

BÀI GIẢNG MÔN HỌC

LẬP TRÌNH JAVA



Nội Dung Học Phần

- Mở đầu : Giới thiệu tổng quan
- Chương 1 : Sơ lược lập trình hướng đối tượng
- Chương 2 : Giới thiệu về Java
- Chương 3 : Các thành phần cơ bản -
Cấu trúc chương trình Java
- Chương 4 : Interface & Package
- Chương 5 : Exception (Xử lý biệt lệ)
- Chương 6 : Lập trình giao diện
- Chương 7 : Lập trình đa tuyến
- Chương 8 : Lập trình các luồng vào ra
- Chương 9 : Lập trình cơ sở dữ liệu

Chương 1

GIỚI THIỆU SƠ LƯỢC LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Mục Tiêu Bài Học

- Thế nào là lập trình hướng đối tượng
- Tìm hiểu về trừu tượng dữ liệu
- Định nghĩa lớp và đối tượng
- Constructor và Destructor
- Tìm hiểu về tính lưu trữ, bao bọc dữ liệu, tính kế thừa và đa hình
- Các ưu điểm của phương pháp lập trình hướng đối tượng

Lập Trình Hướng Đối Tượng

- Lấy đối tượng làm nền tảng cơ sở của phương pháp lập trình
- Phương pháp thiết kế và thực hiện bằng các hệ phần mềm

Trừu Tượng Dữ Liệu

- Là tiến trình xác định và tập hợp các tính chất và các hành động của một thực thể có liên quan đến ứng dụng
- Lợi ích :
 - Tập trung vào vấn đề
 - Xác định những tính chất và hành động thiết yếu
 - Loại trừ những chi tiết không cần thiết

Trừu Tượng Dữ Liệu

**Các tính chất
của một đối
tượng Người**

Tên

Địa chỉ

Tuổi

Chiều cao

Màu tóc

**Các tính chất của
một đối tượng
Khách hàng**

Mã khách hàng

Tên khách hàng

Địa chỉ

Trình Tượng Dữ Liệu (tiếp theo)

Các thuộc tính	Các hành động
Mã khách hàng	Nhập mã khách hàng
Tên của khách hàng	Nhập tên của khách hàng
Địa chỉ của khách hàng	Nhập địa chỉ của khách hàng
Sản phẩm đã mua	Nhập sản phẩm mua được
	Lập hóa đơn

Lớp

- Lớp là một nhóm các đối tượng có chung những tính chất và hành động

Lớp Khách hàng
Mã khách hàng
Tên khách hàng
Địa chỉ khách hàng
Sản phẩm đã mua
Nhập mã khách hàng
Nhập tên
Nhập địa chỉ
Nhập sản phẩm mua được
Lập hóa đơn

Đối Tượng

- Đối tượng là một thể hiện của lớp
 - Nhi
 - Đức
 - Phúc

Đối Tượng (tiếp theo)

- Thuộc tính
 - Tính chất mô tả một đối tượng
- Hành động
 - Dịch vụ mà đối tượng có thể đáp ứng
- Phương thức
 - Đặc tả cách đáp ứng bằng hành động khi được yêu cầu
- Thông điệp
 - Yêu cầu một hành động
- Biến cố
 - Sự kích thích từ đối tượng này gửi sang đối tượng khác

Lớp Và Đối Tượng

- Lớp là một thực thể, còn đối tượng là một thực thể thực tế
- Lớp là một mô hình ý niệm định rõ các tính chất và các hành động được quy định bởi một đối tượng, còn đối tượng là một mô hình thực sự
- Lớp là khuôn mẫu từ đó đối tượng được tạo ra
- Tất cả các đối tượng trong cùng một lớp có các tính chất và các hành động như nhau

Constructor

- Tiến trình tạo ra một đối tượng được gọi là Constructor
- Một Constructor:
 - Cấp phát vùng nhớ
 - Khởi gán những thuộc tính (nếu có)
 - Cho phép truy cập những thuộc tính và phương thức

Destructor

- Tiến trình hủy một đối tượng gọi là Destructor
- Một Destructor:
 - Giải phóng bộ nhớ
 - Cấm truy cập thuộc tính và phương thức

Tính Lưu Trữ

- Tính lưu trữ là khả năng của đối tượng có thể lưu lại dữ liệu của nó sau khi đã bị hủy

Tính Bao Bọc Dữ Liệu

- Tiến trình **che dấu những chi tiết hiện thực** một đối tượng được gọi là tính bao bọc
- Ưu điểm:
 - Tất cả những thuộc tính và phương thức cần thiết đều được tạo
 - Một lớp có thể có nhiều tính chất và phương thức nhưng chỉ một số trong đó được hiển thị cho người dùng

Tính Kế Thừa

LỚP SINH VIÊN	LỚP NHÂN VIÊN	LỚP KHÁCH HÀNG
Tên	Tên	Tên
Địa chỉ	Địa chỉ	Địa chỉ
Điểm môn 1	Lương	Sản phẩm mua được
Điểm môn 2	Chức vụ	Nhập tên
Nhập tên	Nhập tên	Nhập địa chỉ
Nhập địa chỉ	Nhập địa chỉ	Nhập mã sản phẩm
Nhập điểm	Nhập lương	Lập hóa đơn
Tính tổng số điểm	Tính lương	

Tính Kế Thừa (tiếp theo)

Lớp Người
Tên
Địa chỉ
Nhập tên
Nhập địa chỉ

Tính Kế Thừa (tiếp theo)

LỚP NGƯỜI
Tên
Địa chỉ
Nhập tên
Nhập địa chỉ

+

=

Lớp Khách Hàng

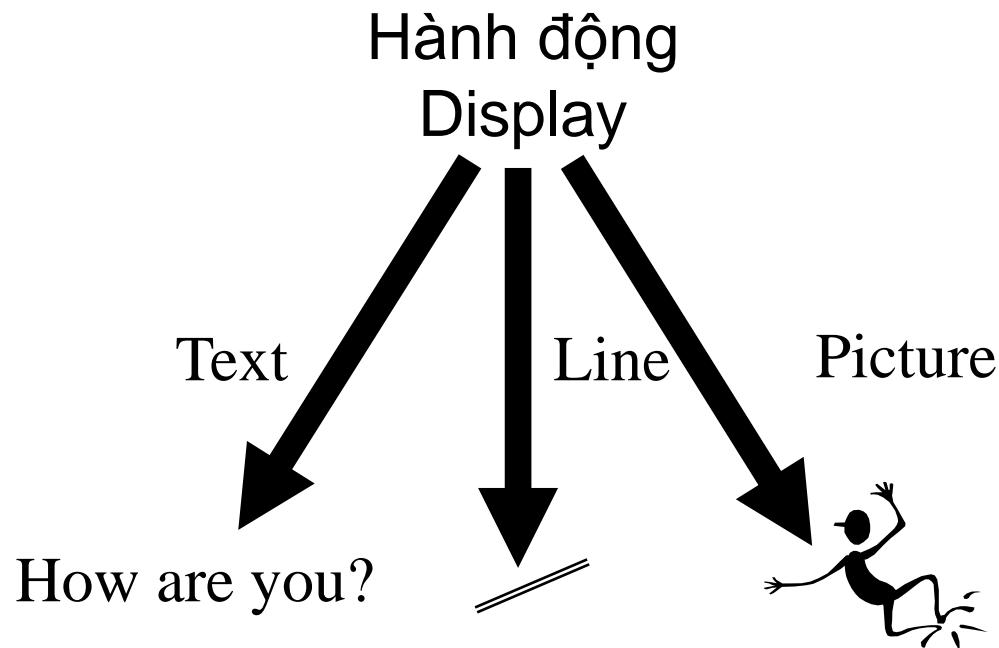
Thêm các thuộc tính và hành động cần thiết vào lớp khách hàng
Nhập mã sản phẩm đã mua
Lập hóa đơn

Tính Kế Thừa (tiếp theo)

- Tính Thừa kế
 - Là cơ chế cho phép một lớp chia sẻ những thuộc tính và những hành động đã định nghĩa trong một hoặc nhiều lớp khác
- Lớp con
 - Là lớp thừa kế từ lớp khác
- Lớp cha
 - Là lớp từ đó một lớp khác thừa kế các ứng xử của nó
- Đa thừa kế
 - Khi một lớp con thừa kế từ hai hoặc nhiều lớp

Tính Đa Hình

- Tính đa hình là thuộc tính cho phép một hành động ứng xử khác nhau trên các lớp khác nhau
- Đa hình trong biên dịch (phương thức nạp chồng-overloading)
- Đa hình trong thông dịch (phương thức ghi đè-overriding)



Ưu điểm của phương pháp hướng đối tượng

- Chia sẻ trong phạm vi một ứng dụng
- Đẩy mạnh sự dùng lại của các đối tượng khi hiện thực những ứng dụng mới
- Về lâu dài, chi phí giảm đáng kể
- Giảm lỗi và rắc rối trong bảo trì
- Điều chỉnh nhanh hơn

TỔNG KẾT

- Tiếp cận hướng đối tượng đưa ra một giải pháp toàn diện cho một bài toán cụ thể
- Trừu tượng dữ liệu là một tiến trình xác định và tập hợp các tính chất và các hành động có quan hệ với một thực thể cụ thể
- Lớp mô tả một thực thể, còn đối tượng là một thực thể thực tế
- Constructor và Destructor
- Tính lưu trữ, bao bọc dữ liệu, tính kế thừa và đa hình

Chương 2

Giới thiệu về ngôn ngữ Java

Nội dung chính

- Nắm được các đặc trưng của Java
- Các kiểu chương trình Java
- Định nghĩa về máy ảo Java
- Các nội dung của JDK(Java Development Kit)
- Các gói chuẩn của Java

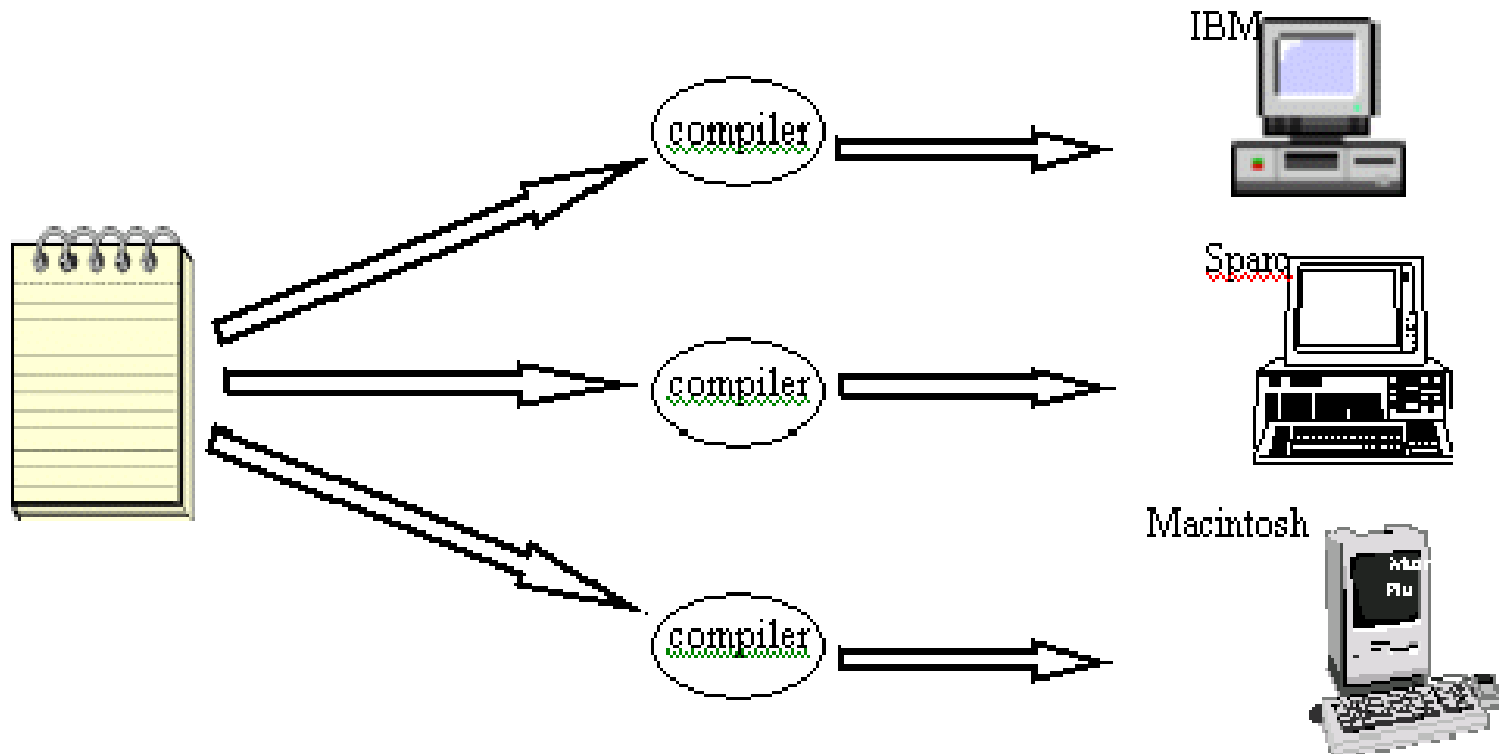
Giới thiệu

- Sự phát triển của Java
- Hướng tới người dùng
- Giống với C / C++

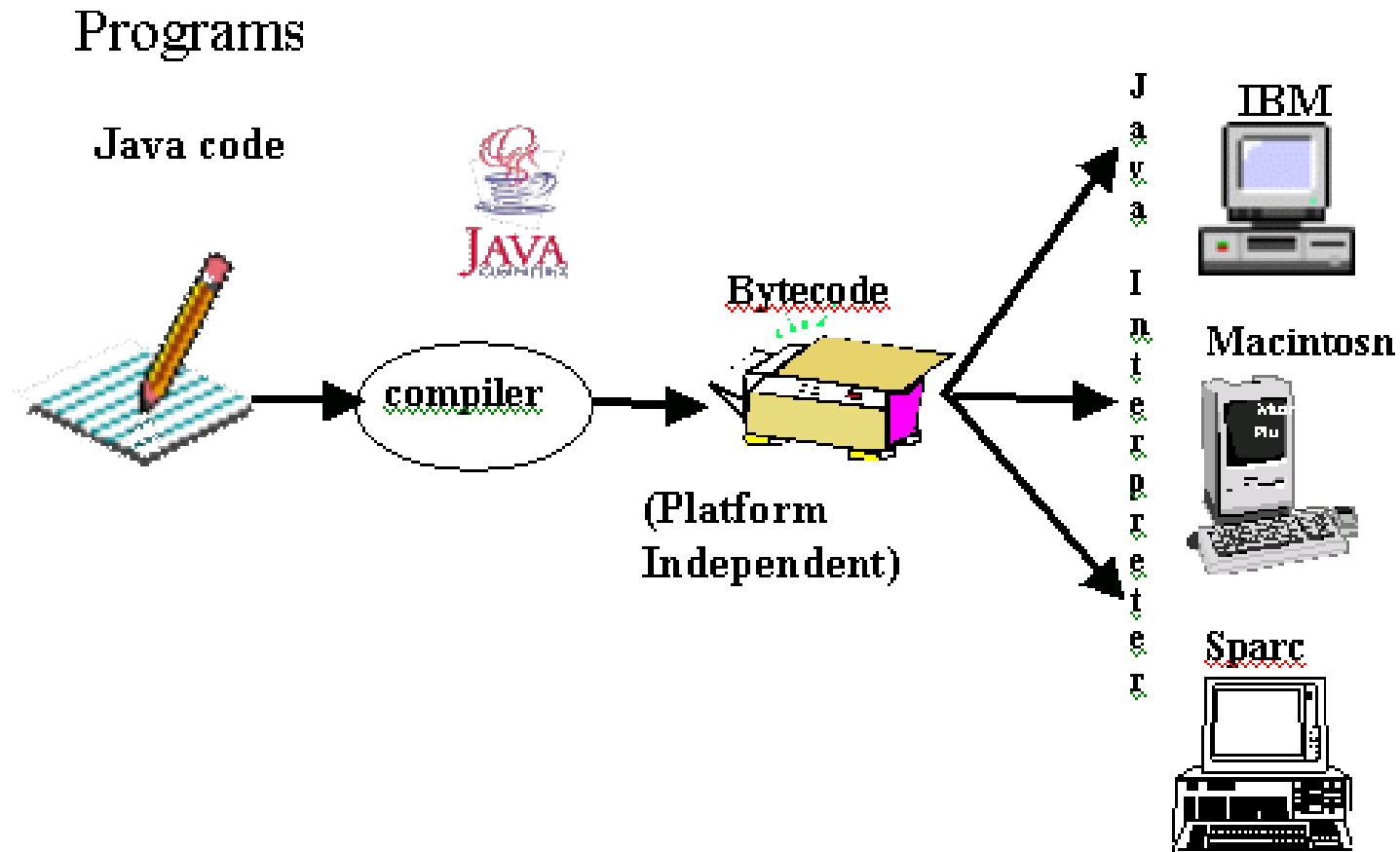
Các đặc trưng của Java

- Đơn giản
- Hướng đối tượng
- Độc lập phần cứng
- Mạnh
- Bảo mật
- Phân tán
- Đa luồng
- Động

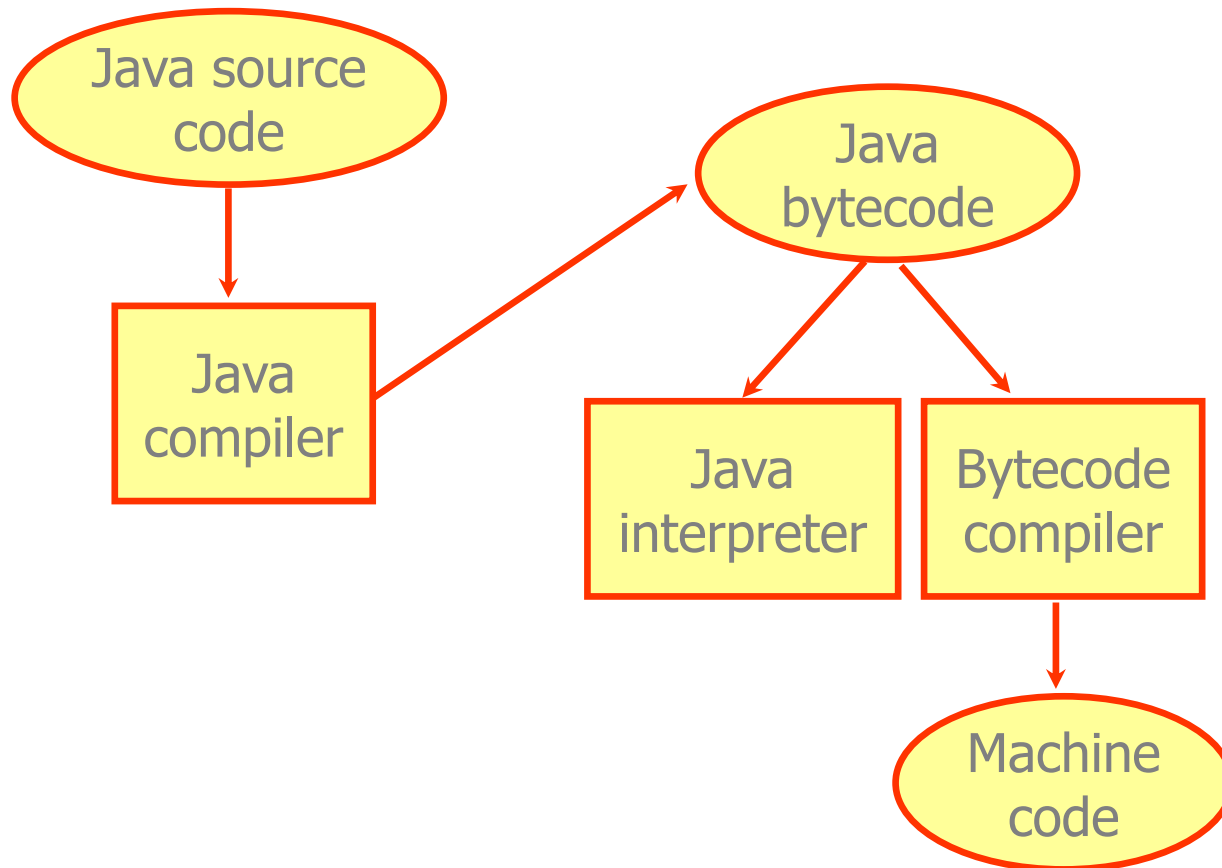
Các chương trình dịch truyền thống



Chương trình dịch Java



Quá trình biên dịch và thực thi chương trình Java



Các loại chương trình Java

- Ứng dụng độc lập (console Application)
- Ứng dụng giao diện (GUI Application)
- Applets
- Servlet
- Ứng dụng cơ sở dữ liệu

Máy ảo Java

- Là một phần mềm dựa trên cơ sở máy tính ảo
- Là tập hợp các lệnh logic để xác định hoạt động của máy tính
- Được xem như là một hệ điều hành thu nhỏ
- Nó thiết lập lớp trừu tượng cho:
 - Phần cứng bên dưới
 - Hệ điều hành
 - Mã đã biên dịch

Các bước để dịch một chương trình Java

- Trình **biên dịch** chuyển mã nguồn thành tập các lệnh không phụ thuộc vào phần cứng cụ thể
- Trình **thông dịch** trên mỗi máy chuyển tập lệnh này thành chương trình thực thi
- Máy ảo tạo ra một môi trường để thực thi các lệnh bằng cách:
 - Nạp các file .class
 - Quản lý bộ nhớ
 - Dọn “rác”

Quản lý bộ nhớ và dọn rác

- Heap là vùng bộ nhớ chia sẻ giữa các Thread
- Bộ nhớ được theo dõi qua các danh sách sau:
 - Danh sách các vùng nhớ chưa cấp phát
 - Danh sách các vùng đã cấp phát
- Gom lại các vùng nhớ chưa dùng liền nhau
- Sắp xếp lại các phần đã dùng để tạo vùng rảnh lớn hơn
- Cấu trúc Handle
- Hàm Finalize

Quá trình kiểm tra file .class

- Tất cả các file .class nạp vào bộ nhớ đều được kiểm tra để đảm bảo an toàn
- Ba phần logic của file .class là:
 - Bytecode
 - Thông tin về class
 - Các thuộc tính của class
- Các thông tin của file .class được xem xét riêng rẽ trong các bảng sau:
 - Bảng Field chứa các thuộc tính
 - Bảng Method chứa các hàm của class
 - Bảng Interface chứa các giao diện và các hằng số

Quá trình kiểm tra file .class (tiếp...)

- Quá trình kiểm tra file .class được thực hiện ở bốn mức:
 - Kiểm tra cú pháp
 - Sự nhất quán về ngữ nghĩa
 - Kiểm tra Bytecode
 - Kiểm tra trong thời gian thực thi

Trình dịch Java

Java Development Kit

- Java 1.0
- Java 1.1
- Java 2
-

Bộ công cụ JDK

- Trình biên dịch, 'javac'
 - **javac [options] sourcecodename.java**
- Trình thông dịch, 'java'
 - **java [options] classname**
- Trình dịch ngược, 'javap'
 - **javap [options] classname**
- Công cụ sinh tài liệu, 'javadoc'
 - **javadoc [options] sourcecodename.java**

- Chương trình tìm lỗi - Debug, 'jdb'
 - **jdb [options] sourcecodename.java**
 - OR
 - **jdb -host -password [options]
sourcecodename.java**
- Chương trình xem Applet ,
'appletviewer'
 - **appletviewer [options]
sourcecodename.java / url**

Các gói chuẩn của Java

- `java.lang`
- `java.applet`
- `java.awt`
- `java.io`
- `java.util`
- `java.net`
- `java.awt.event`
- `java.rmi`
- `java.security`
- `java.sql`

Chương 3

Các kiến thức cơ bản – Cấu trúc chương trình của ngôn ngữ Java

Các thành phần cơ bản

1. TẬP KÝ TỰ CỦA JAVA
2. TỪ KHOÁ
3. TÊN, LỜI CHÚ THÍCH
4. CÂU LỆNH VÀ DẤU CHẤM CÂU
5. CẤU TRÚC CỦA CHƯƠNG TRÌNH JAVA
6. MỘT SỐ CHƯƠNG TRÌNH JAVA MẪU

Tập ký tự của java

Mọi ngôn ngữ đều được xây dựng trên **bộ kí tự**. Bộ kí tự của Java như sau:

- + Các chữ cái in hoa: A, B, C, ... Y, Z
- + Các chữ cái in thường: a, b, c, ... y, z
- + Các chữ số: 0, 1, 2, ..., 9
- + Các dấu câu: , ; : / ? . [] { } ! ' " ~ @ # \$ % ^ & * ()
- + = < >
- + Các dấu ngăn cách không nhìn thấy: dấu cách, dấu tab, dấu xuống hàng enter
- + Dấu gạch nối dưới _

Chú ý:

- + Java phân biệt chữ in hoa và chữ in thường.
- + Có thể dùng các kí tự khác như Φ , θ , ∞ , σ , ϵ , ... hay cả tiếng Việt trong chương trình Java

Bộ từ khóa của Java

abstract	boolean	break	byte
case	catch	char	class
const	continue	default	do
double	else	extends	final
finally	float	for	goto
if	implements	import	instanceof
int	interface	long	native
new	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	try
void	volatile	while	

Tên và Lời chú thích

* *Tên* :

- Tên dùng để xác định các đại lượng khác nhau trong một chương trình.
- Mọi tên được sử dụng trong chương trình đều phải được khai báo.
- Tên hợp lệ là dãy kí tự liền nhau, có thể gồm các chữ cái (a..,z, A..,Z), các chữ số (0..9), kí tự gạch dưới (_).
- Tên phải bắt đầu bằng chữ cái hoặc dấu gạch dưới.
- Tên không được trùng với từ khoá.

