Mục Lục

[**Khái niệm về lập trình hướng đối tượng**. 1](#_Toc494205715)

[**Data type**. 1](#_Toc494205716)

[Miền giá trị của data types 2](#_Toc494205717)

[Các kiểu dữ liệu trong java 2](#_Toc494205718)

[Các kiểu biến 4](#_Toc494205719)

[So sánh trong java 5](#_Toc494205720)

[**Wrapper class là gi?unboxing và autoboxing?** 6](#_Toc494205721)

[**String trong java** 6](#_Toc494205722)

[Java String immutable 7](#_Toc494205723)

[String buffer,String builder 9](#_Toc494205724)

[SubString và StringSequence 10](#_Toc494205725)

[**JaVa IO** 13](#_Toc494205726)

[**Multithread java** 16](#_Toc494205727)

[Các phươg thức được sử dụng trong thread 17](#_Toc494205728)

[Thread piority 19](#_Toc494205729)

[synchronized trong java. 19](#_Toc494205730)

[Concurency in java 20](#_Toc494205731)

[**Java JDBC** 21](#_Toc494205732)

[**Regular expressions trong Java** 22](#_Toc494205733)

[**Collection and generic trong java** 25](#_Toc494205734)

[Collection Interface 25](#_Toc494205735)

[Set Interface 26](#_Toc494205736)

[List Interface 26](#_Toc494205737)

[Queue Interface 26](#_Toc494205738)

[Map Interface 27](#_Toc494205739)

[Iterator trong java 28](#_Toc494205740)

[Generic là gì? 28](#_Toc494205741)

[**Annotation and Enum** 30](#_Toc494205742)

# Khái niệm về lập trình hướng đối tượng.

Lập trình hướng đối tượng là việc lập trình bằng cách tạo ra các đối tượng ảo dựa vào những đặc tính ,hành động của những đối tượng tương ứng trong thực tế,điều này được thể hiện qua các lớp (class), đối tượng (Object) mà hành động là các hàm(method) còn đặc điểm chính là các biến(variable).

Có 4 đặc tính quan trọng của OOP

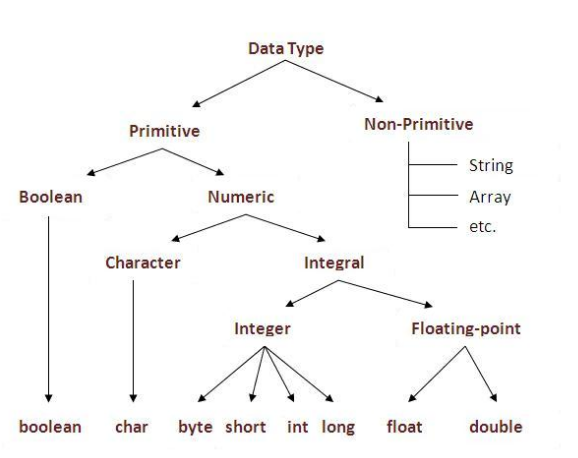
Tính bao đóng

Tính kế thừa

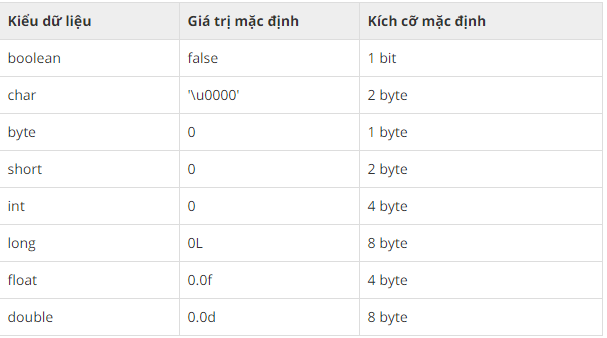
Tính đa hình

Tính trừu tượng

# Data type.

****

## Miền giá trị của data types



## Các kiểu dữ liệu trong java

-Trong java tồn tại 2 kiểu dữ liệu :kiểu dữ liệu nguyên thủy và kiểu dữ liệu tham chiếu

Kiểu dữ liệu nguyên thủy: ví dụ int,float,bool .....

Kiểu dữ liệu tham chiếu:String,các object.

