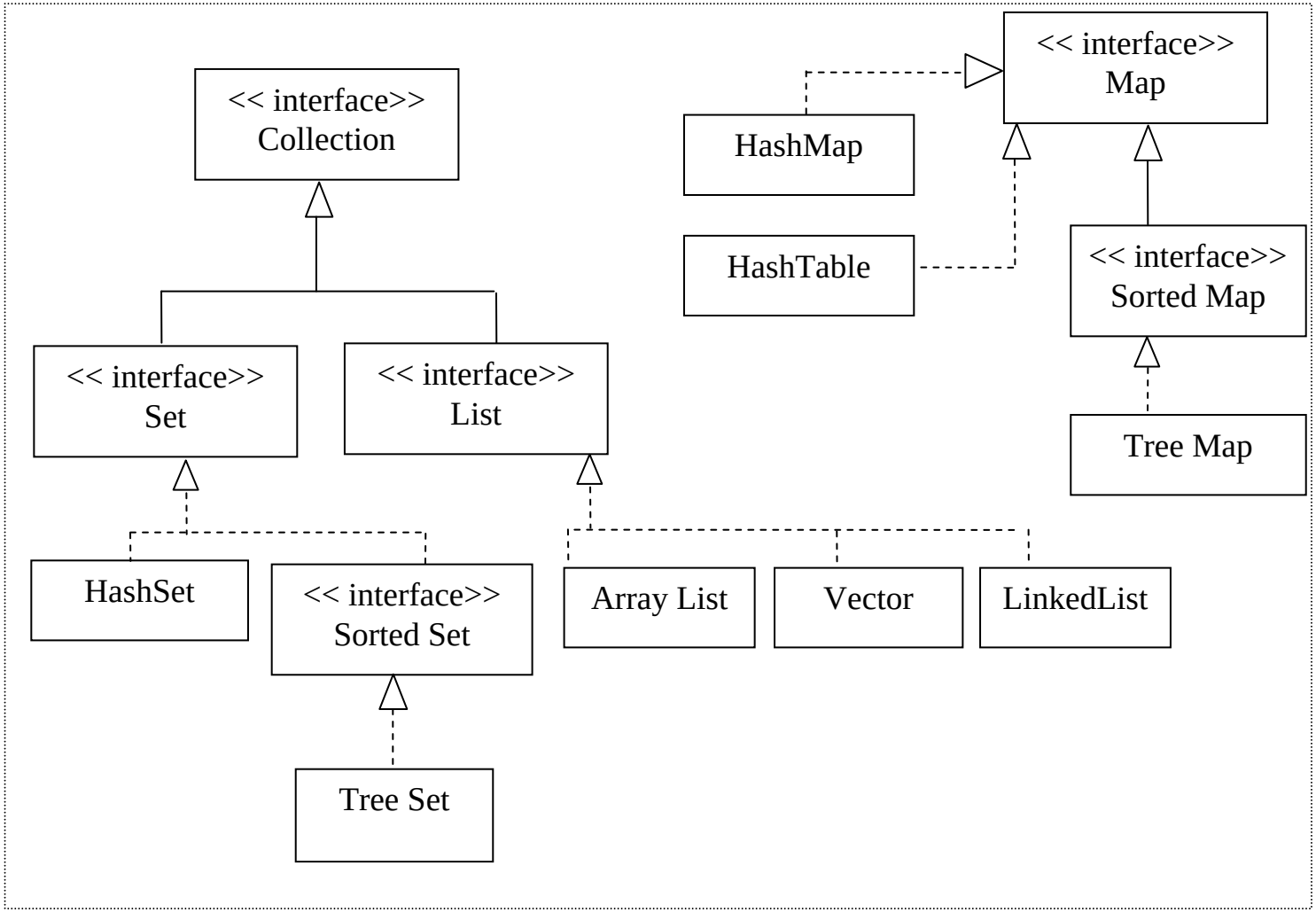
- So sánh mảng dữ liệu nguyên thủy
- Gọi phương thức `equals()` để so sánh mảng đối tượng

```
int a[] = { 1, 3 , 2 , 4 };
int b[] = new int[a.length];
System.arraycopy(a,0,b,0,a.length);
System.out.println(Arrays.equals(a,b));
```

Sắp xếp mảng `sort()`

- Làm việc với các mảng dữ liệu nguyên thủy
- Làm việc với các lớp đối tượng có cài đặt giao diện `Comparable`
  - phương thức `compareTo()`

Các lớp tuyển tập (Container)



Iterator

- Mẫu dùng để duyệt các phần tử của một tập hợp
- Là một interface trong Java:
  - hasNext()
  - next()
  - remove()
- Các lớp Collection cài đặt Iterator

```
import java.util.*;

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

        Collection list = new LinkedList();

        list.add(3);
        list.add(2);
        list.add(1);
        list.add(0);
        list.add("happy new year!");

        Iterator i = list.iterator();
        while (i.hasNext()) {
               System.out.println(i.next());
        }
   }
}
```

```
import java.util.*;

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

        List list = new LinkedList();

        list.add(3);
        list.add(2);
        list.add(1);
        list.add(0);
        list.add("go!");

        for (int i=0; i<list.size(); i++) {
              System.out.println(list.get(i));
        }
   }
}
```