Ví dụ : Nghiem_x, hoan_doi, y_1, y2,.. là tên hợp lệ

Tên và Lời chú thích

* *Lời chú thích :*

- Không có tác dụng tạo ra mã của chương trình, giúp dễ dàng trong việc tìm kiếm, kiểm tra, sửa chương trình
- Lời chú thích có thể được đặt ở bất kỳ đâu trong chương trình
- Cách ghi lời chú thích:

Cách 1 : Chú thích cho nhiều dòng

/* lời chú thích

*/

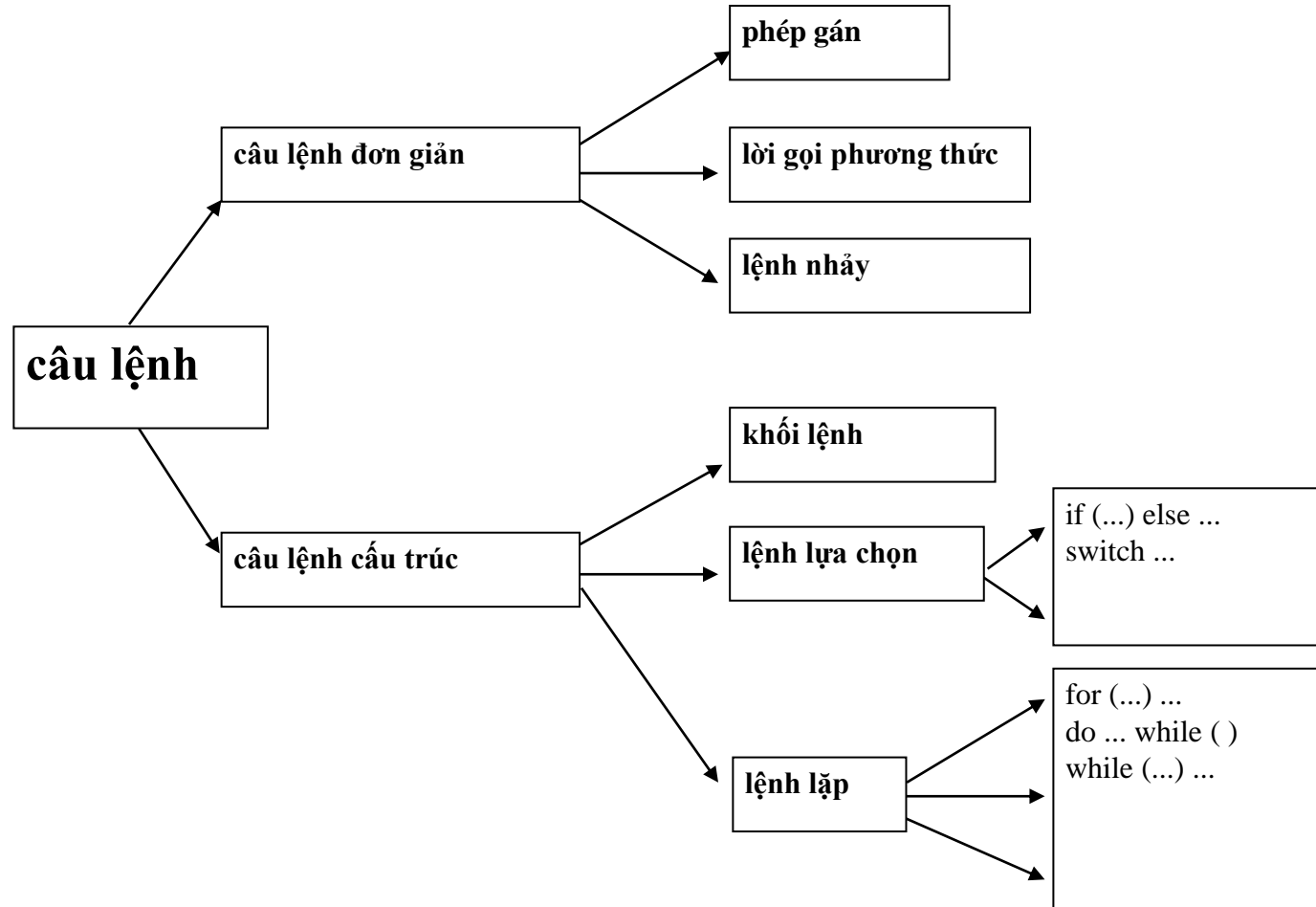
Cách 2 : Chú thích cho một dòng

// lời chú thích

Câu lệnh và Dấu chấm câu

- Một câu lệnh thường gồm phần mô tả dữ liệu và phần lệnh theo quy định của Java.
- Có hai loại:
 - + Câu lệnh đơn giản: là những lệnh không chứa các lệnh khác : lệnh gán, lệnh gọi phương thức, ...
 - + Câu lệnh cấu trúc: gồm khối lệnh, lệnh rẽ nhánh, lệnh lặp
- Khối lệnh là gồm nhiều lệnh được đặt trong dấu { }
- Các câu lệnh cách nhau bởi dấu “;”
- Dấu chấm phẩy “;” dùng để ngăn cách các lệnh, nghĩa là cuối mỗi lệnh, mỗi khai báo đều có chấm phẩy

Câu lệnh và Dấu chấm câu



Cấu trúc một chương trình Java

- Xác lập thông tin môi trường
- Khai báo lớp đối tượng (Class)
- Các thành phần (Tokens):
 - Định danh
 - Từ khóa / từ dự phòng
 - Ký tự phân cách
 - Hằng (Literals)
 - Toán tử

Java Program Structure

// comments about the class

```
public class MyProgram  
{
```

// comments about the method

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

```
{
```

```
}
```

```
}
```

method body

method header

Chương trình Java mẫu

```
// This is a simple program called "Ex1.java"  
// import java.lang.*;  
class Ex1  
{  
    public static void main(String args[])  
    {  
        System.out.println("My first program in Java");  
    }  
}
```

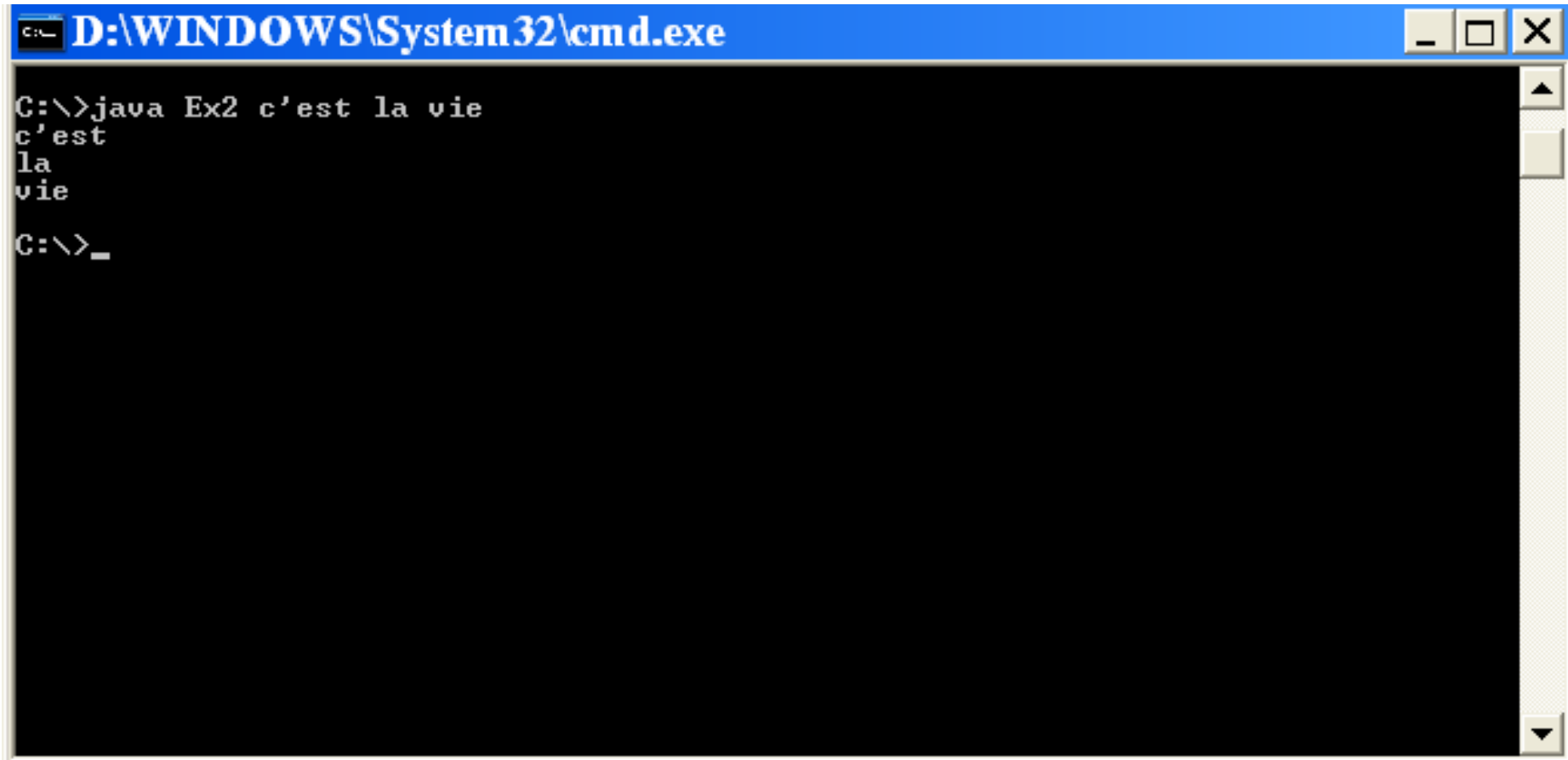
Biên dịch chương trình java

- **..\jdk\bin>javac Ex1.java**
- **..\jdk\bin>java Ex1**
- Kết quả:
My first program in Java

Truyền đối số dòng lệnh

```
class Ex2
{
    public static void main(String s[])
    {
        System.out.println(s[0]);
        System.out.println(s[1]);
        System.out.println(s[2]);
    }
}
```

Truyền đối số trong dòng lệnh (Tiếp theo...)



```
D:\WINDOWS\System32\cmd.exe

C:\>java Ex2 c'est la vie
c'est
la
vie
C:\>_
```

The image shows a Windows command prompt window with a blue title bar. The title bar text is "D:\WINDOWS\System32\cmd.exe". The command prompt shows the command "C:\>java Ex2 c'est la vie" being executed. The output of the command is displayed on the following lines: "c'est", "la", and "vie". The prompt "C:\>_" is shown on the line following the output.

Chương trình tính căn bậc hai của m

```
import java.io.*;
public class canbachai
{
    public static void Tinhcb2(int m)
    {
        double n=0.0;
        n=Math.sqrt(m);
        System.out.println("Can bac hai cua " +m+ "la" +n);
    }
    public static void main(String args[])
    {
        // tạo đối tượng để gọi
        canbachai t=new canbachai();
        t.Tinhcb2(9);
        t.Tinhcb2(16);
    }
}
```

Chương trình tính n^{10}

```
import java.io.*;
class Tinh_n_mu10{
    public static long Tinh(int n)
    {
        long tg=1;
        for(int i=1;i<=10;i++) tg*=n;
        return tg;
    }
    public static void main(String [] arg){
        Tinh_n_mu10 t=new Tinh_n_mu10 ();

        System.out.println("Luy thua 10 cua so la :"+t.Tinh(2));
    }
}
```


Chương trình kiểm tra một số có phải bội của 5 không

```
import java.io.*;
public class Boi5
{
    public static void Kiemtraboi5(int m)
    {
        if(m%5!=0)
            System.out.println(m+"Khong la boi so cua 5");
        else
            System.out.println(m+"La boi cua 5");
    }
    public static void main(String[] args)
    {
        Boi5 dt1=new Boi5();// tạo đối tượng dt1
        dt1.Kiemtraboi5(9);
        Boi5 dt2=new Boi5();// tạo đối tượng dt2
        dt2.Kiemtraboi5(10);
    }
}
```

Chương trình nhận các ký tự gõ vào từ bàn phím và chỉ nhận các ký tự số

```
class Convert
{
    public static void main (String [] args) throws java.io.IOException {
        System.out.print ("Please enter a number: ");
        int num = 0;
        int ch;
        while ((ch = System.in.read ()) != '\n')
            if (ch >= '0' && ch <= '9') {
                num *= 10;
                num += ch - '0';
            }
            else
                break;
        System.out.println ("num = " + num + "num squared = " + num * num);
    }
}
//class
```

Kiểu dữ liệu

- Kiểu dữ liệu cơ sở (Primitive Data Types)
- Kiểu dữ liệu tham chiếu (Reference data types)

Kiểu dữ liệu cơ sở

- 4 kiểu integers:
 - byte, short, int, long
- 2 kiểu số thực:
 - float, double
- 1 kiểu characters:
 - char
- 1 kiểu boolean :
 - boolean

Kiểu dữ liệu cơ sở

<u>Type</u>	<u>Storage</u>	<u>Min Value</u>	<u>Max Value</u>
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
float	32 bits	+/- 3.4×10^{38} with 7 significant digits	
double	64 bits	+/- 1.7×10^{308} with 15 significant digits	

Kiểu dữ liệu tham chiếu

- Mảng (Array)
- Lớp (Class)
- Interface

Các toán tử

- Các loại toán tử:
 - Toán tử số học (Arithmetic operators)
 - Toán tử dạng Bit (Bitwise operators)
 - Toán tử so sánh (Relational operators)
 - Toán tử logic (Logical operators)
 - Toán tử điều kiện (Conditional operator)
 - Toán tử gán (Assignment operator)

Toán tử số học

Arithmetic Operators

+	Addition (Phép cộng)
-	Subtraction (Phép trừ)
*	Multiplication (Phép nhân)
/	Division (Phép chia)
%	Modulus (Lấy số dư)
++	Increment (Tăng dần)
--	Decrement (Giảm dần)

<code>+=</code>	Phép cộng và gán
<code>-=</code>	Phép trừ và gán
<code>*=</code>	Phép nhân và gán
<code>/=</code>	Phép chia và gán
<code>%=</code>	Phép lấy số dư và gán

Ví dụ : `x+=2; // x=x+2`

....

Toán tử so sánh (Relational Operators)

==	So sánh bằng
!=	So sánh khác
<	Nhỏ hơn
>	Lớn hơn
<=	Nhỏ hơn hoặc bằng
>=	Lớn hơn hoặc bằng

Toán tử Bit (Bitwise Operators)

~	Phủ định (NOT)
&	Và (AND)
	Hoặc (OR)
^	Exclusive OR (XOR)
>>	Dịch sang phải (Shift right)
<<	Dịch sang trái (Shift left)

Toán tử Logic (Logical Operators)

&&

Logical AND

||

Logical OR

!

Logical unary NOT

Toán tử điều kiện (Conditional Operator)

- Cú pháp

Biểu thức 1 ? Biểu thức 2 : Biểu thức 3;

ĐK ? BT1 : BT2

- **Biểu thức 1**

Điều kiện kiểu Boolean trả về giá trị True hoặc False

- **Biểu thức 2**

Trả về giá trị nếu kết quả của mệnh đề 1 là True

- **Biểu thức 3**

Trả về giá trị nếu kết quả của mệnh đề 1 là False

Toán tử gán (Assignment Operator)

= Assignment (Phép gán)

Giá trị có thể được gán cho nhiều biến số

- Ví dụ

a = b = c = d = 90;

Thứ tự ưu tiên của các toán tử

Thứ tự	Toán tử
1.	trong ngoặc tính trước
2.	Các toán tử đơn như $+$, $-$, $++$, $--$
3.	Các toán tử số học và các toán tử dịch như $*$, $/$, $+$, $-$, $<<$, $>>$
4.	Các toán tử quan hệ như $>$, $<$, $>=$, $<=$, $=$, $!=$
5.	Các toán tử logic và Bit như $\&\&$, $\ \ $, $\&$, $\ $, \wedge
6.	Các toán tử gán như $=$, $*=$, $/=$, $+=$, $-=$

- Thứ tự của các toán tử có thể được thay đổi bằng cách sử dụng các dấu ngoặc đơn trong mệnh đề

Thứ tự ưu tiên của các toán tử

- What is the order of evaluation in the following expressions?

$$a + b + c + d + e$$

1 2 3 4

$$a - b / c + d * e$$

3 1 4 2

$$a / (b + c) - d \% e$$

2 1 4 3

$$a / (b * (c + (d - e)))$$

4 3 2 1

Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

```
answer = sum / 4 + MAX * lowest;
```

4

1

3

2



Then the result is stored in the variable on the left hand side

Assignment Revisited

- The right and left hand sides of an assignment statement can contain the same variable

First, one is added to the
original value of `count`

```
count = count + 1;
```



Then the result is stored back into `count`
(overwriting the original value)

Ép kiểu (Type Casting)

- Kiểu dữ liệu này được chuyển đổi sang một kiểu dữ liệu khác

Cú pháp: **(kiểu_mới)biểu_thức;**

Ví dụ :

```
int m=8, n=2;
```

```
float x1,x2;
```

```
x1 = m/n; // x1 nhận giá trị là số nguyên;
```

```
x2 = m/float(n); // x2 nhận giá trị là số thực;
```

Ví dụ :

```
float c = 34.29675;
```

```
int b = (int)c + 10;
```

Các kí tự định dạng xuất dữ liệu (Escape Sequences)

Escape Sequence	Mô tả
\n	Xuống dòng mới
\r	Chuyển con trỏ đến đầu dòng hiện hành
\t	Chuyển con trỏ đến vị trí dừng Tab kế tiếp (kí tự Tab)
\\	In dấu \
\'	In dấu nháy đơn (')
\''	In dấu nháy kép (")

Hằng, Biến, Biểu thức

* *Hằng* :

- Hằng là đại lượng có giá trị không thay đổi trong suốt quá trình thực thi của chương trình

* *Biến* :

- Biến là đại lượng có giá trị thay đổi trong suốt quá trình thực thi của chương trình
- Khai báo biến để chương trình dành riêng vùng nhớ thích hợp cho biến đó.
- Mọi lệnh truy cập đến biến là truy cập đến giá trị của nó.

Hằng, Biến, Biểu thức

* *Biến* :

- Cú pháp khai báo biến :
datatype identifier [=value][, identifier [=value]...];
 - Khai báo biến số gồm 3 thành phần:
 - Kiểu dữ liệu của biến số
 - Tên biến
 - Giá trị ban đầu của biến (không bắt buộc)
- VD: int x, y=5, z;

Hằng, Biến, Biểu thức

* ***Biểu thức :***

- Là một dãy các toán tử và toán hạng
 - + Toán hạng: có thể là hằng, hàm, biến, ...
 - + Toán tử : $+$, $-$, $*$, $/$, $\$$, $\%$, $\&$...
- Mỗi biểu thức sẽ có giá trị thuộc một kiểu dữ liệu nào đó
- Ưu tiên cao nhất cho biểu thức con trong cặp ngoặc đơn

Khai báo mảng

* Mảng một chiều:

- `datatype identifier [];`
- `datatype identifier [] = new datatype[size];`
- `datatype identifier [] = {value1,value2,...valueN};`

* Mảng hai chiều :

- `datatype identifier [][];`
- `datatype [][] identifier = new datatype [size][size];`

Lớp và phương thức (Classes & Methods)

Lớp trong Java

// comments about the class

```
public class MyProgram
```

```
{
```

class header



class body

Comments can be added almost anywhere

```
}
```

Lớp trong Java

- Cú pháp khai báo lớp (Class)

class **tênlớp**

{

Thuộc tính các đối tượng

Xây dựng các phương thức

:

Tạo đối tượng(hành động) -> gọi các phương thức đáp ứng

}

Lớp trong Java

- Cú pháp khai báo lớp (Class)

class tênlớp

{

// thuộc tính các đối tượng

kiểu tênbiến;

// xây dựng các phương thức

access_specifier modifier datatype method_name(parameter_list)

{

// body of method

}

}

Các lớp nội (Nested Classes)

- Một lớp được định nghĩa bên trong một lớp khác. Lớp đó được gọi là “lớp nội” (Nesting)
- Các kiểu lớp nội:
 - static
 - Non-static

Phương thức (Methods in Classes)

- Phương thức được định nghĩa như là một hành động hoặc một tác vụ thật sự của đối tượng
- Cú pháp

```
access_specifier modifier datatype method_name(parameter_list)
{
    // body of method
}
```

Ví dụ :

```
public static void main(String s[])
{
    .....
}
```

Ví dụ về sử dụng phương thức

```
class Ex3
```

```
{
```

```
    int x = 10;
```

```
// phương thức
```

```
    public void show( )
```

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

```
        }
```

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

```
        Ex3 t = new Ex3( );           // tạo đối tượng t
```

```
        t.show( );                     // gọi phương thức
```

```
        Ex3 t1 = new Ex3( );          // tạo đối tượng t1
```

```
        t1.x = 20;
```

```
        t1.show( );
```

```
    }
```

```
}
```

Các chỉ định truy xuất của phương thức (Access specifiers)

- Riêng tư (private)
- Ngầm định (default)
- Bảo vệ (protected)
- Công cộng (public)

Những phương thức được nạp chồng : (Methods Overloading)

- Những phương thức được nạp chồng :
 - Cùng ở trong một lớp
 - Có cùng tên
 - Khác nhau về danh sách tham số
- Những phương thức được nạp chồng là một hình thức đa hình trong quá trình biên dịch (compile time)

Ghi đè phương thức (Methods Overriding)

- Những phương thức được ghi đè:
 - Có mặt trong lớp cha (superclass) cũng như lớp kế thừa (subclass)
 - Được định nghĩa lại trong lớp kế thừa (subclass)
- Những phương thức được ghi đè là một hình thức đa hình trong quá trình thực thi (Runtime)

Phương thức khởi tạo (Class Constructors)

- Là một phương thức đặc biệt dùng để khởi tạo giá trị cho các biến thành viên của lớp đối tượng
- Có cùng tên với tên lớp và không có giá trị trả về
- Được gọi khi đối tượng được tạo ra
- Có 2 loại:
 - T tường minh (Explicit constructors)
 - Ngầm định (Implicit constructors)

Phương thức khởi tạo của lớp dẫn xuất (Derived class constructors)

- Có cùng tên với lớp dẫn xuất (subclass)
- Nếu ở lớp dẫn xuất muốn gọi hàm constructor của lớp cơ sở thì câu lệnh gọi hàm constructor của lớp cơ sở phải là câu lệnh đầu tiên trong hàm constructor của lớp dẫn xuất
- Dùng từ khoá *super*

Các lệnh điều khiển

- Điều khiển rẽ nhánh:
 - Mệnh đề **if-else**
 - Mệnh đề **switch-case**
- Vòng lặp (Loops):
 - Vòng lặp **while**
 - Vòng lặp **do-while**
 - Vòng lặp **for**
 - Từ khóa **break** và **continue**

Lệnh if-else

- Cú pháp

if (condition)

{

action1 statements;

}

[else

{

action2 statements;]

}

Lệnh **switch-case**

- Cú pháp

switch (expression)

{

case 'value1': action1 statement(s);
break;

case 'value2': action2 statement(s);
break;

:

:

case 'valueN': actionN statement(s);
break;

default: default_action statement(s);

}

Lệnh lặp while

- Cú pháp

```
while(condition)  
{  
    action statements;  
    :  
    :  
}
```


Lệnh lặp do-while

- Cú pháp

```
do  
{  
    action statements;  
    :  
    :  
} while(condition);
```

Vòng lặp **for**

- Cú pháp

```
for(initialization statements ; condition ; increment statements)
{
    action statements;
    :
    :
}
```

Từ khóa Break và Continue

- Từ khóa “**break;**” dùng để kết thúc sớm một vòng lặp. Nếu nhiều vòng lặp lồng vào với nhau thì sẽ thoát 1 vòng lặp bên trong nhất.
- Khi gặp lệnh “**continue;**” thì chương trình không thực hiện tiếp các lệnh trong thân chu trình mà quay lại bước điều kiện nhằm kiểm tra để thực hiện lại.

Chương 4

Packages & Interfaces

Giới thiệu

- **Những thành phần cơ bản của chương trình Java:**
 - Gói (Packages)
 - Giao diện (Interfaces)
- **Những phần của một chương trình Java:**
 - Lệnh khai báo gói(**package**)
 - Lệnh chỉ định gói được dùng (Lệnh **import**)
 - Khai báo lớp public (một file java chỉ chứa 1 lớp public class)
 - Các lớp khác (classes private to the package)
- Tập tin nguồn Java có thể chứa tất cả hoặc một vài trong số các thành phần trên.

Gói (Packages)

- Gói được xem tương tự như thư mục lưu trữ những lớp, những interfaces và các gói con khác.

- Những ưu điểm khi dùng gói (package):
 - Cho phép tổ chức các lớp vào những đơn vị nhỏ hơn
 - Giúp tránh được tình trạng trùng lặp khi đặt tên lớp, tên interfaces
 - Cho phép bảo vệ các lớp
 - Tên gói (package) có thể được dùng để nhận dạng chức năng của các lớp.

- Những lưu ý khi tạo gói:
 - Mã nguồn phải bắt đầu bằng lệnh 'package'
 - Mã nguồn phải nằm trong cùng thư mục mang tên của gói
 - Tên gói nên bắt đầu bằng ký tự thường (lower case) để phân biệt giữa tên lớp đối tượng và tên gói
 - Những lệnh khác phải viết phía dưới dòng khai báo gói là mệnh đề **import**, kể đến là các lệnh định nghĩa lớp đối tượng
 - Những lớp đối tượng trong gói cần phải được biên dịch.
 - Để chương trình Java có thể sử dụng những gói này, ta phải **import** gói vào trong mã nguồn

- Import gói (Importing packages):
 - Có 2 cách:
 - Xác định lớp hoặc interface của gói cần được import.
 - Hoặc có thể import toàn bộ gói
- vd : `import java.util.Vector;`
`import java.util.*;`

Các bước tạo ra gói (Package)

- Khai báo gói
- Import những gói chuẩn cần thiết
- Khai báo và định nghĩa các lớp đối tượng có trong gói
- Lưu thành tập tin **.java**, và biên dịch những lớp đối tượng đã được định nghĩa trong gói.

Sử dụng những gói do người dùng định nghĩa (user-defined packages)

- Mã nguồn của những chương trình này phải ở cùng thư mục của gói do người dùng định nghĩa. Nếu không ta phải thiết lập đường dẫn.
- Để những chương trình Java khác sử dụng những gói này, import gói vào trong mã nguồn
- Import những lớp đối tượng cần dùng
- Import toàn bộ gói
- Tạo tham chiếu đến những thành viên của gói

Thiết lập đường dẫn

Xác lập CLASSPATH

- Là danh sách các thư mục, giúp cho việc tìm kiếm các tập tin lớp đối tượng tương ứng
- Nên xác lập CLASSPATH trong lúc thực thi (runtime), vì như vậy nó sẽ xác lập đường dẫn cho quá trình thực thi hiện hành

Gói và điều khiển truy xuất (Packages & Access Control)

	<u>public</u>	<u>protected</u>	No modifier	<u>private</u>
Same class	Yes	Yes	Yes	Yes
<u>Same package subclass</u>	Yes	Yes	Yes	No
<u>Same package non-subclass</u>	Yes	Yes	Yes	No
<u>Different package subclass</u>	Yes	Yes	No	No
<u>Different package non-subclass</u>	Yes	No	No	No

Gói java.lang

- Gói **java.lang** được import mặc định.
- Những lớp Wrapper cho các kiểu dữ liệu nguyên thủy:

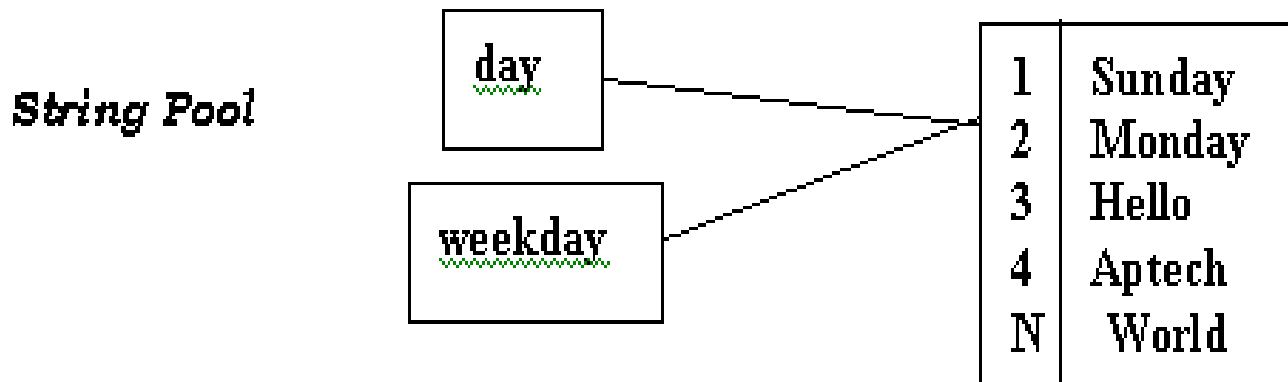
Data type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Lớp String

- Phương thức khởi tạo (Constructor):
 - **String str1 = new String();**
 - **String str2 = new String("Hello World");**
 - **char ch[] = {"A","B","C","D","E"};**
 - **String str3 = new String(ch);**
 - **String str4 = new String(ch,0,2);**

String Pool

- ‘String Pool’ đại diện cho tất cả các ký tự được tạo ra trong chương trình
- Khái niệm ‘String Pool’



Những phương thức của lớp **String**

- **charAt()**
- **startsWith()**
- **endsWith()**
- **copyValueOf()**
- **toCharArray()**
- **indexOf()**
- **toUpperCase()**
- **toLowerCase()**
- **trim()**
- **equals()**

Lớp StringBuffer

- Cung cấp những phương thức khác nhau để thao tác trên đối tượng string (chuỗi ký tự)
- Những đối tượng của lớp này khá linh hoạt
- Cung cấp những phương thức khởi tạo (constructor) đã được nạp chồng (overloaded)
- Những phương thức của lớp **StringBuffer**:
 - **append()**
 - **insert()**
 - **charAt()**
 - **setCharAt()**
 - **setLength()**
 - **getChars()**
 - **reverse()**

Lớp `java.lang.Math`

- `abs()`
- `ceil()`
- `floor()`
- `max()`
- `min()`
- `round()`
- `random()`
- `sqrt()`
- `sin()`
- `cos()`
- `tan()`