Về cách khai báo thì khi khởi tạo biến có kiểu dữ liệu nguyên thủy ,Java sẽ tạo ra và cấp phát giá trị cho biến đó trong stack(vd int a=10)

Khi khai báo biến có kiểu dữ liệu tham chiếu (vd Sinhvien a=new Sinhvien()); Java sẽ khởi tạo 1 biến a trong stack ,biến này lưu trữ địa chỉ của vùng nhớ được cấp phát cho object nằm trong vùng nhớ heap.

*Reference gần giống với con trỏ trong C nhưng giới hạn đến giá trị các ô nhớ có thể truy cập được.*

Ví dụ:

Sinhvien a=new Sinhvien();

Khai báo và cấp phát bộ nhớ cho biến tham chiếu a tức là khai báo 1 biến a nằm trong stack frame của chương trình, biến này lưu trữ địa chỉ của vùng nhớ được cấp phát cho object nằm trong vùng nhớ heap.

Sinhvien b=Sinhvien a; thao tác này tương đương với việc copi dữ liệu nằm trong a sang b,nói các khác là b cũng chứa địa chỉ vùng nhớ được cấp phát cho object nằm trong vùng nhớ heap. (trong C là a và b cùng trỏ đến 1 nơi)

lúc này b có thể thay đổi dữ liệu của a (giống giống con trỏ trong C) và nếu so sánh (a==b) sẽ trả về true.

*=>Lí do vì sao khi nhập dữ liệu kiểu tham chiếu thì trong java phải dùng for thay vì foreach.*

Giả sử:

Sinhvien a[]=new Sinhvien[5];

for(int i =0;i<n;i++){

a[i]=new SinhVien;

a[i].nhap();//cái này đúng

}

for(sinhvien sv :a ){

sv=new sinhvien();

sv.nhap();//cái này ko chạy được

}

Lí do:Vòng foreach trên tương đương với :

for(int i=0;i<a.length;i++){

Sinhvien sv=a[i];//tao ra bien tham chiếu sv gan = a[i]

sv=new sinhvien();//gán sv = đối tượng sinhviên

sv.nhap();//lúc này sv vừa tạo ra đã trỏ đến giá trị đúng trong vùng nhớ heap nhưng a[i] vẫn trỏ về null=> ko lưu đc giá trị

}

## Các kiểu biến

**Biến local**

* Phạm vi trong phương thức, constructor, khối.
* Tồn tại cùng phương thưc, constructor, khối.
* Cần có giá trị khởi tạo trước khi sử dụng.
* biến local lưu trên stack

**Biến instance**

* Phạm vi đối tượng.
* Tồn tại cũng đối tượng.
* Access Modifier (public, protected, default, private).
* Sẽ có giá trị mặc định được gán sau khi đối tượng được tạo. Kiểu số thì là 0, kiểu boolean thì là false...
* Lưu trong bộ nhớ Heap.

**Biến static**

* Phạm vi lớp.
* Tồn tại cùng chương trình.
* Access Modifier (public, protected, default, private).
* Lưu trong bộ nhớ static.
* Giá trị mặc định: kiểu sô là 0, kiểu boolean là false...

Cách đặt tên: public static final thì tên viết HOA và có giá trị khởi tạo (vì là final) còn không thì như biến local và instance. Ví dụ:

public static final int TOTAL = 10;

public static int count;

Cách hiểu về phương thức kiểu static: static ~~ tĩnh,nghĩa là địa chỉ của biến\_phương thức static cố định.Việc gọi đến các biến\_phương thức nói chung tương đương với việc truy cập vào địa chỉ của biến\_phương thức đó trong bộ nhớ,tuy nhiên khi khai báo kiểu non\_static thì địa chỉ của biến và phương thức luôn thay đổi(cấp phát động) và cần có đối tượng có chứa địa chỉ để truy cập vào ô nhớ đó.ví dụ như:

**SinhVien sv=new SinhVien("fresher");**

**sv.data;**

Còn khi dùng từ khóa static thì biến đó được lưu vào vùng nhớ static (cố định)=>nghĩa là mọi thành phần đều có địa chỉ chứa giá trị của biến đó=>có thể truy cập trực tiếp mà không cần phải thông qua các thành phần khác.