Lớp Runtime

- Đóng gói (Encapsulates) môi trường thực thi
- Dùng để quản lý bộ nhớ và thi hành những tiến trình cộng thêm
- Phương thức:
 - **exit(int)**
 - **freeMemory()**
 - **getRuntime()**
 - **gc()**
 - **totalMemory()**
 - **exec(String)**

Lớp System

- Cung cấp những luồng chuẩn như nhập (Input), xuất (Output) và các luồng lỗi(Error Streams)
- Cung cấp khả năng truy xuất đến những thuộc tính của hệ thống thực thi Java, và những thuộc tính môi trường như phiên bản, đường dẫn, nhà cung cấp...
- Phương thức:
 - **exit(int)**
 - **gc()**
 - **getProperties()**
 - **setProperties()**
 - **currentTimeMillis()**
 - **arrayCopy(Object, int, Object, int, int)**

Lớp Class

- Đối tượng của lớp này che giấu trạng thái thực thi của đối tượng trong một ứng dụng Java
- Đối tượng của lớp này có thể tạo ra bằng một trong ba cách sau:
 - Sử dụng phương thức **getClass()** của đối tượng
 - Sử dụng phương thức tĩnh **forName()** của lớp để tạo ra một đối tượng của lớp đó trong lúc đặt tên cho lớp
 - Sử dụng đối tượng **ClassLoader** để nạp một lớp mới

Lớp Object

- Là lớp cha tối cao của tất cả các lớp
- Phương thức:
 - **equals(Object)**
 - **finalize()**
 - **notify()**
 - **notifyAll()**
 - **toString()**
 - **wait()**

Gói `java.util`

- Cung cấp phần lớn những lớp Java hữu dụng và thường xuyên cần đến trong hầu hết các ứng dụng
- Giới thiệu những lớp trừu tượng sau:
 - Hashtable
 - Random
 - Vector
 - StringTokenizer

Lớp Hashtable

- Dẫn xuất từ lớp trừu tượng Dictionary
- Dùng để nối kết những khóa vào những giá trị cụ thể
- Phương thức khởi tạo Hashtable:
 - **Hashtable(int)**
 - **Hashtable(int, float)**
 - **Hashtable()**

Những phương thức của lớp **Hashtable**

- **clear()**
- **done()**
- **contains(Object)**
- **containsKey(Object)**
- **elements()**
- **get(Object key)**
- **isEmpty()**
- **keys()**
- **put(Object, Object)**
- **rehash()**
- **remove(Object key)**
- **size()**
- **toString()**

Lớp Random

- Những phương thức nhận giá trị ngẫu nhiên:
 - **nextDouble()**
 - **nextFloat()**
 - **nextGaussian()**
 - **nextInt()**
 - **nextLong()**
- Phương thức khởi tạo (Constructors):
 - **random()**
 - **random(long)**

Những phương thức của lớp **Random**

- **nextDouble()**
- **nextFloat()**
- **nextGaussian()**
- **nextInt()**
- **nextLong()**
- **setSeed(long)**

Lớp Vector

- Cung cấp khả năng co giãn cho mảng khi thêm phần tử vào mảng
- Lưu trữ những thành phần của kiểu Object
- Một Vector riêng rẽ có thể lưu trữ những phần tử khác nhau, đó là những instance của những lớp khác nhau
- Phương thức khởi tạo (Constructors):
 - **Vector(int)**
 - **Vector(int, int)**
 - **Vector()**

Những phương thức của lớp **Vector**

- **addElement(Object)**
- **capacity()**
- **clone()**
- **contains(Object)**
- **copyInto(Object [])**
- **elementAt(int)**
- **elements()**
- **ensureCapacity(int)**
- **firstElement()**
- **indexOf(Object)**
- **indexOf(Object, int)**
- **insertElementAt(Object, int)**
- **isEmpty()**
- **lastElement()**
- **lastIndexOf(Object)**
- **lastIndexOf(Object, int)**
- **removeAllElements()**
- **removeElement(Object)**
- **removeElementAt(int)**
- **setElementAt(Object, int)**
- **setSize(int)**
- **size()**
- **toString()**
- **trimToSize()**

Lớp StringTokenizer

- Có thể được dùng để tách một chuỗi thành những thành phần cấu thành của nó (constituent tokens)
- Ký tự phân cách có thể được chỉ định khi một đối tượng **StringTokenizer** được khởi tạo
- Phương thức khởi tạo (Constructors):
 - **StringTokenizer(String)**
 - **StringTokenizer(String, String)**
 - **StringTokenizer(String, String, Boolean)**
- Lớp **StringTokenizer** sử dụng giao diện liệt kê (enumeration interface)

Những phương thức của lớp **StringTokenizer**

- **countTokens()**
- **hasMoreElements()**
- **hasMoreTokens()**
- **nextElement()**
- **nextToken()**
- **nextToken(String)**

Interfaces

- Interface chính là lớp abstract thuần túy
- Trong Interface không chứa những phương thức cụ thể (concrete methods)
- Khai báo những phương thức (không có phần cài đặt) mà lớp sử dụng
- Trong Java một lớp chỉ có thể dẫn xuất từ một lớp duy nhất tại cùng một thời điểm
- Một lớp có thể dẫn xuất cùng lúc nhiều Interface
- interface cần phải được implements bởi class

Các bước tạo interface

- Định nghĩa Interface
- Biên dịch Interface
- Implements Interface
- Tính chất của interface:
 - Tất cả phương thức trong interface phải là **public**.
 - Các phương thức phải được định nghĩa trong lớp dẫn xuất interface đó.

Sử dụng Interface

- Interface không thể dẫn xuất (extends) từ lớp, nhưng có thể dẫn xuất từ những interface khác
- Nếu một lớp dẫn xuất từ một interface mà interface đó dẫn xuất từ các interface khác thì lớp đó phải định nghĩa tất cả các phương thức có trong các interface đó. Nếu không lớp đó trở thành lớp abstract
- Khi định nghĩa một interface mới có nghĩa là một kiểu dữ liệu tham chiếu mới được tạo ra.

Chương 5

Xử lý biệt lệ

Giới thiệu về biệt lệ

- Là một kiểu lỗi đặc biệt
- Nó xảy ra trong thời gian thực thi đoạn lệnh
- Thông thường các điều kiện thực thi chương trình gây ra biệt lệ
- Nếu các điều kiện này không được xử lý, thì việc thực thi có thể kết thúc đột ngột

Mục đích của việc xử lý biệt lệ

- Giảm thiểu việc kết thúc bất thường của hệ thống và của chương trình.
- Ví dụ : thao tác xuất/nhập trong một tập tin, nếu việc chuyển đổi kiểu dữ liệu không thực hiện đúng, một biệt lệ sẽ xảy ra và chương trình bị hủy mà không đóng tập tin.
- Lúc đó tập tin sẽ bị hư hại và các nguồn tài nguyên được cấp phát cho tập tin không được thu hồi lại cho hệ thống.

Xử lý biệt lệ

- Khi một biệt lệ xảy ra, đối tượng tương ứng với biệt lệ đó sẽ được tạo ra.
- Đối tượng này sau đó được truyền tới phương thức nơi mà biệt lệ xảy ra.
- Đối tượng này chứa các thông tin chi tiết về biệt lệ. Thông tin này có thể nhận được và xử lý.
- Lớp 'throwable' mà Java cung cấp là lớp trên nhất của lớp biệt lệ.

Mô hình xử lý biệt lệ

- Mô hình được biết đến là mô hình 'catch and throw'
- Khi một lỗi xảy ra, biệt lệ sẽ được chặn và được vào một khối.
- Từ khóa để xử lý biệt lệ:
 - try
 - catch
 - throw
 - throws
 - finally

Cấu trúc của mô hình xử lý biệt lệ

- **Cú pháp**

```
try { .... }
```

```
catch(Exception e1) { .... }
```

```
catch(Exception e2) { .... }
```

```
catch(Exception eN) { .... }
```

```
finally { .... }
```

Mô hình 'Catch and Throw' nâng cao

- Người lập trình chỉ quan tâm tới các lỗi khi cần thiết.
- Một thông báo lỗi có thể được cung cấp trong exception-handler.

Khối 'try' và 'catch'

- Được sử dụng để thực hiện trong mô hình 'catch and throw' của xử lý biệt lệ.
- Khối lệnh 'try' gồm tập hợp các lệnh thực thi
- Một hoặc nhiều khối lệnh 'catch' có thể tiếp theo sau một khối lệnh 'try'
- Các khối lệnh 'catch' này bắt biệt lệ trong khối lệnh 'try'.

Khởi lệnh 'try' và 'catch' (tt)

- Để bắt bất kỳ loại biệt lệ nào, ta có thể chỉ ra kiểu biệt lệ là 'Exception'

catch(Exception e)

- Khi biệt lệ bị bắt không biết thuộc kiểu nào, chúng ta có thể sử dụng lớp 'Exception' để bắt biệt lệ đó.
- Lỗi sẽ được truyền thông qua khối lệnh 'try catch' cho tới khi chúng bắt gặp một 'catch' tham chiếu tới nó, nếu không chương trình sẽ bị kết thúc

Khối lệnh chứa nhiều Catch

- Các khối chứa nhiều 'catch()' xử lý các kiểu biệt lệ khác nhau một cách độc lập.
- Ví dụ

```
try
{ doFileProcessing();
  displayResults();
} catch(LookupException e)
    { handleLookupException(e);
    }
catch(Exception e)
{ System.err.println("Error:"+e.printStackTrace()); }
```

Khối lệnh chứa nhiều Catch (tt)

- Khi sử dụng các 'try' lồng nhau, khối 'try' bên trong được thực thi trước.
- Bất kỳ biệt lệ nào bị chặn trong khối lệnh 'try' sẽ bị bắt trong khối lệnh 'catch' tiếp ngay sau.
- Nếu khối lệnh 'catch' thích hợp không được tìm thấy, thì các khối 'catch' của khối 'try' bên ngoài sẽ được xem xét
- Ngược lại, Java Runtime Environment sẽ xử lý biệt lệ.

Khối 'finally'

- Thực hiện tất cả các việc thu dọn khi biệt lệ xảy ra
- Có thể sử dụng kết hợp với khối 'try'
- Chứa các câu lệnh thu hồi tài nguyên về cho hệ thống hay lệnh in ra các câu thông báo:
 - Đóng tập tin
 - Đóng lại bộ kết quả (được sử dụng trong chương trình cơ sở dữ liệu)
 - Đóng lại các kết nối được tạo trong cơ sở dữ liệu.

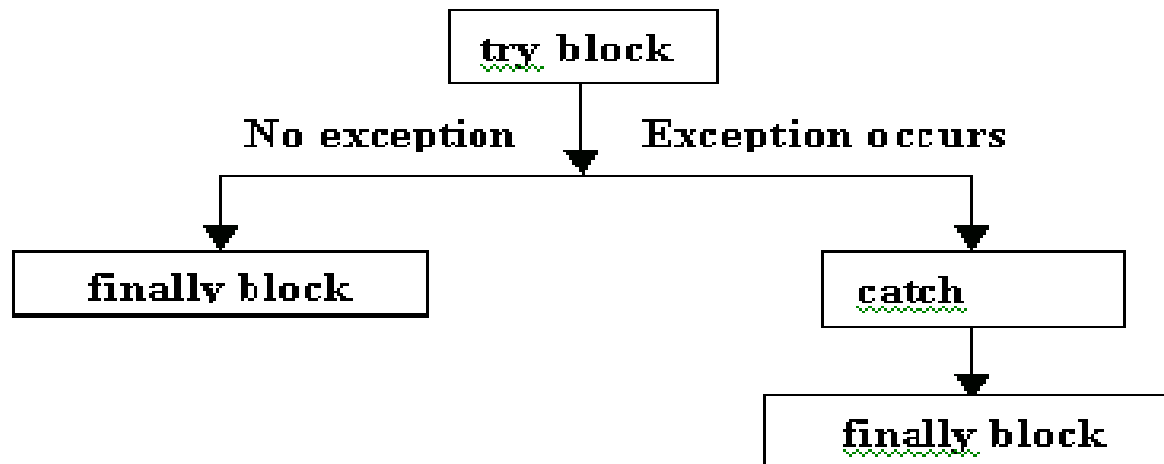
Khởi 'finally' (tt)

- Ví dụ

```
try
{
    doSomethingThatMightThrowAnException( );
}
finally
{
    cleanup( );
}
```


Khối 'finally' (tt)

- Khối này tùy chọn có hoặc không có
- Được đặt sau khối 'catch' cuối cùng.
- Khối 'finally' bảo đảm lúc nào cũng được thực hiện bất chấp biệt lệ có xảy ra hay không.



Flow of the 'try', 'catch' and 'finally' blocks

Các biệt lệ được định nghĩa với lệnh 'throw' và 'throws'

- Các biệt lệ ném ra với sự trợ giúp của từ khoá throw.
- Throw là một đối tượng của một lớp biệt lệ.
- Ví dụ của lệnh 'throw'

```
try{  
    if (flag < 0)  
    {  
        throw new MyException( ) ;// user-defined  
    }  
}
```

Các biệt lệ được định nghĩa với lệnh 'throw' và 'throws'(tt)

- Lớp 'Exception' implements interface 'Throwable' và cung cấp các tính năng hữu dụng để phân phối cho các biệt lệ.
- Một lớp con của lớp Exception là một biệt lệ mới.

Danh sách các biệt lệ

- RuntimeException
- ArithmeticException
- IllegalAccessException
- IllegalArgumentException
- ArrayIndexOutOfBoundsException
- NullPointerException
- SecurityException
- ClassNotFoundException

Danh sách các biệt lệ (tt)

- NumberFormatException
- AWTException
- IOException
- FileNotFoundException
- EOFException
- NoSuchMethodException
- InterruptedException

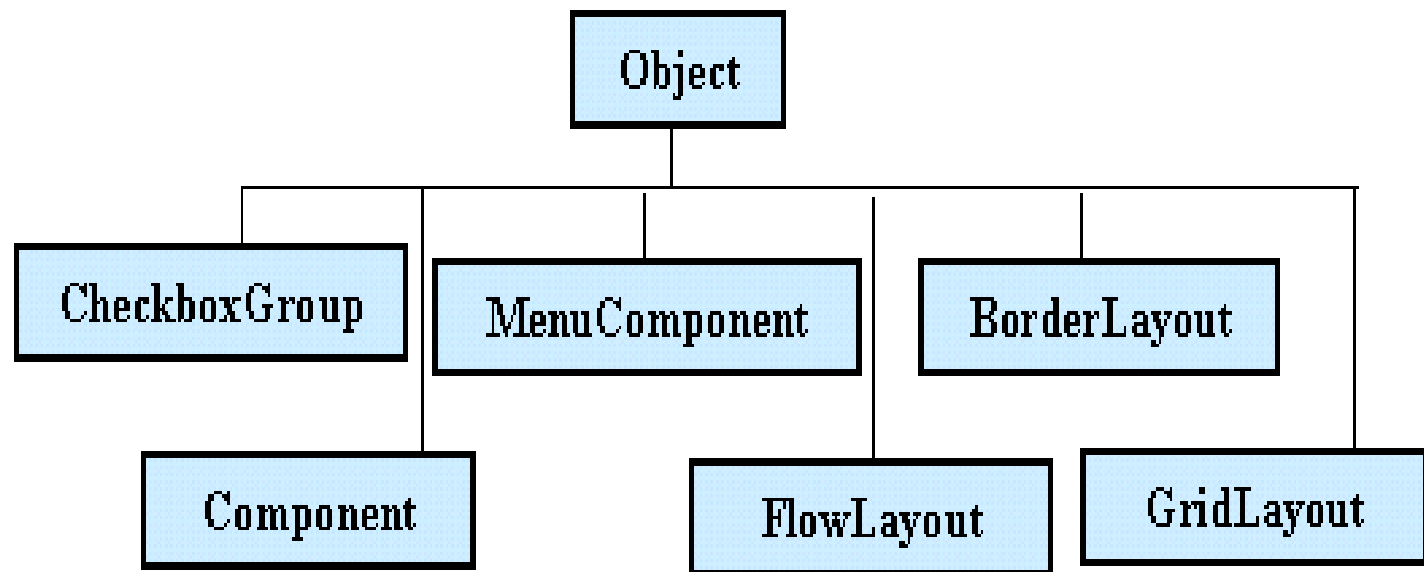
Chương 5

LẬP TRÌNH GIAO DIỆN VỚI AWT-SWING

GIỚI THIỆU VỀ AWT

- AWT viết tắt của **Abstract Windowing Toolkit**
- AWT là tập hợp các lớp Java cho phép chúng ta tạo một GUI
- Cung cấp các thành phần khác nhau để tạo chương trình GUI như:
 - Containers
 - Components
 - Layout managers
 - Graphics và drawing capabilities
 - Fonts
 - Events

- AWT bao gồm các lớp, interfaces và các gói khác



AWT class hierarchy

Components

- Tất cả các thành phần cấu tạo nên chương trình GUI được gọi là component.
- **Ví dụ**
 - Containers,
 - textfields, labels, checkboxes, textareas
 - scrollbars, scrollpanes, dialog

Containers

- Là thành phần mà có thể chứa các thành phần khác
- Có các frames, panels, latches, hooks
- Java.awt chứa một lớp có tên là Container. Lớp này được hai lớp dẫn xuất trực tiếp và không trực tiếp gồm:
 - Frames
 - Panels

Frames

- Là các cửa sổ
- Là lớp con của lớp Windows
- Được hiển thị trong một cửa sổ và có đường viền

Panels

- Là các vùng chứa trong một cửa sổ.
- Hiện thị trong một cửa sổ mà trình duyệt hoặc appletviewer cung cấp và không có đường viền.
- Được sử dụng để nhóm một số các thành phần
- Một panel không thể nhìn thấy vì thế chúng ta cần phải thêm nó vào frame.

Dialog

- Là một lớp con của lớp Window
- Đối tượng dialog được cấu trúc như sau :

Frame myframe = new Frame(“My frame”);

String title = “Title”;

boolean modal = true;

Dialog dlg = new Dialog(myframe, title, modal);

Các Components khác

- Ví dụ
 - textfields, labels, checkboxes, textareas
 - scrollbars, scrollpanes, dialog

Label

- Được dùng để hiển thị chuỗi (String)
- Các hàm tạo dựng:
 - **Label()**
 - **Label(String labeltext)**
 - **Label(String labeltext, int alignment)**
- Các phương thức:
 - **setFont(Font f)**
 - **setText(String s)**
 - **getText()**

TextField

- Là điều khiển text, cho phép hiển thị text hoặc cho user nhập dữ liệu vào.
- Các hàm dựng:
 - **TextField()**
 - **TextField(int columns)**
 - **TextField(String s)**
 - **TextField(String s, int columns)**
- Các phương thức:
 - **setEchoChar(char)**
 - **setText(String s)**
 - **getText()**
 - **setEditable(boolean)**
 - **isEditable()**

TextArea

- Được dùng khi chuỗi hiển thị có từ hai dòng trở lên.

TextArea (tt...)

- Các hàm dựng:
 - **TextArea()**
 - **TextArea(int rows, int cols)**
 - **TextArea(String text)**
 - **TextArea(String text, int rows, int cols)**

Các phương thức của TextArea

- **setText(String)**
- **getText()**
- **setEditable(boolean)**
- **isEditable()**
- **insertText(String, int)**
- **replaceText(String, int, int)**

Button

- Các nút ấn là cách dễ nhất để lấy các sự kiện của người dùng.
- Các hàm tạo dựng:
 - **Button()**
 - **Button(String text)**

Checkboxes và RadioButtons

- Checkboxes cho phép người dùng đưa ra nhiều chọn lựa
- Radiobuttons chỉ cho phép người dùng duy nhất một chọn lựa
- Các hàm dựng để tạo checkbox:
 - **Checkbox()**
 - **Checkbox(String text)**
- Để tạo radiobutton, ta phải tạo đối tượng **CheckBoxGroup** trước khi tạo button

Choice

- Lớp 'Choice' cho phép ta tạo danh sách có nhiều chọn lựa
- Ví dụ

```
Choice colors=new Choice( );  
colors.addItem("Red");  
colors.addItem("Green");
```

Layout Manager

- Các loại layout manager khác nhau:
 - Flow Layout
 - Border Layout
 - Card Layout
 - Grid Layout
 - GridBag Layout
- Layout manager được thiết lập bằng cách gọi phương thức 'setLayout()'

FlowLayout

- FlowLayout là layout manager mặc định cho các applet và các panel
- Với FlowLayout các component sẽ được sắp xếp từ góc trái trên đến góc phải dưới của màn hình theo từng hàng.
- Các constructor:

FlowLayout mylayout = new FlowLayout();

**FlowLayout exLayout = new
FlowLayout(FlowLayout.RIGHT);**

BorderLayout

- BorderLayout mặc định cho Window, Frame và Dialog
- Trình quản lý này có thể bố trí container thành 5 vùng, NORTH, EAST, SOUTH, WEST và CENTER.
- **Ví dụ:** Để thêm một thành phần vào vùng North của container

```
Button b1= new Button("North Button");  
setLayout(new BorderLayout( ));  
add(b1, BorderLayout.NORTH);
```

CardLayout

- Có thể lưu trữ một danh sách các kiểu layout khác nhau
- Mỗi layout được xem như một thẻ (card)
- Thẻ thường là đối tượng Panel
- Một thành phần độc lập như button sẽ điều khiển các thẻ được đặt ở phía trên nhất
- Các bước để tạo CardLayout:
 - Bố trí layout của panel chính là CardLayout
 - Lần lượt thêm các panel khác vào panel chính

GridLayout

- Hỗ trợ việc chia container thành một lưới
- Các thành phần được bố trí trong các ô của lưới.
- Một ô lưới nên chứa ít nhất một thành phần
- Kiểu layout này được sử dụng khi tất cả các components có cùng kích thước
- Hàm constructor
GridLayout gl = new GridLayout(no. of rows, no. of columns);

GridBagLayout

- Bố trí các thành phần một cách chính xác
- Các thành phần không cần có cùng kích thước
- Các thành phần được sắp xếp trong một lưới chứa các dòng và các cột
- Thứ tự đặt các thành phần không tuân theo hướng từ trái-sang-phải và trên-xuống-dưới
- Hàm constructor
GridBagLayout gb = new GridBagLayout();

GridBagLayout

- Để sử dụng layout này, bạn cần phải biết thông tin về kích cỡ và cách bố trí của các thành phần
- Lớp 'GridBagLayoutConstraints' lưu trữ tất cả các thông tin mà lớp GridLayout yêu cầu: Vị trí và kích thước mỗi thành phần

Các lớp quản lý cách tổ chức các thành phần giao diện

Tên lớp	Mô tả
FlowLayout	Xếp các thành phần giao diện trước tiên theo hàng từ trái qua phải, sau đó theo cột từ trên xuống dưới. Cách sắp xếp này là mặc định đối với Panel và Applet.
GridLayout	Các thành phần giao diện được sắp xếp trong các ô lưới hình chữ nhật lần lượt theo hàng từ trái qua phải và theo cột từ trên xuống dưới trong một phần tử chứa. Mỗi thành phần giao diện chứa trong một ô.
BorderLayout	Các thành phần giao diện (ít hơn 5) được đặt vào các vị trí theo các hướng: north (bắc), south (nam), west (tây), east (đông) và center (trung tâm)). Cách sắp xếp này là mặc định đối với lớp Window, Frame và Dialog.
GridBagLayout	Cho phép đặt các thành phần giao diện vào lưới hình chữ nhật, nhưng một thành phần có thể chiếm nhiều ô.

Các phương pháp thiết kế *layout*

Để biết *layout* hay để đặt lại *layout* cho chương trình ứng dụng, chúng ta có thể sử dụng hai hàm của lớp *Container*:

LayoutManager getLayout();

void setLayout(LayoutManager mgr);

Các thành phần giao diện sau khi đã được tạo ra thì phải được đưa vào một phần tử chứa nào đó. Hàm *add()* của lớp *Container* được nạp chồng để thực hiện nhiệm vụ đưa các thành phần vào phần tử chứa.

Component add(Component comp)

Component add(Component comp, int index)

Component add(Component comp, Object constraints)

Component add(Component comp, Object constraints, int index)

Trong đó, đối số *index* được sử dụng để chỉ ra vị trí của ô cần đặt thành phần giao diện *comp* vào. Đối số *constraints* xác định các hướng để đưa *comp* vào phần tử chứa.

Ngược lại, khi cần loại ra khỏi phần tử chứa một thành phần giao diện thì sử dụng các hàm sau:

void remove(int index)

void remove(Component comp)

void removeAll()

Lớp **FlowLayout**

Lớp *FlowLayout* có các hàm tạo lập để sắp hàng các thành phần giao diện:

FlowLayout()

FlowLayout(int alignment)

FlowLayout(int alignment, int horizontalgap, int verticalgap)

public static final int LEFT

public static final int CENTER

public static final int RIGHT

Lớp **GridLayout**

Lớp *GridLayout* cung cấp các hàm tạo lập để sắp hàng các thành phần giao diện:

GridLayout()

GridLayout(int rows, int columns)

GridLayout(int rows, int columns, int hoiongap, int verticalgap)

Tạo ra một lưới hình chữ nhật có $rows \times columns$ ô có khoảng cách giữa các hàng các cột là *horizongap*, *verticalgap*. Một trong hai đối số *rows* hoặc *columns* có thể là 0, nhưng không thể cả hai, *GridLayout(1,0)* là tạo ra lưới có một hàng.

Ví dụ: Mô tả cách sử dụng *GridLayout*

```
import java.awt.*;
import java.applet.*;
public class GridLayoutApplet extends Applet {
    public void init() {
        //Cread a list of colors
        Label xLabel = new Label("X coordinate: ");
        Label yLabel = new Label("Y coordinate: ");
        TextField xInput = new TextField(5);
        TextField yInput = new TextField(5);
```

<Tiếp theo>

// Tạo ra lưới hình chữ nhật có 4 ô và đặt *layout* để sắp xếp các thành phần //
xLabel, xInput, yLabel, yInput

```
setLayout(new GridLayout(2,2) );
```

```
add(xLabel) ; // Đặt xLabel vào ô thứ nhất
```

```
add(xInput) ; // Đặt xInput vào ô thứ hai
```

```
add(yLabel) ; // Đặt yLabel vào ô thứ ba
```

```
add(yInput) ; // Đặt yInput vào ô thứ tư
```

```
}
```

```
}
```

X coordinate:

Y coordinate:

Lớp BorderLayout

Lớp ***BorderLayout*** cho phép đặt một thành phần giao diện vào một trong bốn hướng: *bắc (NORTH)*, *nam (SOUTH)*, *đông (EAST)*, *tây (WEST)* và *ở giữa (CENTER)*.

BorderLayout()

BorderLayout(int horizongap, int verticalgap)

Tạo ra một *layout* mặc định hoặc có khoảng cách giữa các thành phần (tính bằng *pixel*) là *horizongap* theo hàng và *verticalgap* theo cột.

Trường hợp mặc định là *CENTER*, ngược lại, có thể chỉ định hướng để đặt các thành phần *comp* vào phần tử chứa theo *constraint* là một trong các hằng trên.

Ví dụ: Đặt một nút *Button* có tên “Vẽ” ở trên (NORT), trường văn bản “Hiển thị thông báo” ở dưới (SOUTH) và hai thanh trượt (SCROLLBAR) ở hai bên cạnh (WEST và EAST).