Bonus:

Khi gọi các biến\_method Static chỉ cần ClassName.StaticName là vì:

Cần có ClassName để tránh sự nhầm lẫn,chứ thực chất ClassName chưa phải là đối tượng nên nó không thể trỏ đến địa chỉ mang giá trị của StaticName

## So sánh trong java

Trong java có 2 cách so sánh kiểu dữ liệu tham chiếu là == và equals,về bản chất 2 cách so sánh này giống nhau,đều là so sánh địa chỉ chứa object nằm trong heap,tuy vậy method equal thường được override để so sánh các objects(vd như trong String thì equals dùng để so sánh giá trị String thay về địa chỉ.. )

Còn với kiểu dữ liệu nguyên thủy,ko sử dụng được method nên chỉ xài đc ==

Ví dụ:

SinhVien a=new SinhVien("fresher");

SinhVien b=new SinhVien("fresher");

System.out.println(a==b)// false.

System.out.println(a.equals(b))//false

int a=7;

int b=7;

System.out.println(a==b);//true

Phương thức hashCode() được sử dụng khi chúng ta insert object vào HashTable, HashMap hoặc HashSet.

Khi insert một object vào hashtable, phương thức hashCode() sẽ trả về giá trị hash code của object, và giá trị này được sử dụng để xác định vị trí của object được lưu trữ trong hashtable.

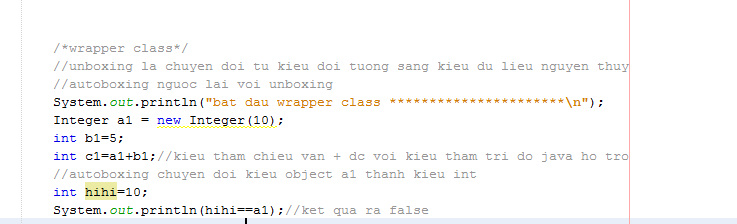
Phương thức equals dùng thường được override để so sánh các object =>thường được sử dụng trong Collection để xem đã tồn tại object đó trong collection kia chưa.

# Wrapper class là gi?unboxing và autoboxing?

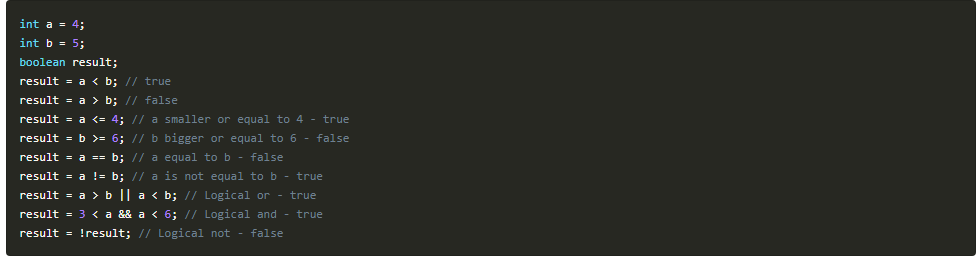
Wrappper Class là những Class JAVA định nghĩa ra tương ứng với những kiểu dữ liệu cơ bản .

ví dụ Integer a=new Integer();

Cụ thể thì như sau:



Các câu so sánh cơ bản



# String trong java

String trong java là 1 mảng các kí tự

String trong java có kiểu tham chiếu,tức là nó có thể sử dụng các phương thức mà Java định nghĩa sẵn.

String trong Java mang đặc tính của cả kiểu dữ liệu tham chiếu và của cả kiểu dữ liệu nguyên thủy,tức là:

Java hỗ trợ JavaStringPool

String Pool là 1 vùng nhớ trong heap,đây là nơi lưu trữ String ,cụ thể khi khai báo 1 String literal bất kì,Java sẽ kiểm tra trong Stringpool đã chứa String đó chưa,nếu có rồi thì Java sẽ khai báo 1 biến nằm trong stackframe của chương trình sau đó cho biến đó trỏ đến String trong Stringpool,còn nếu chưa tồn tại thì sẽ cấp phát bộ nhớ cho String đó trong Stringpool rồi tạo ra 1 biến trong stackframe rồi cho biến đó trỏ vào địa chỉ chứa String vừa tạo.

## Java String immutable

String trong Java là không thể thay đổi được,tức là:

Khi khai báo 1 biến kiểu String,ví dụ:String s="fresher"; thì biến s được lưu trong stack frame của chương trình ,s lưu trữ địa chỉ của vùng nhớ được cấp phát cho object nằm trong vùng String pool,giá trị "fresher " chứa trong String pool sẽ không đổi .Cụ thể thì :

String s = "quang";

String s1 = s;

s1=s1+"fresher";

System.out.println("hascode của s1 là:" + s1.hashCode());

System.out.println("hascode của s là:" + s.hashCode());

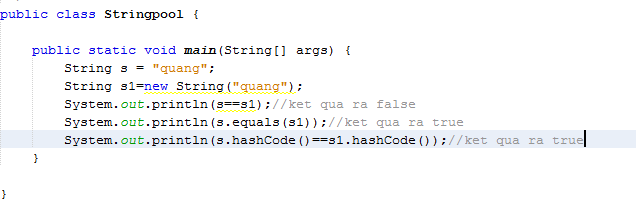
Mặc dù đã cho s1 tham chiếu đến địa chỉ chứa giá trị của s,nhưng sau khi thay đổi giá trị của s1 thì chỉ có s1.hashcode thay đổi còn s.hascode vẫn không đổi từ đó=> khi thay đổi giá trị của chuỗi thì Java sẽ sinh ra một chuỗi mới trong String pool rồi cho s1 trỏ đến cuỗi mới chứ không thay đổi giá trị của chuỗi gốc,đây chính là **Java String immutable**

Tiếp theo là về cách khai báo String:

Có 2 cách khai báo String là:

String s="fresher";

String s=new String("fresher");

Lưu ý là 2 cách khai báo này hoàn toàn khác nhau,nếu khai báo String s="fresher" thì java sẽ lưu trữ biến s trong stack frame của chương trình,biến s sẽ lưu địa chỉ của "fresher " trong String pool. Còn khi khai báo String s=new String("frehser") thì java sẽ tạo ra 2 đối tượng,1 lưu trong heap và 1 lưu trong string pool,s sẽ trỏ đến đối tượng nằm trong heap,đối tượng nằm trong Pool chỉ được sử dụng khi khai báo kiểu String literal.

Ở dòng so sánh đầu tiên so sánh s==s1 ra false vì s trỏ đến String pool còn s1 trỏ vào heap

Dùng hàm s.equals(s1) ra true vì 2 String giống nhau//equals trong String đã được override.

Kết quả s.hashCode()==s1.hashCode() ra true,giả sử s1 được lưu trong heap thì kết quả sẽ ra false => khi khai báo String s=new String("quang"); thì giá trị quang vẫn được lưu vào String pool

Sau đó lại xem lại,nếu "quang" được lưu trong String pool thì tại sao s==s1 lại ra false ?

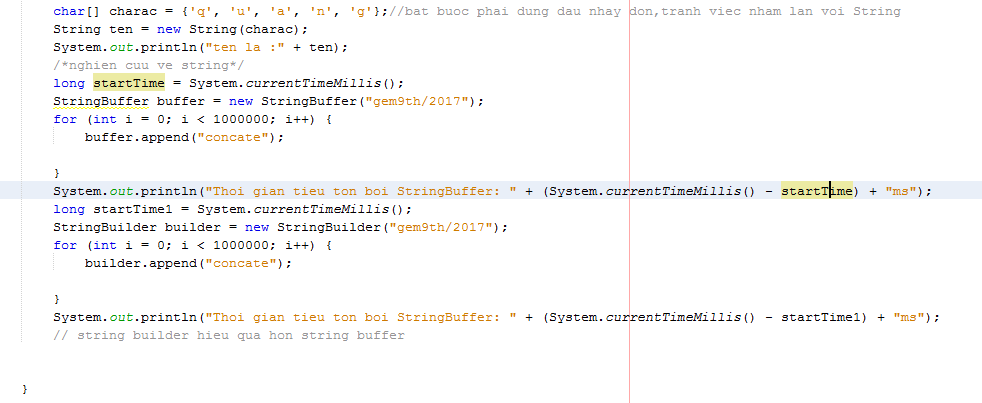
Biến nằm trong stack chỉ tham chiếu đến 1 obj trong heap => ko có chuyện s trỏ đến cả obj trong heap và obj trong stringpool đc.Vấn đề thứ 2 là tại sao hashcode lài giống nhau: Do String là đối tượng đặc biệt => đã override lại hàm hashcode//hashcode là 1 cái mã hóa cho cái ánh xạ giữa biến trong stack và địa chỉ ô nhớ,nghĩa là 2 đối tượng cùng trỏ đến 1 địa chỉ thì hashcode =nhau