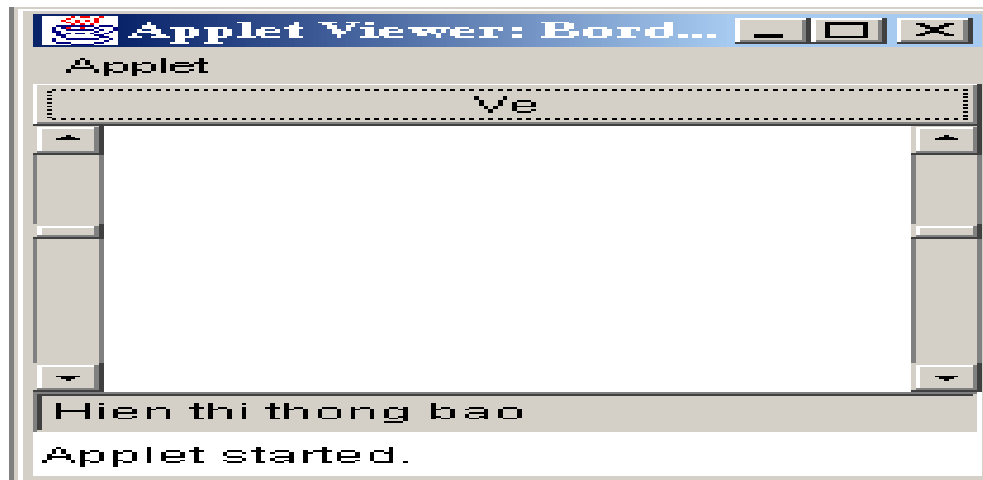
```
import java.awt.*;  
  
import java.applet.*;  
  
public class BorderLayoutApplet extends Applet {  
    public void init() {  
        TextField msg = new TextField("Hien thi thong bao");  
        msg.setEditable(false);  
        Button nutVe = new Button("Ve");  
        Canvas vungVe = new Canvas();  
        vungVe.setSize(150, 150); // Đặt kích thước cho vungVe vẽ tranh  
        vungVe.setBackground( Color.white); // Đặt màu nền là trắng
```

<Tiếp theo>

```
Scrollbar sb1=new Scrollbar(Scrollbar.VERTICAL,0,10,-50,100);  
Scrollbar sb2=new Scrollbar(Scrollbar.VERTICAL,0,10,-50,100);  
setLayout (new BorderLayout ( ) ) ;  
add(nutVe, BorderLayout.NORTH);// Đặt nutVe ở trên (NORTH)  
add(msg, BorderLayout.SOUTH); // Đặt msg ở dưới (SOUTH)  
add(vungVe,BorderLayout.CENTER);// Đặt vungVe ở giữa (CENTER)  
add(sb1, BorderLayout.WEST);      // Đặt sb1 ở bên trái (WEST)  
add(sb2, BorderLayout.EAST);      // Đặt sb2 ở bên phải (EAST)
```

```
}
```

```
}
```



Xử lý sự kiện

- Các ứng dụng với GUI thường được hướng dẫn bởi các sự kiện (*event*). Việc nhấn một nút, mở, đóng các Window hay gõ các ký tự từ bàn phím, v.v. đều tạo ra các sự kiện (*event*) và được gửi tới cho chương trình ứng dụng.
- Các sự kiện (Events) được xử lý bằng các công cụ sau:
 - Abstract Windowing Toolkit
 - Trình duyệt.
 - Các trình xử lý sự kiện do người dùng tạo riêng.
- Các ứng dụng cần đăng ký trình xử lý sự kiện với đối tượng
- Các trình xử lý này được gọi khi có một sự kiện tương ứng xảy ra

Xử lý sự kiện

- Event Listener sẽ lắng nghe một sự kiện cụ thể mà một đối tượng tạo ra
- Mỗi event listener cung cấp các phương thức để xử lý các sự kiện này
- Lớp có cài đặt listener cần định nghĩa những phương thức này

Xử lý sự kiện

- Các bước cần tuân thủ để sử dụng mô hình Event Listener:
 - Cài đặt Listener tương ứng
 - Nhận diện được tất cả các thành phần tạo sự kiện
 - Nhận diện được tất cả các sự kiện được xử lý
 - Cài đặt các phương thức của listener, và viết các đoạn mã để xử lý sự kiện trong các phương thức đó
- Interface định nghĩa các phương thức khác nhau để xử lý mỗi sự kiện

Các sự kiện và Listener tương ứng

- ActionEvent
 - AdjustmentEvent
 - ComponentEvent
 - FocusEvent
 - ItemEvent
 - WindowEvent
 - TextEvent
 - MouseEvent
 - KeyEvent
- ActionListener
 - AdjustmentListener
 - ComponentListener
 - FocusListener
 - ItemListener
 - WindowListener
 - TextListener
 - MouseListener
 - MouseMotionListener
 - KeyListener

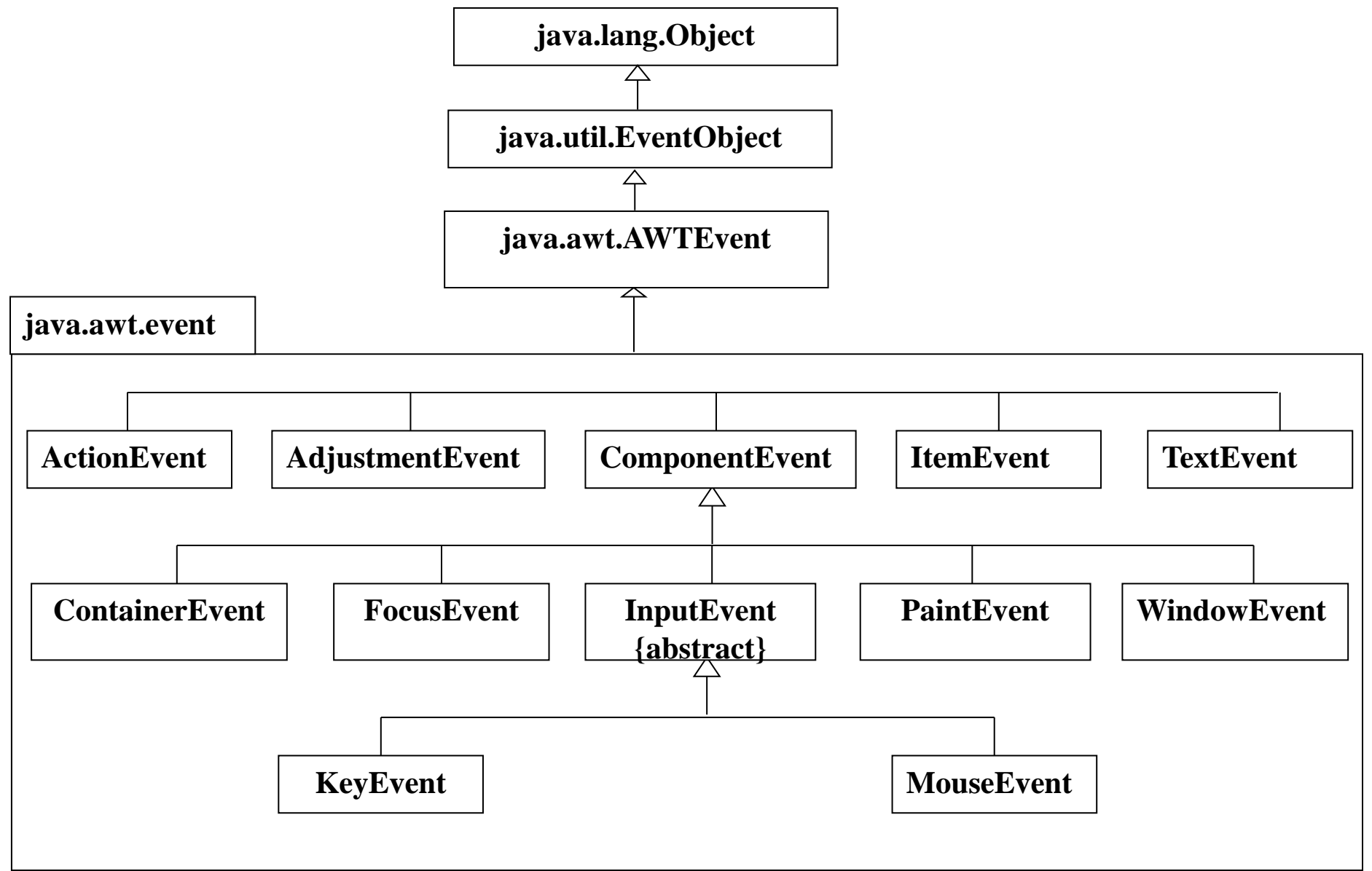
Xử lý sự kiện

➤ Trong Java các sự kiện được thể hiện bằng các đối tượng. Lớp cơ sở nhất, lớp cha của tất cả các lớp con của các sự kiện là lớp *java.util.EventObject*

➤ Các lớp con của *AWTEvent* được chia thành hai nhóm:

1. Các lớp mô tả về ngữ nghĩa của các sự kiện
2. Các lớp sự kiện ở mức thấp

Các lớp xử lý sự kiện



1. Các lớp ngữ nghĩa

a/ **ActionEvent**

Sự kiện này được phát sinh bởi những hoạt động thực hiện trên các thành phần của GUI. Các thành phần gây ra các sự kiện hành động bao gồm:

Button - khi một nút button được kích hoạt

List - khi một mục trong danh sách được kích hoạt đúp

MenuItem - khi một mục trong thực đơn được chọn

TextField - khi gõ phím ENTER trong trường văn bản (*text*).

Lớp **ActionEvent** có các hàm:

String getActionCommand()

Cho biết tên của lệnh tương ứng với sự kiện xảy ra, là tên của nút, mục hay text.

b/ **AdjustmentEvent**

Thành phần gây ra sự kiện căn chỉnh (adjustment):

Scrollbar - khi thực hiện một lần căn chỉnh trong thanh trượt *Scrollbar*.

Lớp này có hàm:

int getValue()

Cho lại giá trị hiện thời được xác định bởi lần căn chỉnh sau cùng.

1. Các lớp ngữ nghĩa

c/ **ItemEvent**

Các thành phần của GUI gây ra các sự kiện về các mục gồm có:

Checkbox - khi trạng thái của hộp kiểm tra *Checkbox* thay đổi

CheckboxMenuItem - khi trạng thái của hộp kiểm tra *Checkbox* ứng với mục của thực đơn thay đổi,

Choice - khi một mục trong danh sách được chọn hoặc bị loại bỏ

List - khi một mục trong danh sách được chọn hoặc bị loại bỏ.

Lớp *ItemEvent* có hàm:

Object getItem()

Cho lại đối tượng được chọn hay vừa bị loại bỏ.

d/ **TextEvent**

Các thành phần của GUI gây ra các sự kiện về *text* gồm có:

TextArea - khi kết thúc bằng nhấn nút ENTER,

TextField - khi kết thúc bằng nhấn nút ENTER.

1. Các lớp ngữ nghĩa

Kiểu sự kiện	Nguồn gây ra sự kiện	Các hàm đón nhận và di dời các sự kiện	Giao diện Listener tương ứng
AcitionEvent	Button	addAcitionlistener	AcitionListener
AdjustmentEvent	List	remove ActiontListener	
	TextField		
	Scrollbar	addAdjustmentListener	AdjustmentListener
		removeAdjustmentListener	
ItemEvent	Choice	addItemListener	ItemListener
	Checkbox	removeItemListener	
	CheckboxMen		
	uItem		
TextEvent	List	addTexListener	TextListener
	TextArea	removeTextListener	
	TextField		

2. Các sự kiện ở mức thấp

a/ **ComponentEvent**

Sự kiện này xuất hiện khi một thành phần bị che giấu, hiển thị hay thay đổi lại kích thước. Lớp *ComponentEvent* có hàm:

Component **getComponent()**

Cho lại đối tượng tham chiếu kiểu *Component*.

b/ **ContainerEvent**

Sự kiện này xuất hiện khi một thành phần được bổ sung hay bị loại bỏ khỏi phần tử chứa (*Container*).

c/ **FocusEvent**

Sự kiện này xuất hiện khi một thành phần nhận được một nút từ bàn phím.

d/ **KeyEvent**

Lớp *KeyEvent* là lớp con của lớp trừu tượng *InputEvent* được sử dụng để xử lý các sự kiện liên quan đến các phím của bàn phím.

int **getKeyCode()**

Đối với các sự kiện **KEY_PRESSED** hoặc **KEY_RELEASED**, hàm này được sử dụng để nhận lại giá trị nguyên tương ứng với mã của phím trên bàn phím.

char **getKeyChar()**

Đối với các sự kiện **KEY_PRESSED**, hàm này được sử dụng để nhận lại giá trị nguyên, mã *Unicode* tương ứng với ký tự của bàn phím.

2. Các sự kiện ở mức thấp

e/ MouseEvent

Lớp *MouseEvent* là lớp con của lớp trừu tượng *InputEvent* được sử dụng để xử lý các tín hiệu của chuột. Lớp này có các hàm:

int getX()

int getY()

Point getPoint()

Các hàm này được sử dụng để nhận lại tọa độ x, y của vị trí liên quan đến sự kiện do chuột gây ra.

void translatePoint(int dx, int dy)

Hàm *translate()* được sử dụng để chuyển tọa độ của sự kiện do chuột gây ra đến (dx, dy).

int getClickCount()

Hàm *getClickCount()* đếm số lần kích chuột.

f/ PaintEvent

Sự kiện này xuất hiện khi một thành phần gọi hàm *paint()/ update()* để vẽ.

g/ WindowEvent

Sự kiện loại này xuất hiện khi thao tác với các Window. Lớp này có hàm:

Window getWindow()

Hàm này cho lại đối tượng của lớp Window ứng với sự kiện liên quan đến Window đã xảy ra.

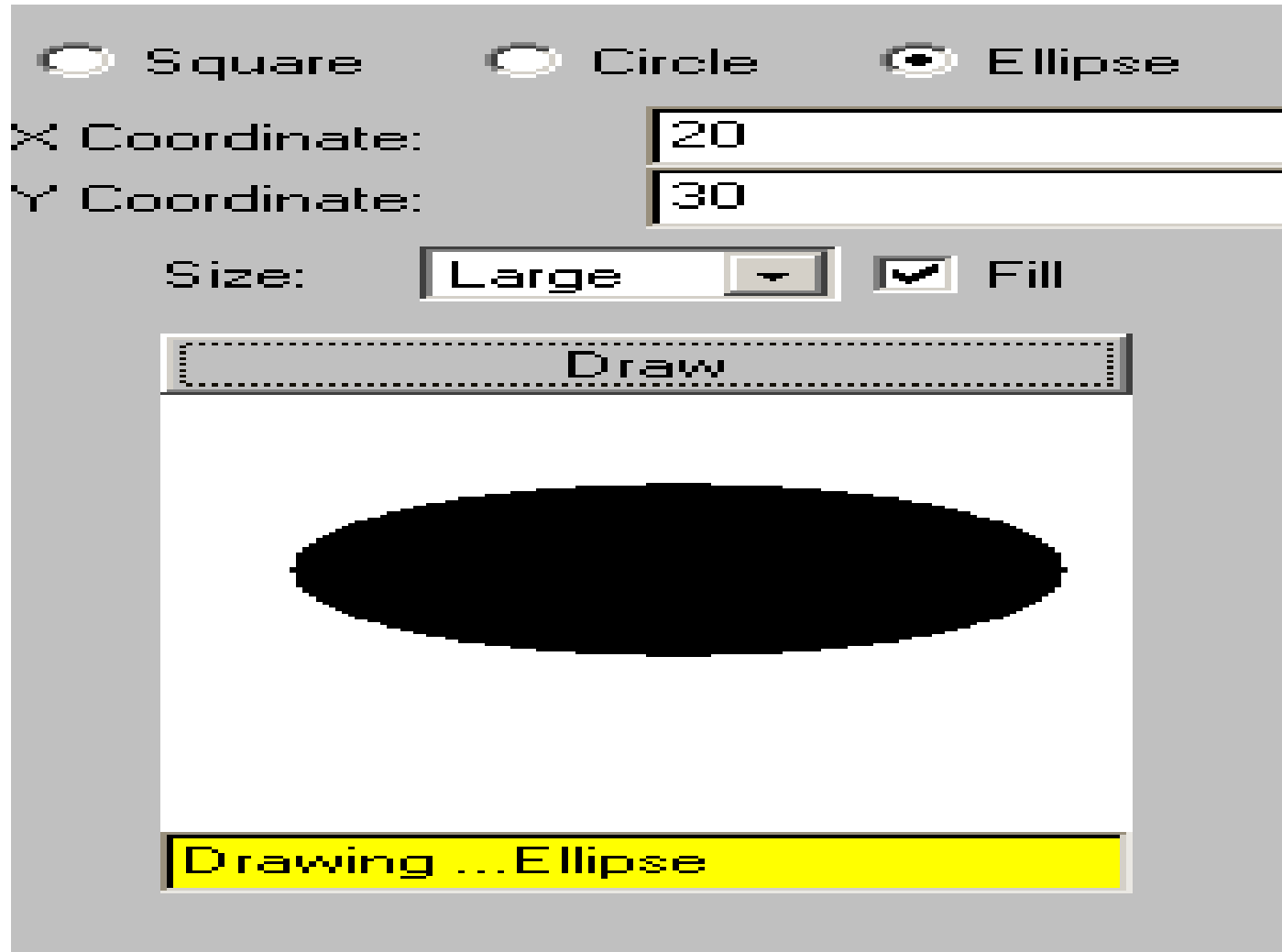
2. Các sự kiện ở mức thấp

Các hàm xử lý các sự kiện ở mức thấp

Kiểu sự kiện	Nguồn gây ra sự kiện	Các hàm đón nhận và di dời các sự kiện	Giao diện Listener tương ứng
ComponentEvent	Component	addComponentListener	ComponentListener
ContainerEvent	Container	removeComponentListener addContainerListener removeContainerListener	ContainerListener
FocusEvent	Component	addFocusListener removeFocusListener	FocusListener
KeyEvent	Component	addKeyListener removeKeyListener	KeyListener
MouseEvent	Component	addMouseListener remoMouseListener addMouseMotionListener remoMouseMotionListener	MouseMotionListener
WindowEvent	Window	addWindowListener removeWindowListener	WindowListener

2. Các sự kiện ở mức thấp

Ví dụ: Xây dựng ứng dụng có giao diện đồ họa GUI



2. Các sự kiện ở mức thấp

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
interface IGeometryConstants { // Định nghĩa Interface gồm các hằng cho trước
    int SQUARE    = 0;
    int CIRCLE    = 1;
    int ELLIPSE   = 2;
    String[] shapeNames = {"Square", "Circle" , "Ellipse"};
    int SMALL     = 0;
    int MEDIUM   = 1;
    int LARGE     = 2;
    String[] sizeNames = {"Small", "Medium" , "Large"};
}
public class DemoApplet extends Applet implements
    IGeometryConstants {
    // Khai báo một Panel để chứa các Checkbox tương ứng với các loại hình shape
    Panel          shapePanel; // Khai báo một Panel
    CheckboxGroup  shapeCBG;   // Khai báo một CheckboxGroup
    Checkbox       squareCB;   // Khai báo một Checkbox ứng với hình vuông
    Checkbox       circleCB;   // Khai báo một Checkbox ứng với hình tròn
    Checkbox       ellipseCB;  // Khai báo một Checkbox ứng với hình ellipse
    // Khai báo một Panel để chứa các Label và các TextField
    Panel          xyPanel;    // Khai báo một Panel xyPanel
    Label          xLabel;     // Khai báo một Label xLabel
    TextField      xInput;     // Khai báo một TextField xInput
    Label          yLabel;     // Khai báo một Label yLabel
    TextField      yInput;     // Khai báo một TextField yInput
```

2. Các sự kiện ở mức thấp

```
// Khai báo một Panel để chứa các Label , Choice và Checkbox
Panel          sizePanel;
Label          sizeLabel;
Choice         sizeChoices;
Checkbox fillCB;
// Khai báo một Panel để chứa shape, coordinates, size and fill Panel leftPanel
Panel          leftPanel;
// Khai báo một Panel để chứa Message display, draw button and canvas
Panel          rightPanel;
Button         drawButton;
DrawRegion    drawRegion;
TextField messageDisplay;
// Định nghĩa lại hàm init()
public void init() {
    makeShapePanel();
    makeXYPanel();
    makeSizePanel();
    makeLeftPanel();
    makeRightPanel();
    // Gọi hàm xử lý các sự kiện
    addListeners();
    // Đưa các Panel vào hệ thống
    add(leftPanel);
    add(rightPanel);
}
```

2. Các sự kiện ở mức thấp

// Đưa các thành phần về các *shape* vào *shapePanel*

```
void makeShapePanel() {  
    shapePanel = new Panel();  
    shapeCBG    = new CheckboxGroup();  
    squareCB    = new Checkbox(shapeNames[SQUARE], shapeCBG, true);  
    circleCB    = new Checkbox(shapeNames[CIRCLE], shapeCBG, false);  
    ellipseCB   = new Checkbox(shapeNames[ELLIPSE], shapeCBG, false);  
    shapePanel.setLayout(new FlowLayout());  
    shapePanel.add(squareCB);  
    shapePanel.add(circleCB);  
    shapePanel.add(ellipseCB);  
}
```

// Đưa các thành phần về các *x,y coordinates* vào *xyPanel*

```
void makeXYPanel() {  
    xyPanel = new Panel();  
    xLabel = new Label("X Coordinate:");  
    yLabel = new Label("Y Coordinate:");  
    xInput = new TextField(5);  
    yInput = new TextField(5);  
    xyPanel.setLayout(new GridLayout(2,2));  
    xyPanel.add(xLabel);  
    xyPanel.add(xInput);  
    xyPanel.add(yLabel);  
    xyPanel.add(yInput);  
}
```

2. Các sự kiện ở mức thấp

// Đưa các thành phần về *size* và *fill* vào *sizePanel*

```
void makeSizePanel() {  
    sizePanel = new Panel();  
    sizeLabel = new Label("Size:");  
    sizeChoices = new Choice();  
    sizeChoices.add(sizeNames[0]);  
    sizeChoices.add(sizeNames[1]);  
    sizeChoices.add(sizeNames[2]);  
    fillCB = new Checkbox("Fill", false);  
    sizePanel.setLayout(new FlowLayout());  
    sizePanel.add(sizeLabel);  
    sizePanel.add(sizeChoices);  
    sizePanel.add(fillCB);  
}
```

// Đưa các thành phần vào *leftPanel*

```
void makeLeftPanel() {  
    leftPanel = new Panel();  
    leftPanel.setLayout(new BorderLayout());  
    leftPanel.add(shapePanel, "North");  
    leftPanel.add(xyPanel, "Center");  
    leftPanel.add(sizePanel, "South");  
}
```

2. Các sự kiện ở mức thấp

// Đưa các thành phần vào *rightPanel*

```
void makeRightPanel() {  
    rightPanel = new Panel();  
    messageDisplay = new TextField("MESSAGE DISPLAY");  
    messageDisplay.setEditable(false);  
    messageDisplay.setBackground(Color.yellow);  
  
    drawButton = new Button("Draw");  
    drawButton.setBackground(Color.lightGray);  
  
    drawRegion = new DrawRegion();  
    drawRegion.setSize(150, 150);  
    drawRegion.setBackground(Color.white);  
    rightPanel.setLayout(new BorderLayout());  
    rightPanel.add(drawButton, BorderLayout.NORTH);  
    rightPanel.add(messageDisplay, BorderLayout.SOUTH);  
    rightPanel.add(drawRegion, BorderLayout.CENTER);  
}
```


2. Các sự kiện ở mức thấp

// Xử lý các sự kiện tương ứng với các việc thực hiện tròn các thành phần đồ họa

```
void addListeners() {  
    drawButton.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent evt) {  
            int shape, xCoord, yCoord, width;  
            messageDisplay.setText("");  
            if(squareCB.getState())  
                shape = SQUARE;  
            else if (circleCB.getState())  
                shape = CIRCLE;  
            else if (ellipseCB.getState())  
                shape = ELLIPSE;  
            else {  
                messageDisplay.setText("Unknow shape");  
                return;  
            }  
            try{  
                xCoord = Integer.parseInt(xInput.getText());  
                yCoord = Integer.parseInt(yInput.getText());  
            }catch(NumberFormatException e){  
                messageDisplay.setText("Illegal coordinate");  
                return;  
            }  
        }  
    });  
}
```

2. Các sự kiện ở mức thấp

```
        switch(sizeChoices.getSelectedIndex()){
            case SMALL: width = 30; break;
            case MEDIUM: width = 60; break;
            case LARGE: width = 120; break;
            default:
                messageDisplay.setText("Unknow size");
                return;
        }

        messageDisplay.setText("Drawing..." + shapeNames[shape]);

        drawRegion.doDraw(shape, xCoord, yCoord,
                           fillCB.getState(), width);
    }
});
xInput.addTextListener(new TextListener() {
    public void textValueChanged(TextEvent evt) {
        checkTF(xInput);
    }
});
yInput.addTextListener(new TextListener() {
    public void textValueChanged(TextEvent evt) {
        checkTF(yInput);
    }
});
} // Kết thúc hàm addListener()
```

2. Các sự kiện ở mức thấp

```
void checkTF(TextField tf){
    messageDisplay.setText("");
    try{
        Integer.parseInt(tf.getText());
    }catch(NumberFormatException e){
        messageDisplay.setText("Illegal coordinate ");
    }
}
```

// Định nghĩa lớp *DrawRegion* mở rộng lớp *Canvas* và sử dụng các hàng ở giao diện

```
class DrawRegion extends Canvas implements IGeometryConstants{
    private int shape, xCoord, yCoord;
    private boolean fillFlag;
    private int width;
    public DrawRegion(){
        setSize(150, 150);
        setBackground(Color.white);
    }
    // Định nghĩa hàm doDraw() để vẽ
    public void doDraw(int h, int x, int y, boolean f, int w){
        this.shape = h;
        this.xCoord = x;
        this.yCoord = y;
        this.fillFlag = f;
        this.width = w;
        repaint();
    }
}
```

2. Các sự kiện ở mức thấp

```
public void paint(Graphics g){
    switch(shape){
        case SQUARE:
            if(fillFlag) g.fillRect(xCoord, yCoord, width,width);
            else g.drawRect(xCoord, yCoord, width,width);
            break;
        case CIRCLE:
            if(fillFlag) g.fillOval(xCoord, yCoord, width,width);
            else g.drawOval(xCoord, yCoord, width,width);
            break;
        case ELLIPSE:
            if(fillFlag) g.fillOval(xCoord,yCoord,width,width/2);
            else g.drawOval(xCoord, yCoord, width,width/2);
            break;
    }
}
```

Menus

- Các loại menu :
 - Pull-down
 - Pop-up menu
- Chỉ có thể đặt các thanh menubar vào trong các Frame mà thôi
- Các thành phần của menu:
 - Menubar
 - Menuitems

Thành phần *Menu*

Lớp *abstract class MenuComponent* là lớp cơ sở cho tất cả các lớp thực hiện những vấn đề liên quan đến thực đơn (*menu*).

Lớp *MenuBar* cài đặt thanh thực đơn và trong đó có thể chứa các thực đơn pull-down.

Lớp *MenuItem* định nghĩa từng mục của thực đơn.

Lớp *Menu* cài đặt các thực đơn *pull-down* để có thể đưa vào một thực đơn bất kỳ.

Lớp *PopUpMenu* biểu diễn cho thực đơn pop-up .

Lớp *CheckboxMenuItem* chứa các mục được chọn để kiểm tra trong các mục.

Việc tạo lập một thanh thực đơn cho một *frame* được thực hiện như sau:

1. *Tạo ra một thanh thực đơn,*

```
MenuBar thanhThDon = new MenuBar();
```

2. *Tạo ra một thực đơn,*

```
Menu thucDon = new Menu("Cac loai banh");
```

3. *Tạo ra các mục trong thực đơn và đưa vào thực đơn,*

```
MenuItem muc = new MenuItem("Bánh dày");
```

```
thucDon.add(muc); // Đưa muc vào thucDon
```

4. *Đưa các thực đơn vào thanh thực đơn,*

```
thanhThDon.add(thucDon); // Đưa thucDon vào thanhThDon
```

5. *Tạo ra một frame và đưa thanh thực đơn vào frame đó.*

```
Frame frame = new Frame("Các món ăn");
```

```
frame.add(thanhThDon); // Đưa thanhThDon vào frame
```

Thành phần *Menu*

Ví dụ: Tạo lập và sử dụng các thực đơn.