## String buffer,String builder

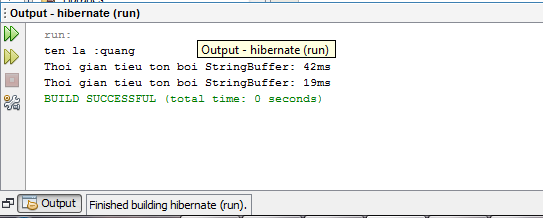
StringBuilder hoặc StringBuffer sẽ giúp java cải hiện hiệu suất so với việc sử dụng quá nhiều String,String thuộc kiểu immutable còn StringBuffer và StringBuilder thuộc loại mutable

Như đã khẳng định ở phần đầu, hai lớp trên đều thuộc lọai **mutable**, chúng ta có thể thay đổi được các giá trị của chúng, hay nói cách khác các chuỗi chứa trong StringBuffer, StringBuilder có thể thay đổi được giá trị. Nhờ đặc tính này mà khi thay đổi các chuỗi trong những đối tượng này JVM không phải tạo các đối tượng mới (đối tượng tạm) và vấn đề về performance như kể trên với String được giải quyết triệt để ở StringBuffer, StringBuilder

Cuối cùng, bạn có muốn biết sự khác biệt giữa StringBuilder và StringBuffer không? Hai lớp này được thiết kế với mục đích giống nhau, đó là thao tác hiệu quả với chuỗi, chúng cũng có các phương thức giống nhau để làm việc với chuỗi. Điểm khác biệt đáng lưu ý: StringBuffer thuộc loại**synchronized** do đó các phương thức của nó đều là “thread safe” (thích hợp với xử lý đa luồng – multi thread), trong khi StringBuilder thì ngược lại, không **synchronized.**Với đặc tính “thread safe”, các phương thức của StringBuffer sẽ chạy chậm hơn so với StringBuilder, vì vậy nếu không làm về Multi-thread bạn nên chọn StringBuilder thay vì chọn StringBuffer ví dụ về hiệu năng của String buffer và builder trong đơn thread.



Kết quả



## SubString và StringSequence

SubString trả về String,SubSequence trả về Char Sequence tức là: Trả về 1 mảng các kí tự chỉ có thể đọc được - nghĩa là kiểu CharSequence không sử dụng được các method như replace()... mà chỉ sử dụng được các method dạng lấy giá trị hoặc tính toán dựa trên CharSequence đó.CharSequence có thể được ép sang String dễ dàng.

**Interface và abstract class trong java**

Trong java thì tính trừu tượng được thể hiện qua interface và abstract class

Java không có đa kế thừa,tuy vậy lại hỗ trợ implement nhiều interface

Interface là 1 class đặc biệt chứa vỏ các phương thức,bắt buộc các class implement interface đó phải override lại các phương thức có trong interface.

Abstract class: là một class cha cho tất cả các class có cùng bản chất. Bản chất ở đây được hiểu là kiểu, loại, nhiệm vụ của class. Hai class cùng hiện thực một interface có thể hoàn toàn khác nhau về bản chất.

–     Interface: là một **chức năng** mà bạn có thể thêm và bất kì class nào. Từ chức năng ở đây không đồng nghĩa với phương thức (hoặc hàm). Interface có thể bao gồm nhiều hàm/phương thức và tất cả chúng cùng phục vụ cho một **chức năng**.

Vậy, bạn không nên nhầm lẫn khi nói về việc một class được **implement** hay **extend**. Nhiều người thường hay đồng nhất là không phân biệt hai từ này, nhưng chính chúng đã nói lên sự khác biệt giữa interface và abstract class. Bạn chỉ có thể thừa kế (extend) từ một class và chỉ có thể hiện thực (implement) các chức năng (interface) cho class của mình.