```
import java.awt.*;
import java.applet.*;
public class MenuDemo extends Applet{
    public static void main(String args[]){
        MenuBar thanhThDon = new MenuBar();
        Menu thucDon = new Menu("Cac loai banh");
        MenuItem b1 = new MenuItem("Banh day");
        thucDon.add(b1);
        MenuItem b3 = new MenuItem("Banh khoai");
        thucDon.add(b3);
        thucDon.addSeparator(); // Tạo ra một thanh phõn cách
        MenuItem b4 = new MenuItem("Banh gio");
        thucDon.add(b4);
        thucDon.add(new CheckboxMenuItem("Banh ran"));
        thanhThDon.add(thucDon);
        Frame frame = new Frame("Cac mon an");
        frame.setMenuBar(thanhThDon);
        frame.pack(); frame.setVisible(true);}
}
```

Chương 6

Applets

Applets

- Là một chương trình Java mà chạy với sự hỗ trợ của trình duyệt web hoặc appletviewer
- Tất cả các applets là lớp con của lớp 'Applet'
- Để tạo một applet, bạn cần import gói sau:
 - **java.applet**

Cấu trúc applet

- Định nghĩa một applet từ bốn sự kiện xảy ra trong quá trình thực thi
- Đối với mỗi sự kiện được định nghĩa bởi một phương thức tương ứng.
- Các phương thức:
 - **init()**
 - **start()**
 - **stop()**
 - **destroy()**

- Các phương thức khác:
 - **paint()**
 - **repaint()**
 - **showStatus()**
 - **getAppletInfo()**
- Các phương thức `init()`, `start()`, `stop()`, `destroy()`, and `paint()` được thừa kế từ `applet`.
- Mỗi phương thức này mặc định là rỗng. Vì thế các phương thức này phải được nạp chồng.

Biên dịch và thực thi applet

- Một applet thì được biên dịch theo cú pháp sau

javac Applet1.java

- Để thực thi một applet, tạo một tập tin HTML có sử dụng thẻ applet
 - Thẻ applet có hai thuộc tính:
 - Width
 - Height
 - Để truyền tham số tới applet, sử dụng thẻ 'param', và tiếp theo là thẻ 'value'
- Applet có thể được thực thi bằng applet viewer

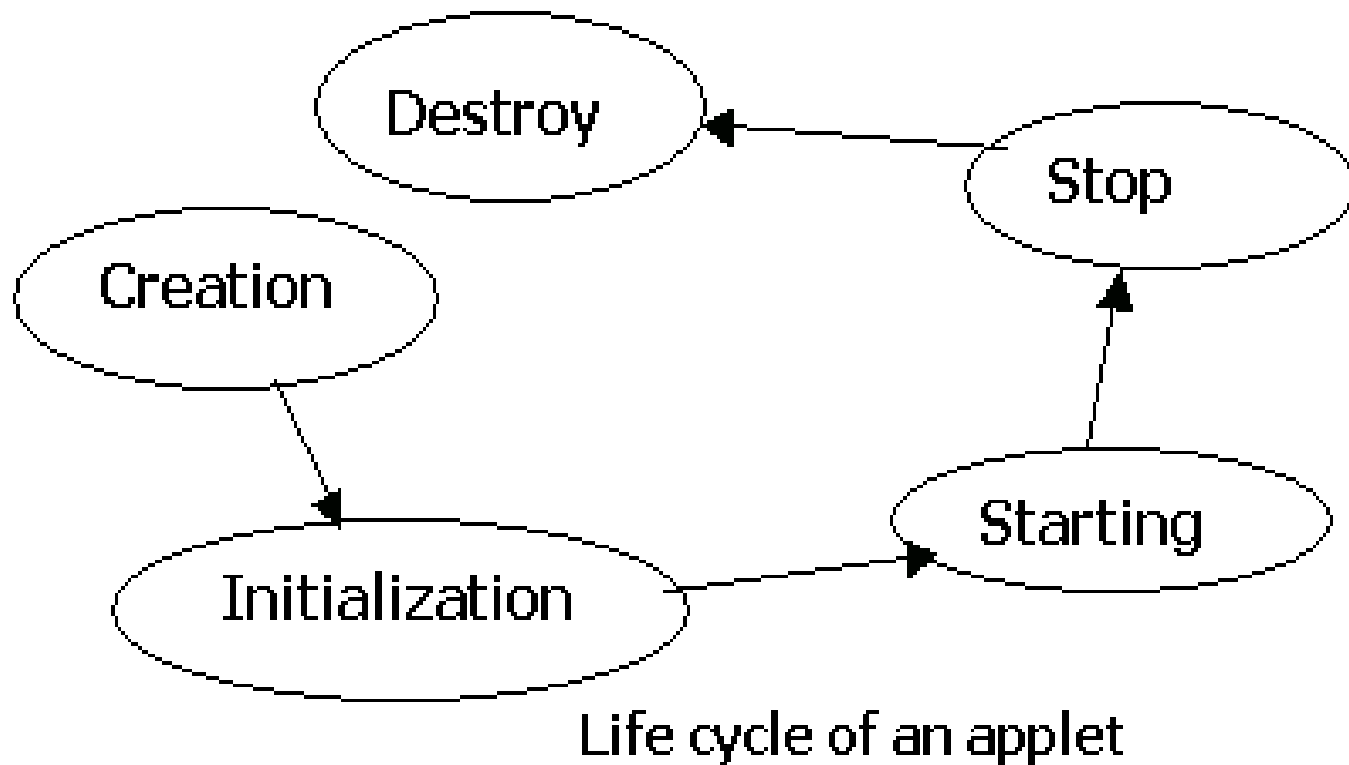
Điểm khác biệt giữa applet và một ứng dụng

- Các ứng dụng khi thực thi phải sử dụng trình biên dịch Java, trong khi các applets thực thi được trên bất kỳ trình duyệt nào mà hỗ trợ Java, hoặc sử dụng 'AppletViewer' trong JDK.
- Một ứng dụng bắt đầu với phương thức 'main()'. Còn đối với applet thì không sử dụng phương thức này
- Một ứng dụng sử dụng 'System.out.println()' để hiển thị, trong khi một applet thì sử dụng các phương thức của lớp Graphics.

Những hạn chế về bảo mật trong applet

- Không thể đọc hoặc viết các tập tin trên hệ thống tập tin của người sử dụng
- Không thể giao tiếp với một site trên internet. Mà chỉ giao tiếp với một dịch vụ trên trang web có applet.
- Không thể chạy bất kỳ chương trình nào trên hệ thống của người đọc
- Không thể load bất kỳ chương trình nào được lưu trên hệ thống của người sử dụng

Chu trình sống của applet



Truyền tham số tới một applet

- Để truyền tham số, sử dụng PARAM trong thẻ HTML
- Ví dụ

```
<applet code = "Mybutton1" width = "100" height = "100">  
<PARAM NAME = "Mybutton" value = "Display Dialog">  
</applet>
```


Lớp đồ họa

- Được cung cấp bởi gói AWT
- Cung cấp một tập hợp các phương thức để vẽ như sau:
 - **Oval**
 - **Rectangle**
 - **Square**
 - **Circle**
 - **Lines**
 - **Text in different fonts**

Graphical Background

- Các phương thức để vẽ nền :
 - **getGraphics()**
 - **repaint()**
 - **update(Graphics g)**
 - **paint(Graphics g)**

Hiển thị chuỗi, ký tự và bytes

- Phương thức để vẽ hoặc hiển thị một chuỗi trên frame

Cú pháp

– **drawString(String str, int xCoor, int yCoor);**

- Phương thức để vẽ hoặc hiển thị các ký tự trên frame

Cú pháp

– **drawChars(char array[], int offset, int length, int xCoor, int yCoor);**

- Phương thức để vẽ hoặc hiển thị bytes trên frame

Cú pháp

– **drawBytes(byte array[], int offset, int length, int xCoor, int yCoor);**

Vẽ các hình thể

- Phương thức được sử dụng để vẽ đường thẳng như sau

Cú pháp

- **drawLine(int x1, int y1, int x2, int y2);**

- Các phương thức được sử dụng để vẽ đường tròn như sau

Cú pháp

- **drawOval(int xCoor, int yCoor, int width, int height);**
- **setColor(Color c);**
- **fillOval(int xCoor, int yCoor, int width, int height);**

- Phương thức sử dụng để vẽ hình vuông:

Cú pháp

- `drawRect(int xCoor, int yCoor, int width, int height);`
- `fillRect(int xCoor, int yCoor, int width, int height);`

- Các phương thức được sử dụng để vẽ hình vuông có góc tròn

Cú pháp

- `drawRoundRect(int xCoor, int yCoor, int width, int height, int arcWidth, int arcHeight);`
- `fillRoundRect (int xCoor, int yCoor, int width, int height, int arcWidth, int arcHeight);`

3D Rectangles & Arcs

- Các phương thức được sử dụng để vẽ hình 3D **Cú pháp**
 - **draw3DRect(int xCoord, int yCoord, int width, int height, boolean raised);**
 - **drawArc(int xCoord, int yCoord, int width, int height, int arcwidth, int archeight);**
 - **fillArc(int xCoord, int yCoord, int width, int height, int arcwidth, int archeight);**

Drawing PolyLines

- Các phương thức được sử dụng để vẽ nhiều đoạn thẳng

Cú pháp

- **drawPolyline(int xArray[], int yArray[], int totalPoints);**
- **g.setFont(new Font("Times Roman", Font.BOLD,15));**

Vẽ và tô các hình đa giác

- Các phương thức để vẽ và tô các hình đa giác

Cú pháp

- **drawPolygon(int x[], int y[], int numPoints);**
- **fillPolygon(int x[], int y[], int numPoints);**

Màu

- Java sử dụng màu RGB
- Bảng các giá trị màu

Element	Range
Red	0-255
Green	0-255
Blue	0-255

- Cú pháp của hàm dựng để tạo một màu
color(int red, int green, int blue);

- Bảng trình bày các giá trị màu RGB thông thường

Color	Red	Green	Blue
White	255	255	255
Light Gray	192	192	192
Gray	128	128	128
Dark Gray	64	64	64
Black	0	0	0
Pink	255	175	175
Orange	255	200	0
Yellow	255	255	0
Magenta	255	0	255

Font

- Gói java.awt package cung cấp bởi lớp 'Font'
- Các phương thức của lớp Font:
 - **getAllFont()**
 - **getLocalGraphicsEnvironment()**
 - **getFont()**
 - **getFontList()**

- Hàm dựng Font nhận 3 tham số
 - Tên font trong chuỗi định dạng; tên này có trong phương thức `getFontList()`.
 - Kiểu của font.
Ví dụ : `Font.BOLD`, `Font.PLAIN`, `Font.ITALIC`
 - Kích thước của font.
- Ví dụ

```
Font f1 = new Font("SansSerif", Font.ITALIC, 16);  
g.setFont(f1);
```

Lớp FontMetrics

- Đo các ký tự khác nhau hiển thị trong các font khác nhau.
- Việc đo bao gồm 'height', 'baseline', 'ascent', 'descent' và 'leading' của font.
- Nó không cụ thể vì nó là một lớp trừu tượng

Lớp FontMetrics (tiếp theo...)

- Phương thức:
 - **getFontMetrics(f1)**
 - **getHeight()**
 - **getAscent()**
 - **getDescent()**
 - **getLeading()**
 - **getName()**

Kiểu vẽ

- Có 2 kiểu vẽ.

- XOR

setXORMode(Color c)

- Paint

setPaintMode()

Lập Trình Đa Tuyến

Lập trình đa tuyến

- Tiến trình, đa nhiệm và đa luồng
- Xử lý đa luồng trong Java
- Mức ưu tiên của luồng
- Vấn đề đồng bộ hoá
- Bài toán tắc nghẽn

Tiến trình, đa nhiệm và đa luồng

- **Tiến trình** : là một chương trình chạy trên hệ điều hành và được quản lý thông qua các thẻ
- **Tiểu trình** : là một đơn vị xử lý cơ bản của hệ thống. Một tiến trình sở hữu nhiều tiểu trình
- **Đơn nhiệm** : tại một thời điểm chỉ có một tiến trình
- **Đa nhiệm**: ở cùng một thời điểm có nhiều hơn một tiến trình thực hiện đồng thời trên cùng một máy tính.

Có hai kỹ thuật đa nhiệm:

- + Đa nhiệm dựa trên các *tiến trình*
- + Đa nhiệm dựa trên các *luồng*

Tiến trình, đa nhiệm và đa luồng

- Đa nhiệm có thể thực hiện được theo hai cách:
 - + Phụ thuộc vào hệ điều hành, nó có thể cho tạm ngừng chương trình mà không cần tham khảo các chương trình đó.
 - + Các chương trình chỉ bị dừng lại khi chúng tự nguyện nhường điều khiển cho chương trình khác.
- Nhiều hệ điều hành hiện nay đã hỗ trợ đa luồng, Java hỗ trợ đa nhiệm dựa trên các luồng và cung cấp các đặc tính ở mức cao cho lập trình đa luồng.

Tạo và quản lý luồng

- Khi chương trình Java thực thi hàm main() tức là luồng main được thực thi.
- Tuyến này được tạo ra một cách tự động, tại đây :
 - Các luồng con sẽ được tạo ra từ đó
 - Nó là luồng cuối cùng kết thúc việc thực thi. Ngay khi luồng main() ngừng thực thi, chương trình bị chấm dứt

Lập trình đa luồng

- Với Java ta có thể xây dựng các chương trình đa luồng
- Một ứng dụng có thể bao gồm nhiều luồng, mỗi luồng được gán công việc cụ thể và được thực thi đồng thời với các luồng khác
- Java cung cấp hai giải pháp tạo lập luồng:
 - Thiết lập lớp con của Thread
 - Cài đặt lớp xử lý luồng từ giao diện

Runnable

Lập trình đa luồng

Cách thứ nhất :

Tạo ra một lớp kế thừa từ lớp Thread và ghi đè phương thức run của lớp Thread như sau:

```
class MyClass extends Thread
{
    // Một số thuộc tính
    public void run()
    {
        // Các lệnh cần thực hiện theo luồng
    }
    // Một số hàm khác được viết đè hay được bổ sung
}
```

Khi chương trình chạy nó sẽ gọi một hàm đặc biệt đã được khai báo trong Thread đó là start() để bắt đầu một luồng đã được tạo ra.

Lập trình đa luồng

Cách thứ hai:

+ Java giải quyết hạn chế trên bằng cách xây dựng lớp để tạo ra các luồng thực hiện trên cơ sở cài đặt giao diện hỗ trợ luồng.

+ Tạo ra một lớp triển khai từ giao diện Runnable, cài đặt phương thức run

```
class MyClass implements Runnable  
{  
  
    // Các thuộc tính  
    // Nạp chồng hay viết đè một số hàm  
    public void run()  
        {  
            . . .  
        }  
  
}
```

Trạng thái và các phương thức của lớp Thread

- **Trạng thái:**

- **born**
- **ready to run**
- **running**
- **sleeping**
- **waiting**
- **ready**
- **blocked**
- **dead**

- **Phương thức:**

- **start()**
- **sleep()**
- **wait()**
- **notify()**
- **run()**
- **stop()**

Các trạng thái của Thread

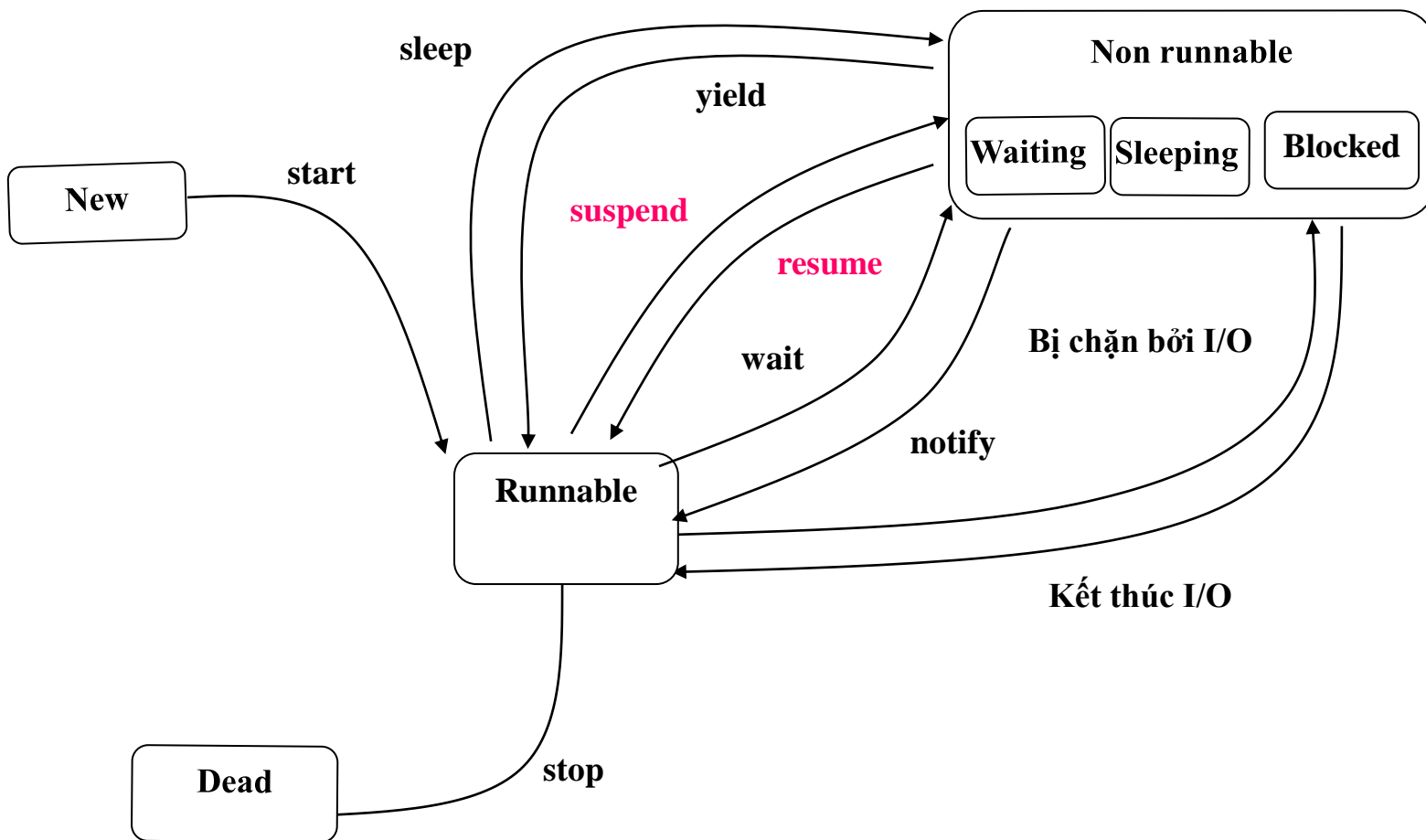
Một luồng có thể ở một trong các trạng thái sau:

- + **New**: Khi một luồng mới được tạo ra với toán tử `new()` và sẵn sàng hoạt động.

- + **Runnable**: Trạng thái mà luồng đang chiếm CPU để thực hiện, khi bắt đầu thì nó gọi hàm `start()`. Bộ lập lịch phân luồng của hệ điều hành sẽ quyết định luồng nào sẽ được chuyển về trạng thái Runnable và hoạt động. Cũng cần lưu ý rằng ở một thời điểm, một luồng ở trạng thái Runnable có thể hoặc không thể thực hiện.

- + **Non runnable (blocked)**: Từ trạng thái runnable chuyển sang trạng thái ngừng thực hiện (“bị chặn”) khi gọi một trong các hàm: `sleep()`, `suspend()`, `wait()`, *hay bị chặn lại ở Input/output*. Trong trạng thái bị chặn có ba trạng thái con:

Các trạng thái của Thread



Các trạng thái của Thread

Một luồng có thể ở một trong các trạng thái sau:

- + **Waiting**: khi ở trạng thái Runnable, một luồng thực hiện hàm wait() thì nó sẽ chuyển sang trạng thái chờ đợi (Waiting).

- + **Sleeping**: khi ở trạng thái Runnable, một luồng thực hiện hàm sleep() thì nó sẽ chuyển sang trạng thái ngủ (Sleeping).

- + **Blocked**: khi ở trạng thái Runnable, một luồng bị chặn lại bởi những yêu cầu về tài nguyên, như yêu cầu vào/ra (*I/O*), thì nó sẽ chuyển sang trạng bị chặn (Blocked).

Các trạng thái của Thread

Mỗi luồng phải thoát ra khỏi trạng thái Blocked để quay về trạng thái Runnable, khi:

- + Nếu một luồng đã được cho đi “ngủ” (sleep) sau khoảng thời gian bằng số micro giây n đã được truyền vào tham số của hàm sleep(n).

- + Nếu một luồng bị chặn lại vì vào/ra và quá trình này đã kết thúc.

- + Nếu luồng bị chặn lại khi gọi hàm wait(), sau đó được thông báo tiếp tục bằng cách gọi hàm notify() hoặc notifyAll().

- + Nếu một luồng bị chặn lại để chờ monitor của đối tượng đang bị chiếm giữ bởi luồng khác, khi monitor đó được giải phóng thì luồng bị chặn này có thể tiếp tục thực hiện (khái niệm monitor được đề cập ở phần sau).

Các trạng thái của Thread

- + Nếu một luồng bị chặn lại bởi lời gọi hàm `suspend()`, muốn thực hiện thì trước đó phải gọi hàm `resume()`.

- + Hàm `suspend()` có tác dụng tạm ngừng tuyến, ít được dùng do không nhả tài nguyên của hệ thống, dễ dẫn đến deadlock.

Nếu ta gọi các hàm không phù hợp đối với các luồng thì JVM sẽ phát sinh ra ngoại lệ `IllegalThreadStateException`.

- + **Dead**: Luồng chuyển sang trạng thái “chết” khi nó kết thúc hoạt động bình thường, hoặc gặp phải ngoại lệ không thực hiện tiếp được.

- + Trong trường hợp đặc biệt, bạn có thể gọi hàm `stop()` để kết thúc (“giết chết”) một luồng.

Mức ưu tiên của các luồng

- + Trong Java, mỗi luồng có một mức ưu tiên thực hiện nhất định.
- + Khi chương trình chính thực hiện sẽ tạo ra luồng chính, luồng cha. Luồng này sẽ tạo ra các luồng con, và cứ thế tiếp tục.
- + Theo mặc định, một luồng sẽ kế thừa mức ưu tiên của luồng cha của nó. Bạn có thể tăng hay giảm mức ưu tiên của luồng bằng cách sử dụng hàm **setPriority()**.
- + Mức ưu tiên của các luồng có thể đặt lại trong khoảng từ MIN_PRIORITY (Trong lớp Thread được mặc định bằng 1) và MAX_PRIORITY (mặc định bằng 10), hoặc NORM_PRIORITY (mặc định là 5).
- + Luồng có mức ưu tiên cao nhất tiếp tục thực hiện cho đến khi:
 - Nó nhường quyền điều khiển cho luồng khác bằng cách gọi hàm yield()
 - Nó dừng thực hiện (bị “dead” hoặc chuyển sang trạng thái bị chặn).

Mức ưu tiên của các luồng

+ Vấn đề nảy sinh là chọn luồng nào để thực hiện khi có nhiều hơn một luồng sẵn sàng thực hiện và có cùng một mức ưu tiên cao nhất? Nói chung, một số cơ sở sử dụng bộ lập lịch lựa chọn ngẫu nhiên, hoặc lựa chọn chúng để thực hiện theo thứ tự xuất hiện.

Ví dụ:

Chúng ta hãy xét chương trình hiển thị các quả bóng màu xanh hoặc đỏ nảy (chuyển) theo những đường nhất định.

Mỗi khi nhấn nút “Blue ball” thì có 5 luồng được tạo ra với mức ưu tiên thông thường (mức 5) để hiển thị và di chuyển các quả bóng xanh.

Khi nhấn nút “Red ball” thì cũng có 5 luồng được tạo ra với mức ưu tiên (mức 7) cao hơn mức thông thường để hiển thị và di chuyển các quả bóng đỏ.

Để kết thúc trò chơi bạn nhấn nút “Close”.

Mức ưu tiên của các luồng

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Bounce{
    public static void main(String arg[]){
        JFrame fr = new BounceFrame();
        fr.show();
    }
}
class BounceFrame extends JFrame{
    public BounceFrame(){
        setSize(300, 200);
        setTitle("Bong chuyen");
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
}
```


Mức ưu tiên của các luồng

```
Container contentPane = getContentPane();
canvas = new JPanel();
contentPane.add(canvas, "Center");
JPanel p = new JPanel();
addButton(p, "Blue ball", new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        for(int i = 0; i < 5; i++){
            Ball b = new Ball(canvas, Color.blue);
            b.setPriority(Thread.NORM_PRIORITY);
            b.start(); }
    }
});
addButton(p, "Red ball", new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        for(int i = 0; i < 5; i++){
            Ball b = new Ball(canvas, Color.red);
            b.setPriority(Thread.NORM_PRIORITY + 2);
            b.start();
        }
    }
});
```

Mức ưu tiên của các luồng

```
addButton(p, "Close", new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        canvas.setVisible(false);
        System.exit(0);
    }
});
contentPane.add(p, "South");
}
public void addButton(Container c, String title, ActionListener a){
    JButton b = new JButton(title);
    c.add(b);
    b.addActionListener(a);
}
private JPanel canvas;
}
```

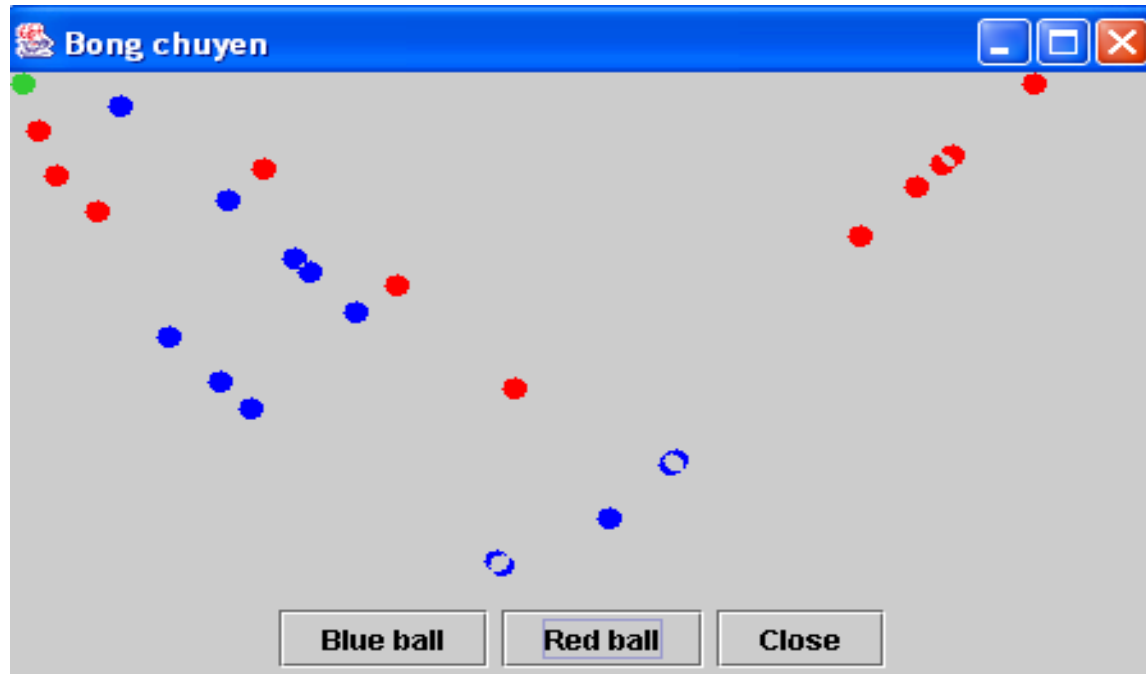
Mức ưu tiên của các luồng

```
class Ball extends Thread{
    public Ball(JPanel b, Color c){
        box = b; color = c;
    }
    public void draw(){
        Graphics g = box.getGraphics(); g.setColor(color);
        g.fillOval(x, y, XSIZE, YSIZE); g.dispose();
    }
    public void move(){
        if(!box.isVisible()) return;
        Graphics g = box.getGraphics();
        g.setXORMode(box.getBackground()); g.setColor(color);
        g.fillOval(x, y, XSIZE, YSIZE);
        x += dx;  y += dy;
        Dimension d = box.getSize();
        if(x < 0){
            x = 0; dx = -dx;
        }
        if(x + XSIZE >= d.width){
            x = d.width - XSIZE; dx = -dx;
        }
        if(y < 0){
            y = 0; dy = -dy; }
    }
}
```

Mức ưu tiên của các luồng

```
if(y + YSIZE >= d.height){
    y = d.height - YSIZE; dy = -dy;
}
g.fillOval(x, y, XSIZE, YSIZE);
g.dispose();
}
public void run(){
    try{
        for(int i = 1; i <= 1000; i++){
            move();    sleep(5);
        }
    }catch(InterruptedException e){
    }
}
private JPanel box;
private static final int XSIZE = 10;
private static final int YSIZE = 10;
private int x = 0;
private int y = 0;
private int dx = 2;
private int dy = 2;
private Color color;
}
```

Mức ưu tiên của các luồng



Chạy chương trình trên chúng ta nhận thấy hình như những quả bóng đỏ nảy nhanh hơn vì các luồng thực hiện chúng có mức ưu tiên cao hơn.

Mức ưu tiên của các luồng

- + Các luồng có mức ưu tiên thấp hơn sẽ không có cơ hội thực hiện nếu những luồng cao hơn không nhường, hoặc nhường bằng hàm `yield()`.
- + Nếu có những luồng đang ở trạng thái `Runnable` mà có mức ưu tiên ít nhất bằng mức ưu tiên của luồng vừa nhường thì một trong số chúng được xếp lịch để thực hiện.
- + Bộ lập lịch thường xuyên tính lại mức ưu tiên của các luồng đang thực hiện
- + Tìm luồng có mức ưu tiên cao nhất để thực hiện.

Đa tuyến với Applets

- Trong Java ta có thể tạo ra các tuyến thi hành song song bằng cách triển khai giao diện Runnable
- Java không hỗ trợ kế thừa bội
- Muốn kế thừa từ một lớp nào đó mà lại muốn đa tuyến thì bắt buộc sử dụng giao diện Runnable
- Các chương trình Java dựa trên Applet thường sử dụng nhiều hơn một tuyến

Đa tuyến với Applets

- Trong đa tuyến với Applets, Lớp 'java.applet.Applet' là lớp con được tạo ra một Applet người sử dụng đã định nghĩa
- Lớp con của Applet không thể dẫn xuất được trực tiếp từ lớp Thread.
- Cách để lớp con Applet là tuyến:
 - implements Runnable
 - Truyền đối tượng Runnable vào hàm constructor của Thread.

Sự đồng bộ

- Khi nhiều tuyến truy cập tài nguyên dùng chung
- Tài nguyên không thể chia sẻ, khi đó tài nguyên có thể bị phá hỏng

Ví dụ : Một luồng đọc dữ liệu, trong khi luồng khác lại thay đổi

- Cần cho phép một luồng hoàn thành tác vụ của nó, rồi cho phép luồng kế tiếp thực thi

Sự đồng bộ

- Thâm nhập các tài nguyên/dữ liệu bởi nhiều tuyến
- Sự đồng bộ (Synchronization)
- Sự quan sát (Monitor)

Sự đồng bộ

- Để thâm nhập sự quan sát của một đối tượng, lập trình viên sử dụng từ khóa '**synchronized**' khi khai báo phương thức.
- Mỗi một đối tượng sẽ có một bộ quản lý khóa, chỉ cho một phương thức "**synchronized**" của đối tượng đó chạy tại một thời điểm
- Khi một tuyến đang được thực thi trong phạm vi một phương thức đồng bộ (**synchronized**), bất kỳ tuyến khác hoặc phương thức đồng bộ khác mà cố gắng gọi nó trong thời gian đó sẽ phải đợi

Sự đồng bộ

- Các luồng chia sẻ với nhau cùng một không gian bộ nhớ, nghĩa là chúng có thể chia sẻ với nhau các tài nguyên.
- Khi có nhiều hơn một luồng cùng muốn sử dụng một tài nguyên sẽ xuất hiện tình trạng căng thẳng, ở đó chỉ cho phép một luồng được quyền truy cập.
- Để cho các luồng chia sẻ với nhau được các tài nguyên và hoạt động hiệu quả, luôn đảm bảo nhất quán dữ liệu thì phải có cơ chế đồng bộ chúng.

Sự đồng bộ

- Mấu chốt của sự đồng bộ là khái niệm “monitor” (giám sát) hay còn gọi là “semaphore” (cờ hiệu)
- Khái niệm “semaphore” thường được sử dụng để điều khiển đồng bộ các hoạt động truy cập vào những tài nguyên dùng chung.
- Một luồng muốn truy cập vào một tài nguyên dùng chung (như biến dữ liệu) thì trước tiên nó phải yêu cầu để có được monitor riêng.
- Khi có được monitor thì luồng như có được “chìa khoá” để “mở cửa” vào miền “tranh chấp” để sử dụng những tài nguyên đó.

Sự đồng bộ

- Cơ chế monitor thực hiện hai nguyên tắc đồng bộ chính:
 - + Không một luồng nào khác được phân monitor khi có một luồng đã yêu cầu và đang chiếm giữ. Những luồng có yêu cầu monitor sẽ phải chờ cho đến khi monitor được giải phóng.
 - + Khi có một luồng giải phóng (ra khỏi) monitor, một luồng đang chờ *monitor* có thể truy cập vào tài nguyên dùng chung tương ứng với monitor đó.
- Mọi đối tượng trong Java đều có monitor, mỗi đối tượng có thể được sử dụng như một khoá loại trừ nhau, cung cấp khả năng để đồng bộ truy cập vào những tài nguyên chia sẻ.
- Trong lập trình có hai cách để thực hiện đồng bộ:
 - + **Các hàm được đồng bộ**
 - + **Các khối được đồng bộ**

Sự đồng bộ

- Hàm của một lớp chỉ cho phép một luồng được thực hiện ở một thời điểm thì nó phải khai báo *synchronized*, được gọi là *hàm đồng bộ*.
- Một luồng muốn thực hiện hàm đồng bộ thì nó phải chờ để có được monitor của đối tượng có hàm đó.
- Trong khi một luồng đang thực hiện hàm đồng bộ thì tất cả các luồng khác muốn thực hiện hàm này của cùng một đối tượng, đều phải chờ cho đến khi luồng đó thực hiện xong và được giải phóng.
- Bằng cách đó, những hàm được đồng bộ sẽ không bao giờ bị tắc nghẽn.

Sự đồng bộ

- Những hàm không được đồng bộ của đối tượng có thể được gọi thực hiện mọi lúc bởi bất kỳ đối tượng nào.
- Khi chạy, chương trình sẽ thi hành tuần tự các lệnh cho đến khi kết thúc chương trình.
- Trong Java, hàm đồng bộ có thể khai báo **static**. Các lớp cũng có thể có các monitor tương tự như đối với các đối tượng.
- Một luồng yêu cầu monitor của lớp trước khi nó có thể thực hiện với một hàm được đồng bộ tĩnh (static) nào đó trong lớp, đồng thời các luồng khác muốn thực hiện những hàm như thế của cùng một lớp thì **bị chặn lại**.

Sự đồng bộ

Ví dụ: Hệ thống ngân hàng có 10 tài khoản, trong đó có các giao dịch chuyển tiền giữa các tài khoản với nhau một cách ngẫu nhiên. Chương trình tạo ra 10 luồng cho 10 tài khoản. Mỗi giao dịch được một luồng phục vụ sẽ chuyển một lượng tiền ngẫu nhiên từ một tài khoản sang tài khoản khác.

- Nếu chương trình thực hiện với 10 luồng hoạt động không đồng bộ để chuyển tiền giữa các tài khoản trong ngân hàng.
- Vấn đề sẽ nảy sinh khi có hai luồng đồng thời muốn chuyển tiền vào cùng một tài khoản. Giả sử hai luồng cùng thực hiện:

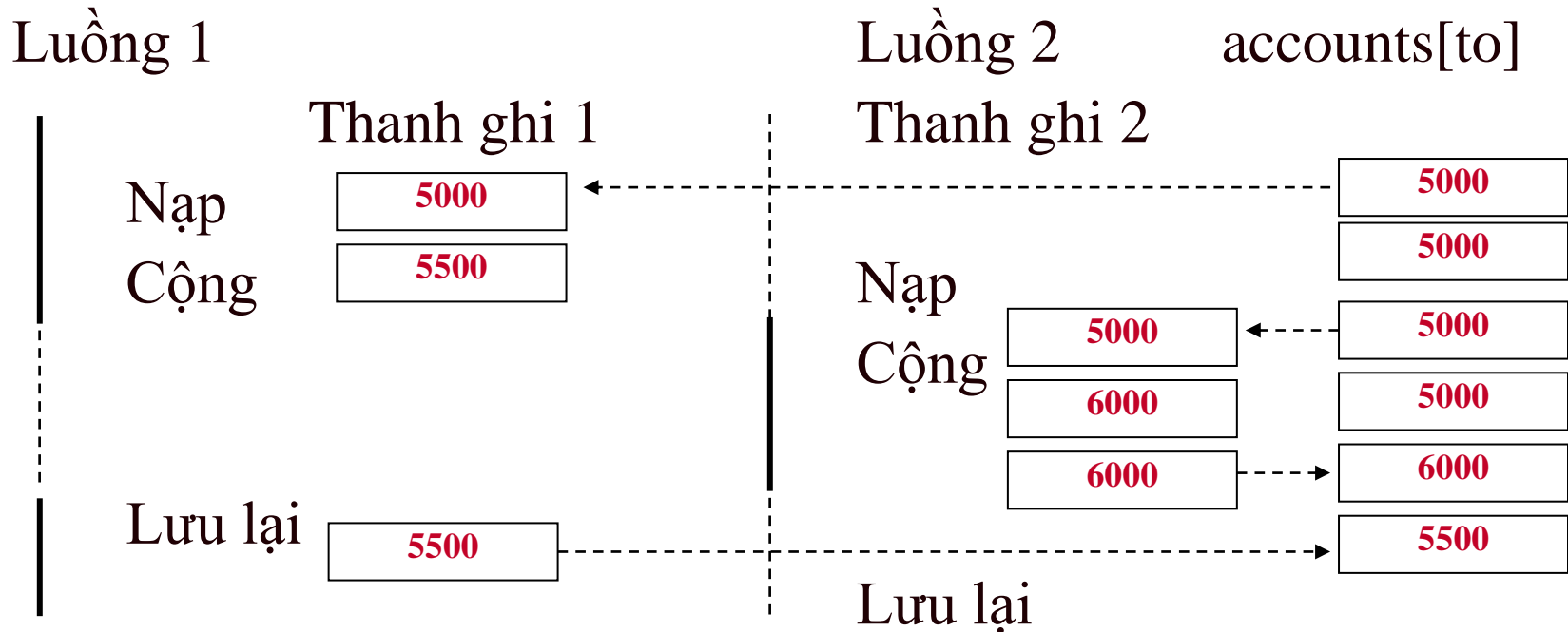
accounts[to] += amount;

Câu lệnh này được thực hiện như sau:

1. Nạp accounts[to] vào thanh ghi.
2. Cộng số tiền trong tài khoản accounts với amount
3. Lưu lại kết quả cho accounts[to].

Sự đồng bộ

- Chúng ta có thể giả thiết luồng thứ nhất thực hiện bước 1 và 2 với amount = 500, sau đó nó bị ngắt. Luồng thứ hai có thể thực hiện trọn vẹn cả ba bước trên với amount = 1000, sau đó luồng thứ nhất kết thúc việc cập nhật bằng cách thực hiện nốt bước 3. Quá trình này được mô tả như ở hình sau :



- Kết thúc luồng thứ nhất, `accounts[to]` có 6000, nhưng ngay sau đó luồng thứ hai kết thúc thì cũng chính tài khoản đó chưa chuyển tiền đi đâu cả, nhưng lại chỉ còn 5500. Đúng ra nó phải là 6500

Sự đồng bộ

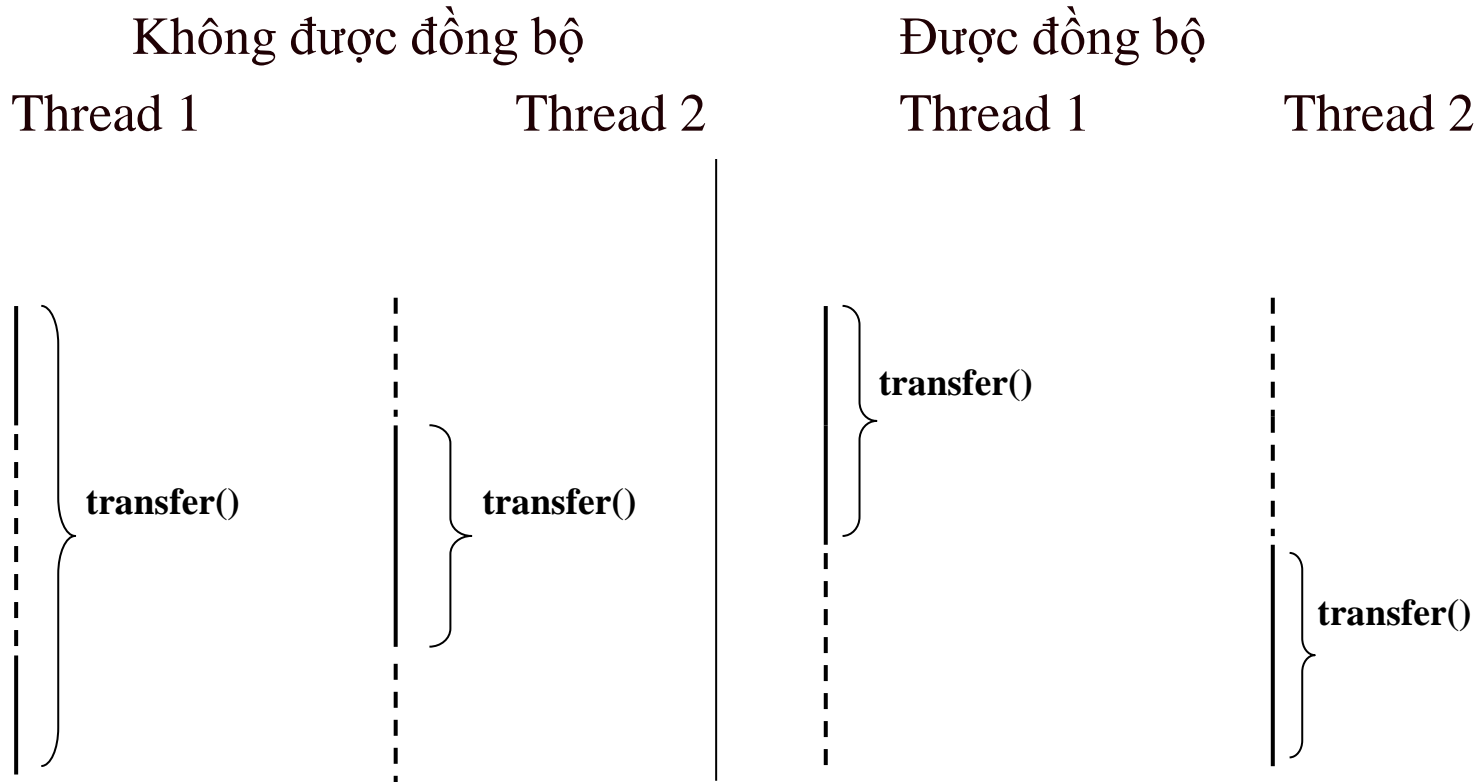
- Java sử dụng cơ chế đồng bộ khá hiệu quả là `monitor`. Một hàm sẽ không bị ngắt nếu bạn khai báo nó là `synchronized`.

```
public synchronized void transfer(int from, int to,
int amount){
    if(accounts[from] < amount) return;
    accounts[from] -= amount;
    accounts[to] += amount;
    numTransacts++;
    if(numTransacts % NTEST == 0) test();
}

public synchronized void test(){
    int sum = 0;
    for(int i = 0; i < accounts.length; i++)
        sum += accounts[i];
    System.out.println("Giao dịch: " + numTransacts
        + " tong so: " + sum);
}
```

Sự đồng bộ

- Khi có một luồng gọi hàm được đồng bộ thì nó được đảm bảo rằng hàm này phải thực hiện xong thì luồng khác mới được sử dụng đối với cùng một đối tượng.
- Hoạt động của các luồng không đồng bộ và đồng bộ của hai luồng thực hiện gọi hàm `transfer()`



Các khoá đối tượng

- Khi một luồng gọi một hàm được đồng bộ thì đối tượng của nó bị “khóa”, giống như khóa cửa phòng.
- Như vậy, khi một luồng khác muốn gọi hàm được đồng bộ của cùng đối tượng đó thì sẽ không mở được.
- Sau khi thực hiện xong, luồng ở bên trong giải phóng hàm được đồng bộ vừa sử dụng, ra khỏi đối tượng và đưa chìa khóa ra ngoài bậc cửa để những luồng khác có thể tiếp tục công việc của mình.

Các khoá đối tượng

- Một luồng có thể giữ nhiều khoá đối tượng ở cùng một thời điểm, như trong khi đang thực hiện một lời gọi hàm đồng bộ của một đối tượng, nó lại gọi tiếp hàm đồng bộ của đối tượng khác.
- Nhưng, tại mỗi thời điểm, mỗi khoá đối tượng chỉ được một luồng sở hữu.
- Chúng ta hãy phân tích chi tiết hơn hoạt động của hệ thống ngân hàng. Một giao dịch chuyển tiền sẽ không thực hiện được nếu không còn đủ tiền. Phải chờ cho đến các tài khoản khác chuyển tiền đến và khi có đủ thì mới thực hiện được giao dịch đó.

```
public synchronized void transfer(int from, int to, int amount){  
    while(accounts[from] < amount)  
        wait();  
    // Chuyển tiền  
}
```

Các khoá đối tượng

- Chúng ta sẽ làm gì khi trong tài khoản không có đủ tiền? Tất nhiên là phải chờ cho đến khi có đủ tiền trong tài khoản. Nhưng `transfer()` là hàm được đồng bộ.
- Do đó, khi một luồng đã chiếm được khoá đối tượng thì luồng khác sẽ không có cơ hội sở hữu trừ nó, cho đến khi luồng trước giải phóng khoá đó.
- Khi `wait()` được gọi ở trong hàm được đồng bộ (như ở `transfer()`), luồng hiện thời sẽ bị chặn lại và trao lại khoá đối tượng cho luồng khác.
- Có sự khác nhau thực sự giữa luồng đang chờ để sử dụng hàm đồng bộ với hàm bị chặn lại bởi hàm `wait()`.

Các khoá đối tượng

- Khi một luồng gọi `wait()` thì nó được đưa vào danh sách hàng đợi.
 - + Cho đến khi các luồng chưa được đưa ra khỏi danh sách hàng đợi thì bộ lập lịch sẽ bỏ qua và do vậy chúng không thể tiếp tục được.
 - + Để đưa một luồng ra khỏi danh sách hàng đợi thì phải có một luồng khác gọi `notify()` hoặc `notifyAll()` trên cùng một đối tượng.
 - + *`notify()`* đưa một luồng bất kỳ ra khỏi danh sách hàng đợi.
 - + *`notifyAll()`* đưa tất cả các luồng ra khỏi danh sách hàng đợi.

Các khoá đối tượng

- Những luồng đưa ra khỏi danh sách hàng đợi sẽ được bộ lập lịch kích hoạt chúng. Ngay tức khắc luồng nào chiếm được khoá đối tượng thì sẽ bắt đầu thực hiện.
- Như vậy, trong hàm transfer() chúng ta gọi notifyAll() khi kết thúc việc chuyển tiền để một trong các luồng có thể được tiếp tục thực hiện và tránh bế tắc.
- Cuối cùng chương trình sử dụng cơ chế đồng bộ được viết lại như sau:

```
public class SynBankTransfer{
    public static void main(String arg[]){
        Bank b = new Bank(NACCOUNTS, INI_BALANCE);
        for(int i = 0; i < NACCOUNTS; i++){
            TransferThread t = new TransferThread(b, i, INI_BALANCE);
            t.setPriority(Thread.NORM_PRIORITY + i % 2);
            t.start();
        }
    }
    public static final int NACCOUNTS = 10;
    public static final int INI_BALANCE = 10000;
}
```

Các khoá đối tượng

```
class Bank{
    public static final int NTEST = 1000;
    private int[] accounts;
    private long numTransacts = 0;
    public Bank(int n, int initBalance){
        accounts = new int[n];
        for(int i = 0; i < accounts.length; i++)
            accounts[i] = initBalance;
        numTransacts = 0;
    }
    public void transfer(int from, int to, int amount){
        while(accounts[from] < amount) wait();
        accounts[from] -= amount;
        accounts[to] += amount;
        numTransacts++;
        notifyAll();
        if(numTransacts % NTEST == 0) test();
    }
}
```

Các khoá đối tượng

```
public synchronized void test(){
    int sum = 0;
    for(int i = 0; i < accounts.length; i++)
        sum += accounts[i];
    System.out.println("Giao dich: " + numTransacts + " tong so: " + sum);
}

public int size(){
    return accounts.length;
}
}

class TransferThread extends Thread{
    private Bank bank;
    private int fromAcc;
    private int maxAmount;
    public TransferThread(Bank b, int from, int max){
        bank = b;
        fromAcc = from;    maxAmount = max;
    }
}
```

Các khoá đối tượng

```
public void run(){  
    try{  
        while(!interrupted()){  
            int toAcc = (int)(bank.size() * Math.random());  
            int amount = (int)(maxAmount * Math.random());  
            bank.transfer(fromAcc, toAcc, amount);  
            sleep(1);  
        }  
    }catch(InterruptedException e){  
    }  
}
```

Nếu bạn chạy chương trình với các hàm `transfer()`, `test()` được đồng bộ thì mọi việc sẽ thực hiện chính xác đúng theo yêu cầu. Tuy nhiên, bạn cũng có thể nhận thấy chương trình sẽ chạy chậm hơn chút ít bởi vì phải trả giá cho cơ chế đồng bộ nhằm đảm bảo cho hệ thống hoạt động chính xác, đảm bảo nhất quán dữ liệu, hoặc tránh gây ra tắc nghẽn.

Deadlock

- Một “deadlock” xảy ra khi hai tuyến có một phụ thuộc vòng quanh trên một cặp đối tượng đồng bộ
- Cơ chế đồng bộ trong Java là rất tiện lợi, khá mạnh, nhưng không giải quyết được mọi vấn đề nảy sinh trong quá trình xử lý đa luồng.

Deadlock

Ví dụ : ở Account 1 có 2000\$, Account 2 có 3000\$ và Thread 1 cần chuyển 3000\$ từ Account 1 sang Account 2, ngược lại Thread 2 cần chuyển 3500\$ từ Account 2 sang Account 1.

- Khi đó, Thread 1 và Thread 2 rơi vào tình trạng *chết tắc* hoặc *tắc nghẽn* vì chúng chặn lẫn nhau.
- Một hệ thống mà tất cả các luồng (tiến trình) bị chặn lại để chờ lẫn nhau và không một luồng (tiến trình) nào thực hiện tiếp thì được gọi là hệ thống bị chết tắc (tắc nghẽn).
- Trong tình huống ở trên, cả hai luồng đều phải gọi wait() xử lý hai tài khoản đều không đủ số tiền để chuyển.

Deadlock

- Trong chương trình `SynBankTransfer.java`, hiện tượng tắc nghẽn không xuất hiện bởi một lý do đơn giản. Mỗi giao dịch chuyển tiền nhiều nhất là 10000\$.
- Có 10 tài khoản với tổng số tiền là 100000\$. Do đó, ở mọi thời điểm đều có ít nhất một tài khoản có không ít hơn 10000\$, nghĩa là luồng phụ trách tài khoản đó được phép thực hiện.
- Tuy nhiên, khi lập trình ta có thể gây ra tình huống khác có thể làm xuất hiện tắc nghẽn

Deadlock

- Trong `SynBankTransfer.java` thay vì gọi `notifyAll()` ta gọi `notify()`.
- Như ở trên đã phân tích, `notifyAll()` thông báo cho tất cả các luồng đang chờ để có đủ tiền chuyển đi có thể tiếp tục thực hiện, còn `notify()` chỉ báo cho một luồng được tiếp tục.
- Khi đó, nếu luồng được thông báo lại không thể thực hiện, vì không đủ tiền để chuyển chẳng hạn, thì tất cả các luồng khác cũng sẽ bị chặn lại.

Deadlock

- Chúng ta hãy xét ví dụ sau:
 - + Account 1: 19000\$
 - + Tất cả các Account còn lại đều có 9000\$
 - + Thread 1: chuyển 9500\$ từ Account 1 sang Account 2
 - + Tất cả các luồng khác đều chuyển sang tài khoản khác một lượng tiền là 9100\$.
- Chỉ có Thread 1 đủ tiền để chuyển còn các luồng khác bị chặn lại. Thread 1 thực hiện chuyển tiền xong ta có:
 - + Account 1: 9500\$
 - + Account 2: 18500\$
 - + Tất cả các Account còn lại đều có 9000\$

Deadlock

- Giả sử Thread 1 gọi `notify()`. Hàm này chỉ thông báo cho một luồng ngẫu nhiên để nó có thể tiếp tục thực hiện. Giả sử đó là Thread 3. Nhưng luồng này cũng không chuyển được vì không đủ tiền ở tài khoản Account 3, nên phải chờ (gọi `wait()`).
- Thread 1 vẫn tiếp tục thực hiện. Một giao dịch mới ngẫu nhiên lại được tạo ra.
- Chẳng hạn Thread 1 chuyển 9600\$ từ Account 1 sang Account 2. Bây giờ Thread lại gọi `wait()`, và như vậy tất cả các luồng đều rơi vào tình trạng tắc nghẽn.
- Qua ví dụ trên cho thấy, một *ngôn ngữ lập trình có cơ chế hỗ trợ đồng bộ là chưa đủ để giải quyết vấn đề tắc nghẽn*. Quan trọng là khi thiết kế chương trình, ta phải đảm bảo rằng ở mọi thời điểm có ít nhất một luồng (tiến trình) tiếp tục thực hiện.

Phương thức finalize()

- Java cung cấp một cách để làm sạch một tiến trình trước khi điều khiển trở lại hệ điều hành
- Phương thức finalize(), nếu hiện diện sẽ được thực thi trên mỗi đối tượng, trước khi sự dọn rác
- Câu lệnh của phương thức finalize() như sau :
 - **protected void finalize() throws Throwable**
- Tham chiếu không phải là sự dọn rác; chỉ các đối tượng mới được dọn rác

Lập Trình

Các Luồng Vào Ra

Các luồng

- Các luồng là những “đường ống” để gửi và nhận thông tin trong các chương trình java.
- Khi một luồng đọc hoặc ghi, các luồng khác bị khoá.
- Nếu lỗi xảy ra trong khi đọc hoặc ghi luồng, một biệt lệ sẽ được tạo ra.
- Lớp ‘java.lang.System’ định nghĩa luồng nhập và xuất chuẩn.

Các lớp luồng I/O

- Lớp System.out.
- Lớp System.in.
- Lớp System.err.

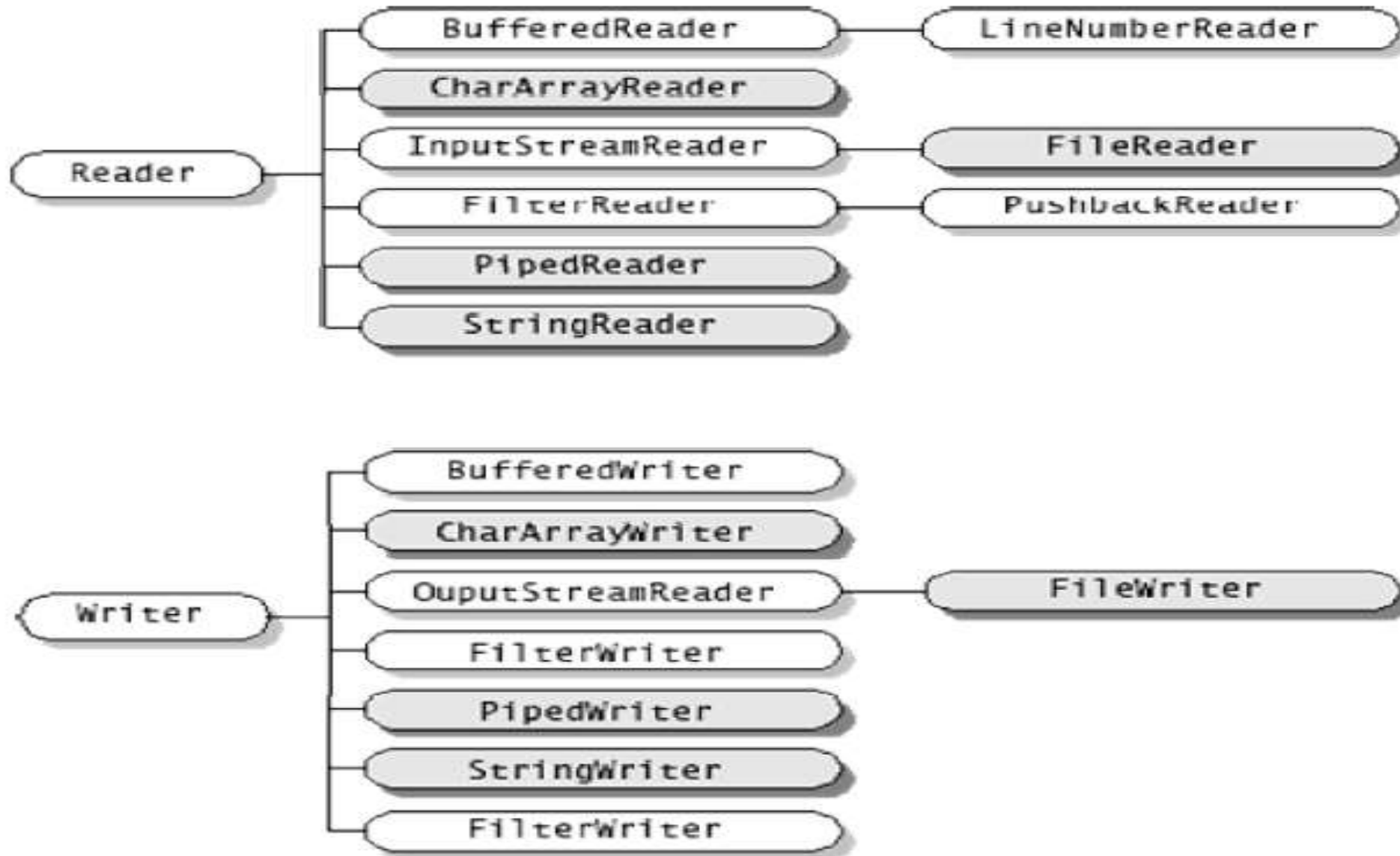
Các lớp luồng I/O

- Để xử lý mọi loại dữ liệu, java chia luồng thành 2 loại : **luồng byte (byte stream)** và **luồng ký tự (character stream)**
- Lớp **InputStream** và **OutputStream** là hai lớp cơ sở cho mọi luồng nhập xuất **hướng byte**
- Lớp **Reader/ Writer** hai lớp cơ sở cho việc đọc ghi **hướng ký tự**

Lớp nhập xuất hướng ký tự

- Reader và Writer là hai lớp cơ sở trừu tượng cho luồng hướng ký tự
- Cung cấp một giao diện chung cho tất cả các lớp đọc ghi hướng ký tự
- Mỗi lần đọc/ghi ra luồng thì đọc 2 byte tương ứng với một ký tự

Mô hình phân cấp đọc/ghi hướng ký tự



Lớp Reader

- Lớp Reader và InputStream có một giao diện giống nhau, chúng chỉ khác nhau về kiểu dữ liệu đầu vào
- Lớp Reader có các phương thức đọc một ký tự hoặc mảng các ký tự
- Các phương thức:
 - **int read()**
 - **int read(char cbuf[])**
 - **int read(char cbuf[], int offset, int length)**

Lớp Writer

- Lớp Writer và OutputStream có một giao diện giống nhau, chúng chỉ khác nhau về kiểu dữ liệu đầu vào
- Lớp Writer định nghĩa các phương thức để ghi một ký tự hoặc mảng các ký tự ra luồng
- Các phương thức:
 - `int write(int c)`
 - `int write(char cbuf[])`
 - `int write(char cbuf[], int offset, int length)`

Lớp Writer

- Hỗ trợ các phương thức sau :
 - **flush()**
 - **close()**

Nhập/xuất chuỗi và mảng ký tự

- Hỗ trợ nhập và xuất từ các vùng đệm bộ nhớ
- Lớp 'CharArrayReader' không bổ sung phương thức mới vào các phương thức mà lớp 'Reader' cung cấp.

Nhập/xuất chuỗi và mảng ký tự

- Lớp 'CharArrayWriter' bổ sung phương thức sau đây vào phương thức của lớp 'Writer' cung cấp:
 - **reset()**
 - **size()**
 - **toCharArray()**
 - **toString()**
 - **writeTo()**

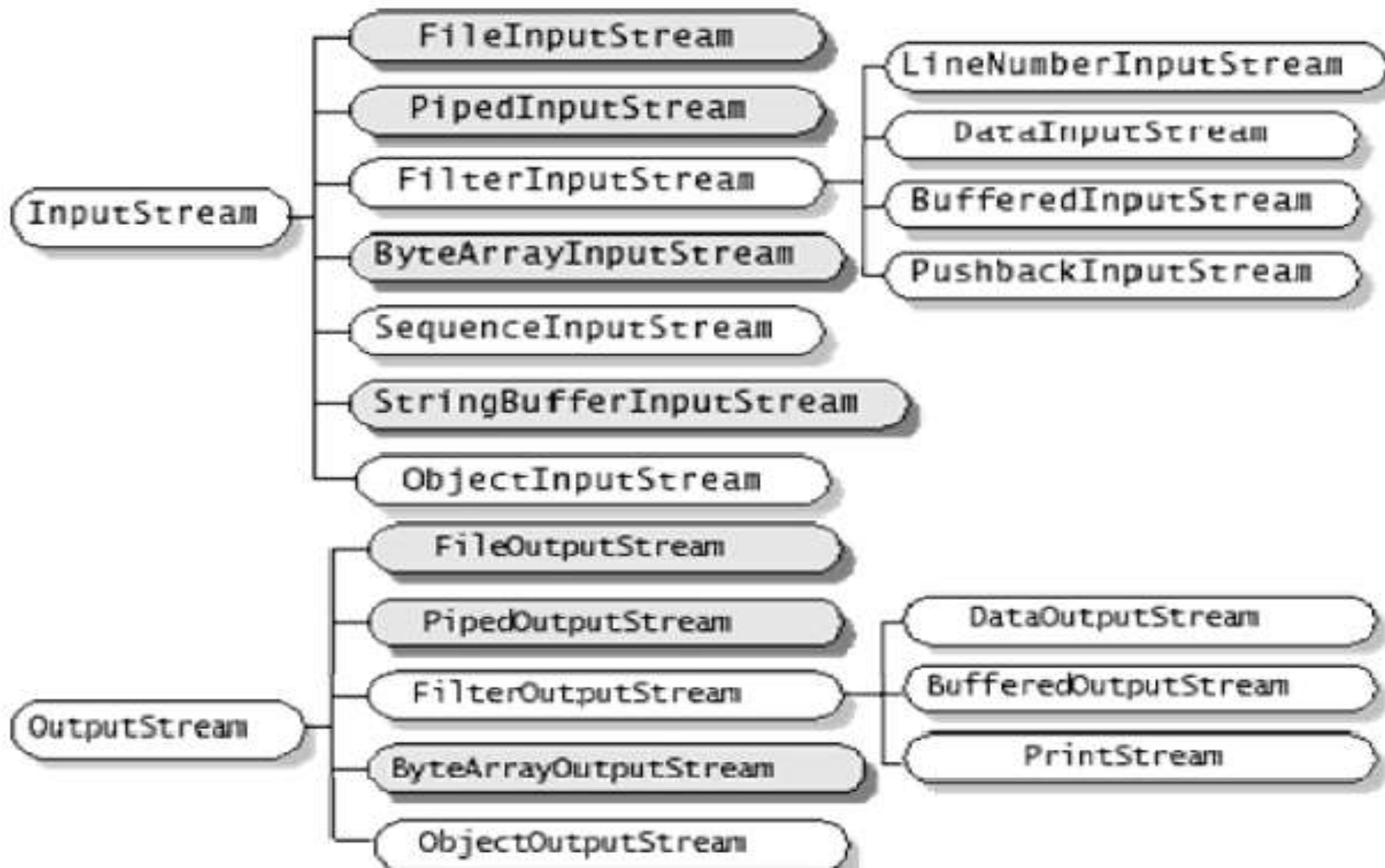
Nhập/xuất chuỗi và mảng ký tự

- Lớp 'StringReader' trợ giúp đọc các ký tự đầu vào từ xâu chuỗi.
- Nó không bổ sung bất kỳ phương thức nào mà lớp Reader cung cấp.
- Lớp 'StringWriter' trợ giúp để ghi luồng kết xuất ký tự ra một đối tượng 'StringBuffer'.
- Lớp này bổ sung thêm các phương thức sau:
 - **getBuffer()**
 - **toString()**

Lớp hướng byte

- Để có thể đọc ghi 1 byte, ta sử dụng luồng hướng byte
- Hai lớp **InputStream** và **OutputStream** là hai lớp cơ sở trừu tượng cho các luồng hướng byte
- Mỗi lần đọc/ghi ra luồng thì đọc **8 bits ra luồng**

Mô hình phân cấp đọc/ghi hướng byte



Lớp InputStream

- Là lớp trừu tượng
- Định nghĩa cách nhận dữ liệu
- Cung cấp số phương thức dùng để đọc các luồng dữ liệu làm đầu vào.
- Trong lớp InputStream có các phương thức cho việc đọc một byte hoặc mảng các byte
- Các phương thức:
 - **int read()**
 - **int read(byte cbuf[])**
 - **int read(byte cbuf[], int offset, int length)**

Lớp OutputStream

- Là lớp trừu tượng.
- Định nghĩa cách ghi dữ liệu vào luồng.
- Cung cấp tập các phương thức trợ giúp trong việc tạo, ghi và xử lý các luồng xuất.
- Lớp OutputStream có các phương thức để ghi một byte hoặc mảng các byte ra luồng
- Các phương thức:
 - **write(int c)**
 - **write(byte cbuf[])**
 - **write(byte[], int offset, int length)**

Nhập mảng các Byte

- Sử dụng các đệm bộ nhớ
- Lớp **ByteArrayInputStream**
- Tạo ra một luồng nhập từ đệm bộ nhớ về mảng các byte.
 - Không hỗ trợ các phương thức mới
 - Các phương thức nạp chồng của lớp `InputStream`, giống như `'read()'`, `'skip()'`, `'available()'` và `'reset()'`.

Xuất mảng các Byte

- sử dụng các vùng đệm bộ nhớ
- Lớp **ByteArrayOutputStream**
 - Tạo ra một luồng kết xuất trên mảng byte
 - Cung cấp các khả năng bổ sung cho mảng kết xuất tăng trưởng nhằm chứa chỗ cho dữ liệu mới ghi vào.
 - Cũng cung cấp các phương thức để chuyển đổi luồng tới mảng byte, hay đối tượng String.

- Phương thức của lớp **ByteArrayOutputStream** :
 - **reset()**
 - **size()**
 - **writeTo()**

Các lớp nhập/xuất File

- Các lớp này trợ giúp trong Java để hỗ trợ các thao tác nhập và xuất:
 - File
 - FileDescriptor
 - FileInputStream
 - FileOutputStream
 - FileReader
 - FileWriter
- Các lớp File, FileDescriptor, và RandomAccessFile được sử dụng hỗ trợ trực tiếp hoặc truy cập nhập/xuất ngẫu nhiên.

Lớp tập tin

- Được sử dụng truy cập các đối tượng tập tin và thư mục
- Những tập tin có tên được đặt tên theo quy ước của hệ điều hành.
- Lớp này cung cấp phương thức khởi tạo để tạo ra các thư mục và tập tin
- Tất cả các thao tác thư mục và tập tin đều được sử dụng các phương thức truy cập và các phương thức thư mục mà các lớp tập tin cung cấp

Lớp tệp tin

- Để xử lý tệp tin ngoại trú, ta sử dụng các luồng liên quan đến tệp tin như :
FileInputStream, FileOutputStream,
FileReader, FileWriter
- FileInputStream và FileOutputStream phục vụ cho việc đọc ghi tệp tin hướng Byte
- FileReader và FileWriter phục vụ cho việc đọc ghi tệp tin hướng ký tự

Các hàm tạo của các lớp tương ứng để liên kết luồng với một tệp tin cụ thể

- **public void** FileInputStream (String FileName)
- **public void** FileInputStream (File file)
- **public void** FileOutputStream (String FileName)
- **public void** FileOutputStream (File file)
- **public void** FileWriter (String FileName)
- **public void** FileWriter (File file)
- **public void** FileReader (String FileName)
- **public void** FileReader (File file)

Nhập / xuất lọc

- Lọc:
 - về cơ bản được sử dụng để thích ứng các luồng theo các nhu cầu của chương trình cụ thể.
 - Bộ lọc nằm giữa luồng nhập và luồng xuất.
 - Thực hiện một số tiến trình đặc biệt trên các byte được chuyển giao từ đầu vào đến kết xuất.
 - Có thể phối hợp để thực hiện một dãy các tùy chọn lọc.

Lớp FilterInputStream

- Là lớp trừu tượng.
- Là cha của tất cả các lớp luồng nhập đã lọc.
- Cung cấp khả năng tạo ra một luồng từ luồng khác.
- Một luồng có thể đọc và cung cấp dưới dạng kết xuất cho luồng khác.
- Duy trì một dãy các đối tượng của lớp 'InputStream'
- Cho phép tạo ra nhiều bộ lọc kết xích

Lớp FilterOutputStream

- Là dạng hỗ trợ cho lớp 'FilterInputStream'.
- Là cha của tất cả các lớp luồng kết xuất.
- Duy trì đối tượng của lớp 'OutputStream' như là một biến 'out'.
- Dữ liệu ghi ra lớp này có thể sửa đổi để thực hiện các thao tác lọc, và sau đó phản hồi đến đối tượng 'OutputStream'.

Vùng đệm nhập/xuất

- Vì các thao tác với ổ cứng, mạng thường lâu hơn so với thao tác bộ nhớ trong
- Kỹ thuật sử dụng vùng đệm nhằm tăng tốc độ đọc/ghi
- Với kỹ thuật vùng đệm sẽ giảm được số lần đọc/ghi luồng
- Trong Java ta có thể tạo ra vùng đệm của các lớp :
 - `BufferInputStream`
 - `BufferOutputStream`
 - `BufferedReader`
 - `BufferWriter`

Vùng đệm nhập/xuất

- Vùng đệm:
 - Là kho lưu trữ dữ liệu.
 - Có thể cung cấp dữ liệu thay vì quay trở lại nguồn dữ liệu gốc ban đầu.
 - Java sử dụng vùng đệm nhập và kết xuất để tạm thời lập cache dữ liệu được đọc hoặc ghi vào một luồng.
- Trong khi thực hiện vùng đệm nhập:
 - Số lượng byte lớn được đọc cùng thời điểm và lưu trữ trong một vùng đệm nhập.
 - Khi chương trình đọc luồng nhập, các byte nhập được đọc vào vùng đệm nhập.

Vùng đệm nhập/xuất (tt...)

- Trong trường hợp vùng đệm kết xuất, một chương trình ghi ra một luồng.
- Dữ liệu kết xuất được lưu trữ trong một vùng đệm kết xuất.
- Dữ liệu được lưu trữ cho đến khi vùng đệm trở nên đầy, hay luồng kết xuất được xả trống.
- Kết thúc, vùng đệm kết xuất được chuyển gửi đến đích của luồng xuất.

Vùng đệm nhập/xuất

Các phương thức tạo dựng luồng đệm :

***public** BufferedInputStream(InputStream)*

***public** BufferedInputStream (InputStream in, int bufferSize)*

***public** BufferedOutputStream (OutputStream out)*

***public** BufferedOutputStream (OutputStream out, int
bufferSize)*

***public** BufferedReader (Reader in)*

***public** BufferedReader (Reader in, int bufferSize)*

***public** BufferedWriter (Writer out)*

***public** BufferedWriter (Writer out, int bufferSize)*

Lớp BufferedInputStream

- Tự động tạo ra và duy trì vùng đệm để hỗ trợ vùng đệm nhập.
- Lớp 'BufferedInputStream' là một bộ đệm, nó có thể áp dụng cho một số các đối tượng nhất định của lớp 'InputStream'.
- Cũng có thể phối hợp các tập tin đầu vào khác.
- Sử dụng vài biến để triển khai vùng đệm nhập.

Lớp BufferedInputStream (tt...)

- Định nghĩa hai phương thức thiết lập:
 - Một cho phép chỉ định kích thước của vùng đệm nhập.
 - Phương thức kia thì không.
- Cả hai phương thức thiết lập đều tiếp nhận một đối tượng của lớp 'InputStream' như một tham số.
- Nạp chồng các phương thức truy cập mà InputStream cung cấp, và không đưa vào bất kỳ phương thức mới nào.

Lớp BufferedOutputStream

- Thực hiện vùng đệm kết xuất theo cách tương ứng với lớp 'BufferedInputStream'.
- Định nghĩa hai phương thức thiết lập. Nó cho phép chúng ta ấn định kích thước của vùng đệm xuất trong một phương thức thiết lập, cũng giống như cung cấp kích thước vùng đệm mặc định.
- Nạp chồng tất cả phương thức của lớp 'OutputStream' và không đưa vào bất kỳ phương thức nào.

Lớp PrintWriter

- Thực hiện một kết xuất.
- Lớp này có phương thức bổ sung, trợ giúp in các kiểu dữ liệu cơ bản
- Lớp PrintWriter thay thế lớp 'PrintStream'
- Thực tế cải thiện lớp 'PrintStream'; lớp này dùng một dấu tách dòng phụ thuộc nền tảng để các dòng thay vì ký tự '\n'.
- Cung cấp phần hỗ trợ cho các ký tự unicode so với PrintStream.
- Các phương thức:
 - **checkError()**
 - **setError()**

Giao diện DataInput

- Được sử dụng để đọc các byte từ luồng nhị phân
- Cho phép chúng ta chuyển đổi dữ liệu từ từ khuôn dạng UTF-8 được sửa đổi Java đến dạng chuỗi
- Định nghĩa số phương thức, bao gồm các phương thức để đọc các kiểu dữ liệu nguyên thủy.

Giao diện DataInput

- `boolean readBoolean()` : Đọc giá trị logic từ luồng
- `byte readByte()` : Đọc một byte từ luồng
- `char readChar()` : Đọc một kí tự từ luồng
- `double readDouble()` : Đọc một số double từ luồng
- `float readFloat()` : Đọc một số thực từ luồng
- `void readFully(byte []b)` : Đọc một mảng byte từ luồng và ghi vào mảng
- `void readFully(byte []b, int off, int len)` : Đọc len byte từ luồng và ghi vào mảng từ vị trí off
- `int readInt()` : Đọc một số nguyên
- `String readLine()` : Đọc một xâu kí tự cho đến khi gặp kí tự xuống dòng và bỏ qua kí tự xuống dòng
- `long readLong()` : Đọc một số long
- `short readShort()` : Đọc một số short
- `int readUnsignedByte()` : Đọc một số nguyên không dấu (0..255)
- `int readUnsignedShort()` : Đọc số nguyên không dấu (0..65535)
- `String readUTF()` : Đọc một xâu kí tự Unicode
- `int skipBytes(int n)` : Bỏ qua n byte từ luồng

Giao diện `DataOutput`

- Được sử dụng để xây dựng lại dữ liệu một số kiểu dữ liệu nguyên thủy vào trong dãy các byte
- Ghi các byte dữ liệu vào luồng nhị phân
- Cho phép chúng ta chuyển đổi một chuỗi vào khuôn dạng UTF-8 được sửa đổi Java và viết nó vào trong một dãy.
- Định nghĩa một số phương thức và tất cả phương thức kích hoạt `IOException` trong trường hợp lỗi.

Giao diện DataOutput

- `void write(byte[] b)` : Ghi một mảng byte ra luồng
- `void write(byte []b, int off, int len)` : Ghi một mảng byte ra luồng tại vị trí off, len byte
- `void write(int b)` : Ghi một byte ra luồng
- `void writeBoolean(boolean b)` : Ghi một giá trị logic ra luồng
- `void writeByte(int b)` : Ghi ra luồng phần thập của b
- `void writeBytes(String s)` : Ghi một chuỗi ra luồng
- `void writeChar(int b)` : Ghi một kí tự ra luồng
- `void writeChars(String s)` : Ghi một chuỗi kí tự ra luồng
- `void writeDouble(double b)` : Ghi một số double ra luồng
- `void writeFloat(float b)` : Ghi một số thực ra luồng
- `void writeInt(int b)` : Ghi một số nguyên ra luồng
- `void writeLong(long b)` : Ghi một số long ra luồng
- `void writeShort(int b)` : Ghi một số short ra luồng
- `void writeUTF(String s)` : Ghi một chuỗi kí tự Unicode ra luồng

Lớp RandomAccessFile

- Cung cấp khả năng thực hiện I/O theo các vị trí cụ thể bên trong một tập tin.
- dữ liệu có thể đọc hoặc ghi ngẫu nhiên ở những vị trí bên trong tập tin thay vì một kho lưu trữ thông tin liên tục.
- phương thức 'seek()' hỗ trợ truy cập ngẫu nhiên.
- Thực hiện cả đọc và ghi dữ liệu.
- Hỗ trợ các cấp phép đọc và ghi tập tin cơ bản.
- Kế thừa các phương thức từ các lớp 'DataInput' và 'DataOutput'

Các phương thức của lớp RandomAccessFile

- **seek()**
- **getFilePointer()**
- **length()**

Gói java.awt.print

- Gồm có các interface
 - Pageable:
 - Định nghĩa các phương thức dùng để các đối tượng biểu thị các trang sẽ được in.
 - Chỉ định số trang đã được in, và trang hiện tại hay là tranh giới trang đã được in
 - Printable:
 - Chỉ định phương thức 'print()' sử dụng để in một trang trên đối tượng 'Graphics'
 - PrinterGraphics:
 - Cung cấp khả năng truy cập đối tượng 'PrinterJob'

Gói java.awt.print

- interface 'PrinterGraphics' cung cấp các lớp sau:
 - Paper
 - Book
 - PageFormat
 - PrinterJob
- Gói 'java.awt.print' kích hoạt các ngoại lệ:
 - PrinterException
 - PrinterIOException
 - PrinterAbortException

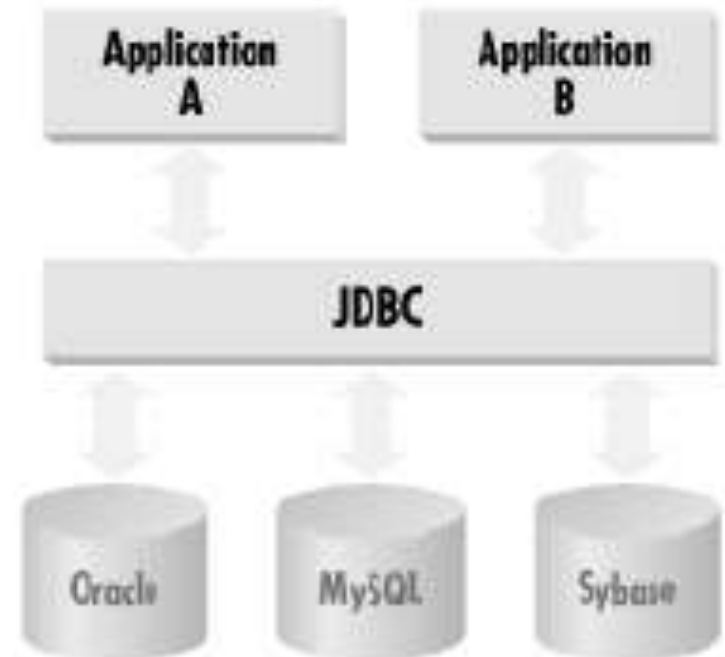
LẬP TRÌNH

KẾT NỐI CƠ SỞ DỮ LIỆU

Java Database Connectivity

Tổng Quan

- JDBC cung cấp tập các lớp và interface cho phép chương trình Java có thể làm việc được với hệ cơ sở dữ liệu
- Tập các lớp của JDBC có thể làm việc được với mọi hệ quản trị cơ sở dữ liệu



Database Driver

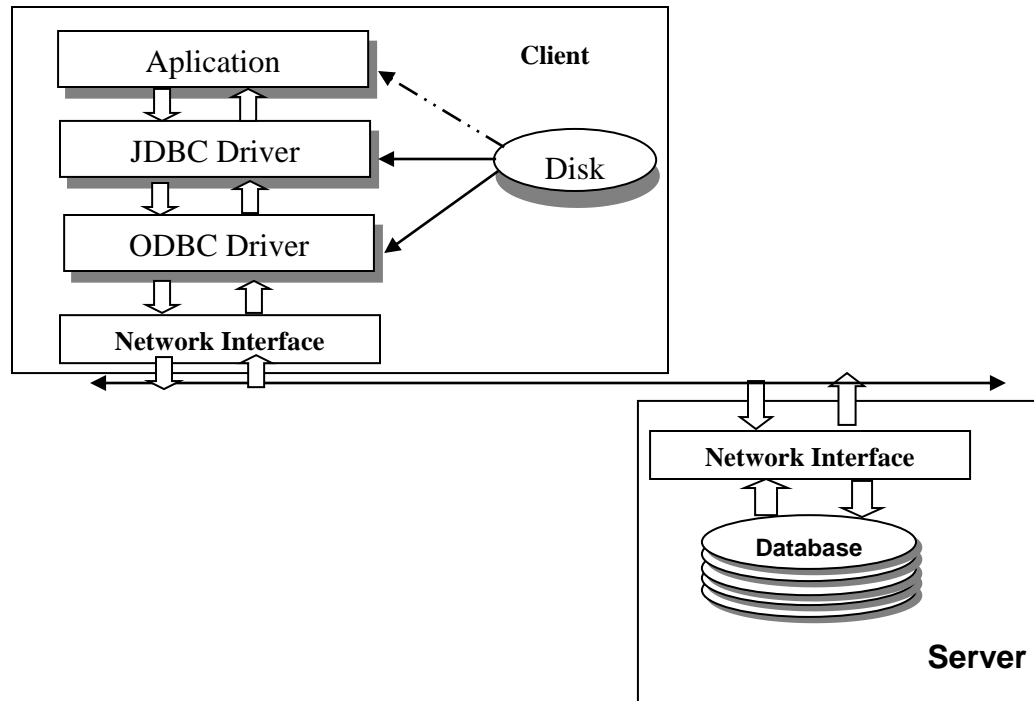
- Bảo đảm ứng dụng java tương tác với mọi cơ sở dữ liệu dưới một cách thức chuẩn và duy nhất.
- Bảo đảm những yêu cầu từ chương trình sẽ được biểu diễn trong cơ sở dữ liệu dưới một ngôn ngữ mà cơ sở dữ liệu hiểu được
- Nhận các yêu cầu từ client, chuyển nó vào định dạng mà cơ sở dữ liệu có thể hiểu được và thể hiện trong cơ sở dữ liệu.
- Nhận các phản hồi, chuyển nó ngược lại định dạng dữ liệu java và thể hiện trong ứng dụng.

JDBC Driver

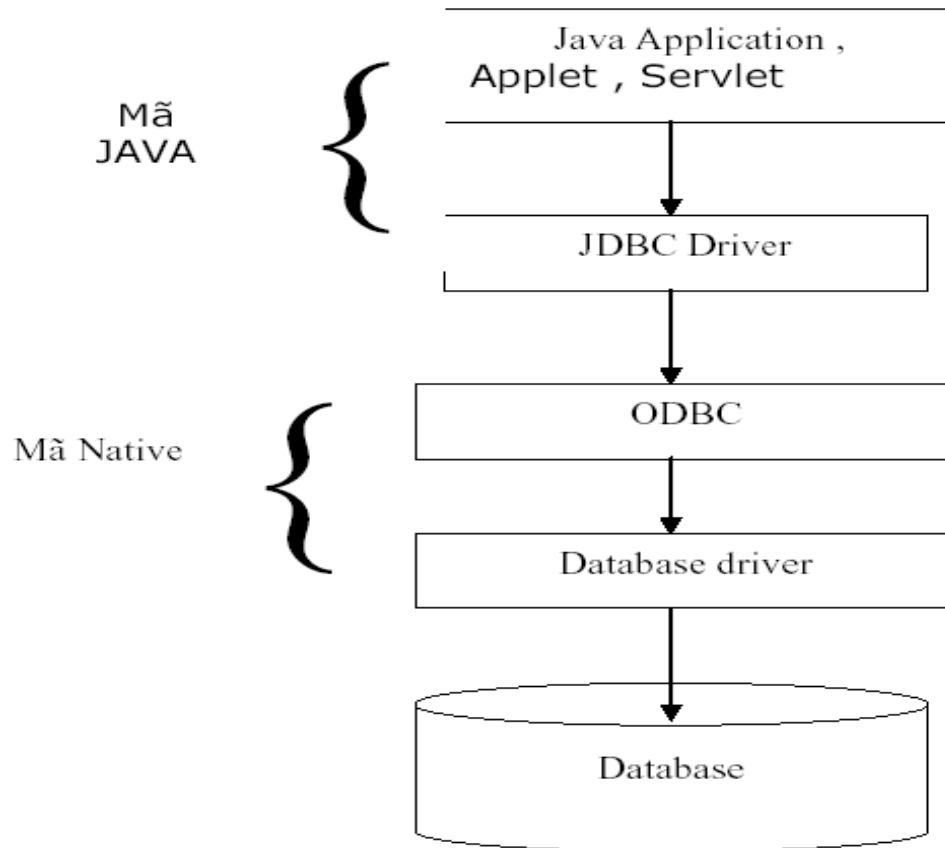
- Có 4 loại JDBC Driver
 - Loại 1 : JDBC sử dụng cầu nối ODBC
 - Loại 2 : JDBC kết nối trực tiếp với các trình điều khiển
 - Loại 3 : JDBC kết nối thông qua các ứng dụng mạng trung gian
 - Loại 4 : JDBC kết nối thông qua các trình điều khiển đặc thù ở xa
- Loại 2,3,4 nói chung được viết bởi nhà cung cấp cơ sở dữ liệu, hiệu quả hơn loại 1 nhưng thực hiện phức tạp hơn.

JDBC SỬ DỤNG CẦU NỐI ODBC

- jdk hỗ trợ cầu nối jdbc-odbc (jdbc-odbc bridge).
- Mềm dẻo nhưng không hiệu quả.



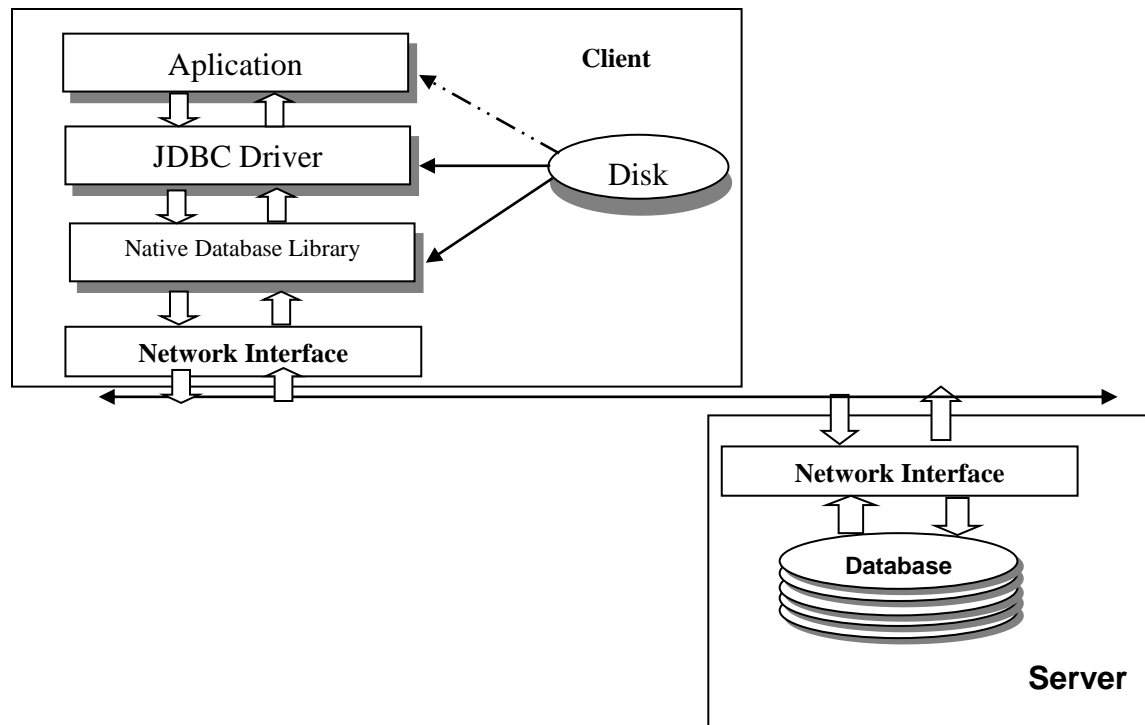
JDBC sử dụng cầu nối ODBC



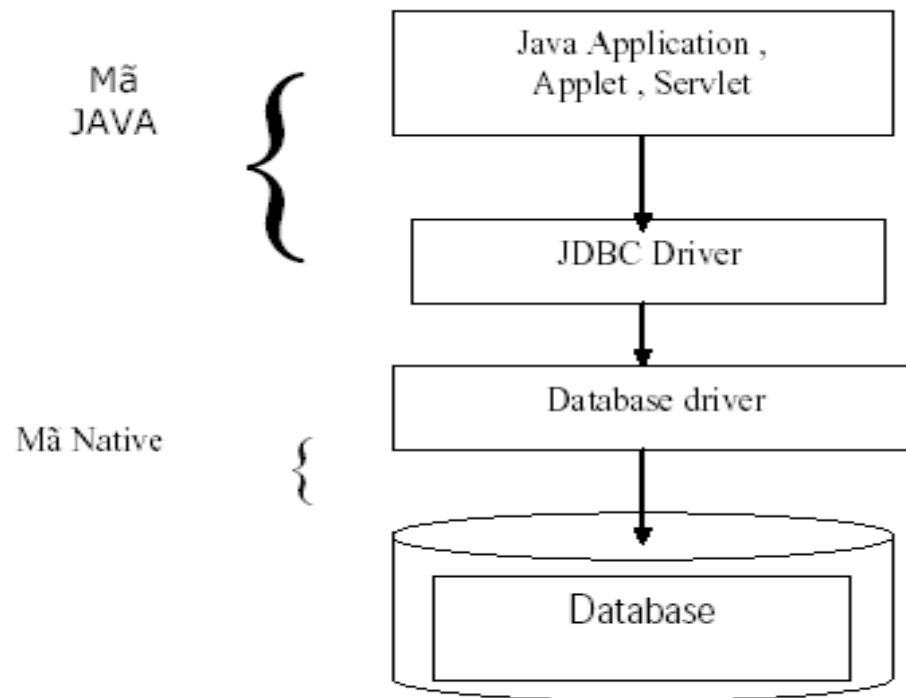
Mô hình truy cập CSDL qua cầu nối JDBC

JDBC KẾT NỐI TRỰC TIẾP VỚI CÁC TRÌNH ĐIỀU KHIỂN CƠ SỞ DỮ LIỆU

- Loại này cho phép JDBC giao tiếp trực tiếp với các driver hay các hàm API của cơ sở dữ liệu.



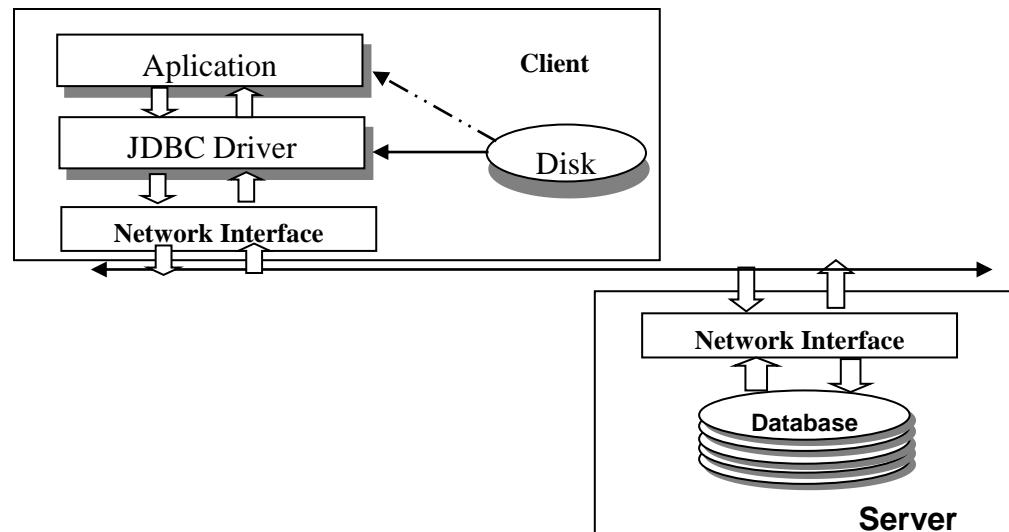
JDBC kết nối trực tiếp với các trình điều khiển cơ sở dữ liệu



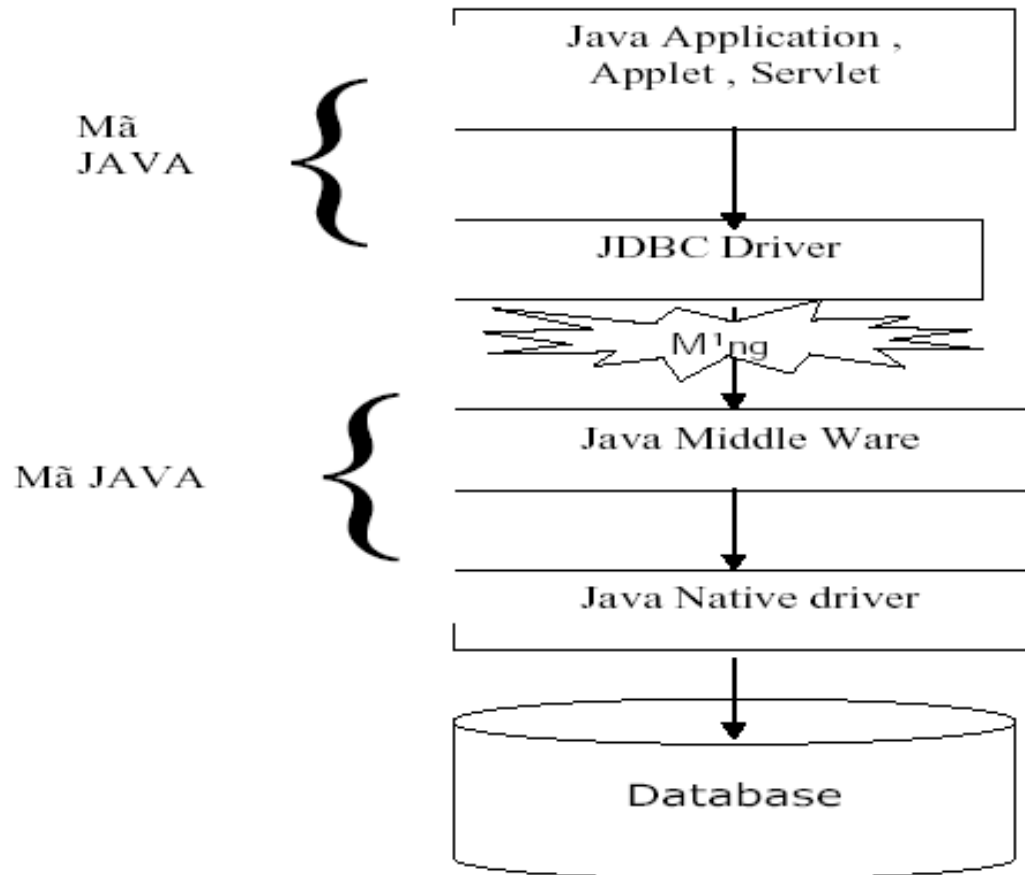
Mô hình kết nối trực tiếp

JDBC KẾT NỐI THÔNG QUA ỨNG DỤNG MẠNG TRUNG GIAN

- 100% java
- Có khả năng giao tiếp trực tiếp với hệ cơ sở dữ liệu không cần chuyển đổi



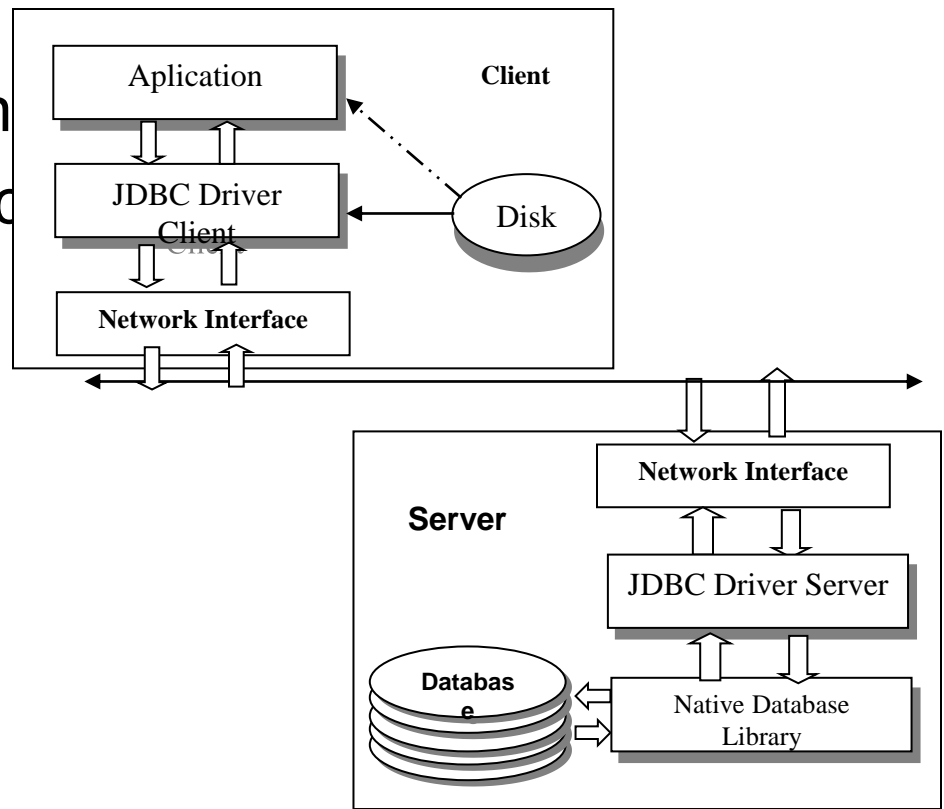
JDBC KẾT NỐI THÔNG QUA ỨNG DỤNG MẠNG TRUNG GIAN



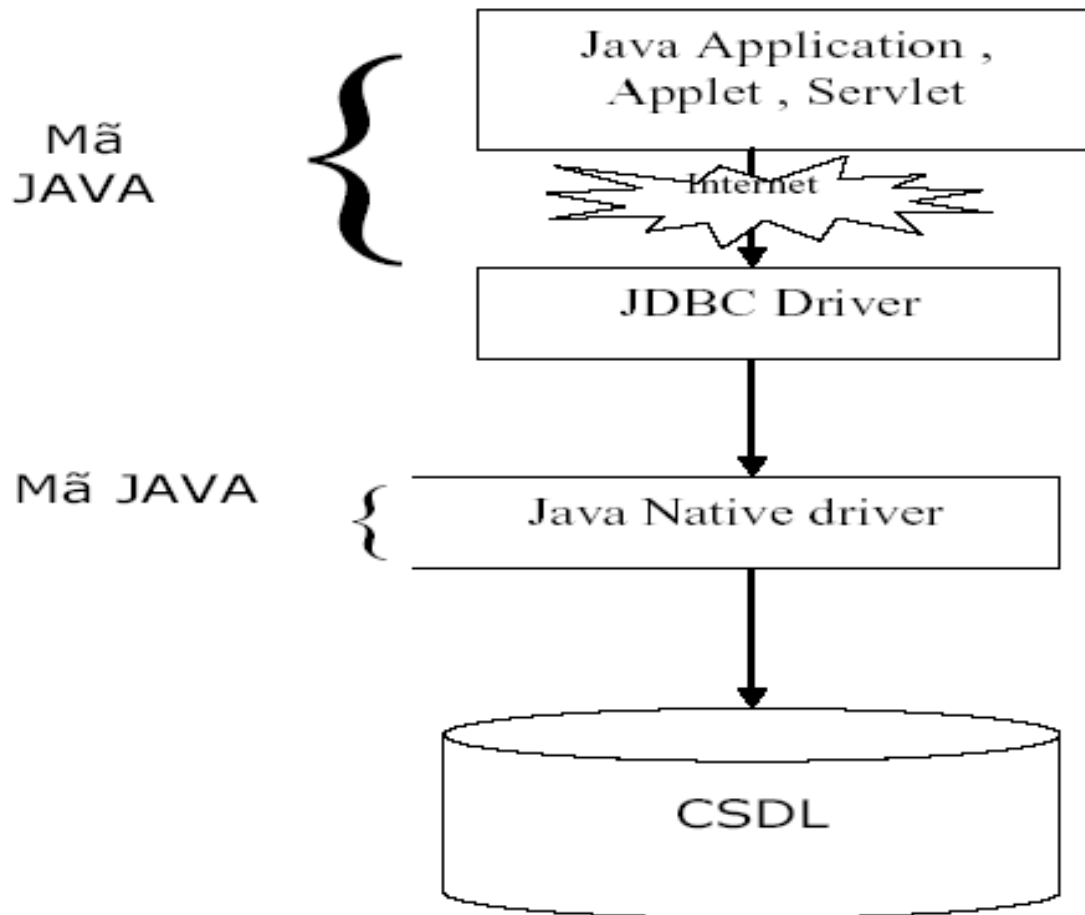
JDBC KẾT NỐI THÔNG QUA CÁC TRÌNH ĐIỀU KHIỂN ĐẶC THÙ Ở XA

- Drivers

- Có thể chuyển các yêu cầu đến các csdl nằm ở xa.
- Có thể giao tiếp với n
- Không phải của nhà c
- Tất cả bằng mã java



JDBC KẾT NỐI THÔNG QUA CÁC TRÌNH ĐIỀU KHIỂN ĐẶC THÙ Ở XA



Gói Java.sql

- Cung cấp tập hợp các lớp và interface làm việc với cơ sở dữ liệu
- Các lớp
 - DriverManager
 - Date, Time
 - Timestamp
 - Types
- Các Interfaces
 - Driver
 - Connection
 - DatabaseMetaData
 - Statement
 - PreparedStatement
 - CallableStatement
 - ResultSet
 - ResultSetMetaData

Gói Java.sql

- **CallableStatement** : Giao diện chứa các phương thức cho phép làm việc với các thủ tục lưu trữ
- **DatabaseMetaData** : Cho phép xem các thông tin về cơ sở dữ liệu
- **PreparedStatement** : Giao diện cho phép thực thi các câu lệnh SQL chứa tham số

Gói Java.sql

- **Connection** : thể hiện một kết nối đến cơ sở dữ liệu
- **Driver**: giao diện mà các trình điều khiển phải cài đặt
- **ResultSet** : thể hiện một tập các bản ghi lấy về từ cơ sở dữ liệu
- **Statement** : giao diện cho phép ta thực thi các phát biểu SQL

Gói Java.sql

- **Date** : lớp biểu diễn kiểu Date
- **DriverPropertyInfo** : Chứa các thuộc tính của trình điều khiển đã nạp
- **Timestamp** : lớp biểu diễn cho SQL TimeTemp
- **DriverManager** : lớp quản lý các trình điều khiển
- **Time** : lớp biểu diễn kiểu Time
- **Types** : lớp định nghĩa các hằng tương ứng với các kiểu dữ liệu SQL hay còn gọi là kiểu dữ liệu JDBC

Các bước để kết nối cơ sở dữ liệu

- Nạp trình điều khiển
- Tạo thông tin kết nối và đối tượng Connection
- Tạo đối tượng Statement để thực thi các lệnh truy vấn SQL
- Xử lý dữ liệu
- Đóng kết nối

Nạp Driver

- Lớp DriverManager chịu trách nhiệm nạp driver và tạo kết nối đến cơ sở dữ liệu.
- Để nạp và đăng kí trình điều khiển, ta gọi lệnh :

Class.forName(String)

Ví dụ :

DriverManager.registerDriver(new sun.jdbc.odbc.JdbcOdbcDriver());

Hoặc:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Tương đương

new sun.jdbc.odbc.JdbcOdbcDriver();

Chú ý : Với các trình điều khiển khác nhau thì String của phương thức Class.forName(String) sẽ khác nhau

Tạo thông tin kết nối

- Tiếp tục tạo đối tượng Connection bằng cách gọi phương thức getConnection của lớp DriverManager để yêu cầu trình điều khiển nạp bởi Class.forName() trước đó tiếp nhận thông tin và thực thi kết nối

```
conn = DriverManager.getConnection(url,  
    "username", "password");
```

Trong đó :

+ url : chuỗi nêu lên đặc điểm csdl có dạng

jdbc:subprotocol:subname

- subprotocol : giao thức con tương ứng với csdl

- subname : tên csdl

+ username : tên đăng nhập csdl

+ password : mật khẩu đăng nhập csdl

Tạo thông tin kết nối

Ví dụ :

Nạp trình điều khiển của Access :

```
Class.forName("sun.jdbc.odbc. JdbcOdbcDriver ");  
Connection conn = DriverManager.getConnection("  
jdbc:odbc:DBName","username","password");
```

Nạp trình điều khiển của MySQL :

```
Class.forName("com.mysql.jdbc.Driver");  
Connection conn = DriverManager.getConnection(  
"jdbc:mysql://ServrName:3306/DBName","UserName","Password");
```

Nạp trình điều khiển của SQL Server :

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");  
Connection conn = DriverManager.getConnection("  
jdbc:microsoft:sqlserver://ServerName:1433;  
DatabaseName=DBName","UserName","Password");
```

Đối tượng Statement

Tất cả các lệnh tác động đến cơ sở dữ liệu đều có thể thực hiện thông qua một trong ba đối tượng :

- **Statement** : Thực thi các câu lệnh SQL không có tham số
- **PreparedStatement** : Thực thi các câu lệnh SQL có chứa tham số
- **CallableStatement** : Làm việc với các thủ tục lưu trữ

Đối tượng Statement

- Đối tượng Connection chứa liên kết trực tiếp đến csdl.
- Sử dụng đối tượng Connection để tạo đối tượng Statement.

`Statement state = conn.createStatement();`

- Đối tượng này có nhiệm vụ gửi các câu lệnh sql đến cơ sở dữ liệu
- **executeQuery(String)** or **executeUpdate(String)** method
- Cùng một đối tượng Statement có thể sử dụng cho nhiều câu lệnh sql khác nhau.

Đối tượng Statement

- Có 3 phương thức thực thi :
 - executeQuery()
 - executeUpdate()
 - execute()
- The executeQuery()
 - Nhận câu lệnh SQL (select) làm đối số, trả lại đối tượng ResultSet

Ví dụ :

```
ResultSet rs = state.executeQuery("SELECT * FROM  
Books");
```

Đối tượng Statement

- Phương thức `executeUpdate()`
 - Nhận các câu lệnh sql dạng cập nhật
 - Trả lại số nguyên biểu thị số hàng được cập nhật.
 - UPDATE, INSERT, or DELETE.
- Phương thức `execute()`
 - Được áp dụng cho trường hợp không rõ loại sql nào được thực hiện.
 - Được áp dụng cho trường hợp câu lệnh sql được tạo ra tự động bởi chương trình.

ResultSet

- Chứa một hoặc nhiều hàng dữ liệu từ việc thực hiện câu lệnh truy vấn.
- Có thể lấy dữ liệu từng hàng dữ liệu một trong ResultSet.
- Sử dụng phương thức next() để di chuyển đến hàng dữ liệu tiếp theo trong ResultSet.
- Hàm next() trả lại true chỉ rằng hàng chứa dữ liệu, trả lại false hàng cuối cùng, không chứa dữ liệu.
- Thực hiện

```
while (rs.next())  
{  
    // examine a row from the results  
}
```

ResultSet

- **next** : di chuyển con trỏ sang tập bản ghi kế tiếp, trả về true nếu thành công, ngược lại false
- **previous** : di chuyển con trỏ về bản ghi trước bản ghi hiện tại
- **last** : di chuyển con trỏ về bản ghi cuối cùng trong tập bản ghi
- **first** : di chuyển con trỏ về bản ghi đầu tiên trong tập bản ghi

ResultSet

- **beforeFirst** : di chuyển con trỏ về trước bản ghi đầu tiên trong tập bản ghi
- **afterLast** : di chuyển con trỏ về sau bản ghi cuối cùng trong tập bản ghi
- **absolute(int pos)** : di chuyển con trỏ về bản ghi thứ pos tính từ bản ghi đầu tiên nếu pos là số dương (hoặc tính từ bản ghi cuối cùng nếu pos là số âm)
- **relative(int pos)** : di chuyển con trỏ về trước bản ghi hiện tại pos bản ghi nếu pos là số âm, hoặc di chuyển về phía sau pos bản ghi so với bản ghi hiện tại nếu pos là số dương

ResultSet

- Để lấy dữ liệu ở các cột trên mỗi hàng của ResultSet, ta dùng các phương thức

- **gettype(int | String)**

- Đối số là chỉ số cột tính từ 1.
- Áp dụng cho các cột có kiểu dữ liệu là int, float, Date.....

Ví dụ :

- `String isbn = rs.getString(1); // column 1`
- `float price = rs.getDouble("Price");// column 2`

ResultSetMetadata

- Đối tượng này cho biết thông tin về ResultSet

```
ResultSet rs = stmt.executeQuery(SQLString);  
ResultSetMetaData rsmd = rs.getMetaData();  
int numberOfColumns = rsmd.getColumnCount();  
getColumnName(int column)
```

Database Metadata

- Đối tượng này cho biết thông tin về cơ sở dữ liệu.

Chương trình mẫu

```
import java.sql.*;
import javax.sql.*;
public class Connect {
    public static void main(String args[]) throws ClassNotFoundException,SQLException
    {
        System.out.println("Ket noi CSDL");
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url="jdbc:odbc:DBName";
            Connection conn=DriverManager.getConnection(url,"login","password");
            Statement stmt=conn.createStatement();
            String sql1="INSERT INTO TableName(Id,TenKH,DiaChi,Luong)
            VALUES('8','Nguyen C','HCM','900)";
            stmt.executeUpdate(sql1);
            //Cap nhat du lieu
            String sql2="UPDATE TableName SET Luong=Luong+Luong*0.1";
            int n=stmt.executeUpdate(sql2);
            if (n < 1) System.out.println("Khong co ban ghi nao duong cap nhat");
            else System.out.println("Co "+ n +" ban ghi duong cap nhat");
```

Chương trình mẫu

```
String sql="SELECT Id,TenKH,DiaChi,Luong FROM TableName";  
ResultSet rs=stmt.executeQuery(sql);  
while (rs.next())  
{ int id=rs.getInt("Id");  
double l=rs.getDouble("Luong");  
String s=rs.getString("TenKH");  
String d=rs.getString("DiaChi");  
System.out.println("ID=" +id + " " + s+ " " + d + " Luong=" + l) ;  
}  
} catch(SQLException e) {System.out.println("Error");}  
}  
}
```

Chương trình mẫu

```
import java.sql.*;

class JDBCdemo1 {
    public static void main(String[] args) {
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con=DriverManager.getConnection("jdbc:odbc:DATA");
            Statement stmt = con.createStatement();
            String sql="Select * from Table1";
            ResultSet rs = stmt.executeQuery(sql);
            ResultSetMetaData rsmd = rs.getMetaData();
            int numberOfColumns = rsmd.getColumnCount();
            for(int j=1; j<=numberOfColumns;j++) {
                System.out.println(rsmd.getColumnLabel(j));    }
            while(rs.next()) {
                for(int i=1; i<=numberOfColumns;i++){
                    System.out.println(rs.getObject(i));    }}
                rs.close();
                stmt.close();
            } catch(Exception e){ System.out.println("Error " + e); }
        }
    }
}
```

Hết !!!