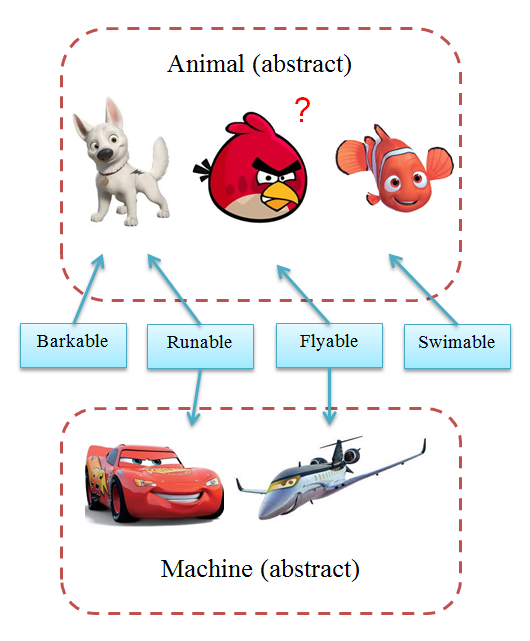
Theo cách ngắn gọn, quan hệ giữa một class khi thừa kế một abstract class được gọi là **is-a**, và một class khi hiện thực một interface được gọi là **can-do** (hoặc **–able**).

Hãy xem ví dụ sau, tôi có:

–          Interface: Barkable, Runable, Flyable, Swimable.

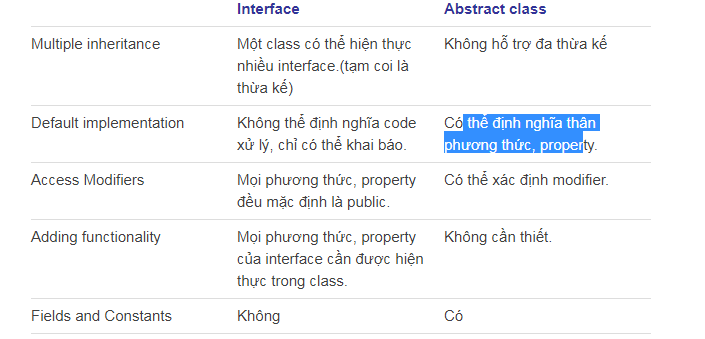
–          Abstract class Animal và các sub class: Bolt, AngryBird và Nemo.

–          Abstract class Machine và các sub class: McQueen, Siddeley.

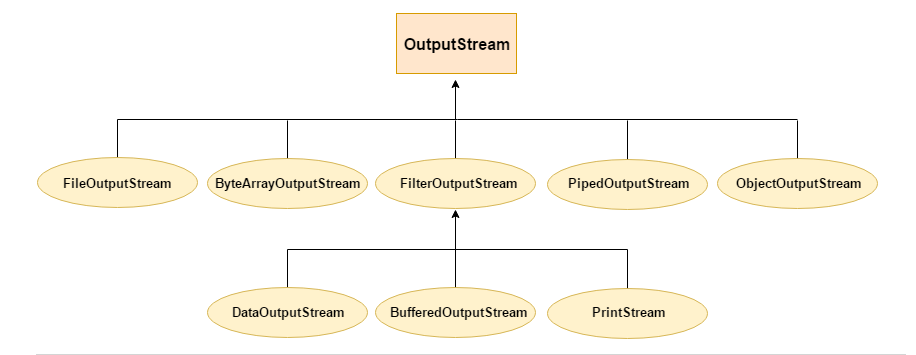
[](https://yinyangit.wordpress.com/2012/01/15/oop-interface-vs-abstract-class/abstract-class-interface-oop/)

Như bạn thấy, mặc dù cả McQueen và Bolt đều được hiện thực interface Runable, nhưng chúng hoàn toàn thuộc hai loại khác nhau. Và tất nhiên một class có thể **can-do**nhiều thứ, ví dụ như Bolt có thể chạy và “bow wow”.

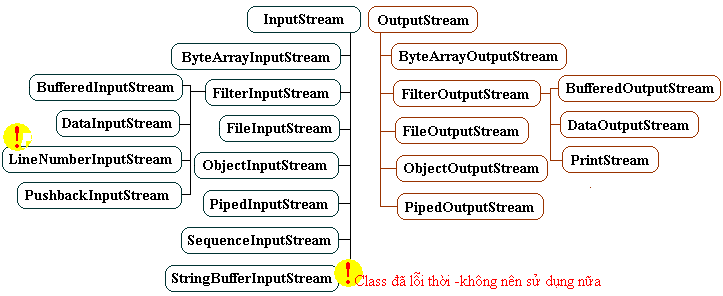
[](https://yinyangit.wordpress.com/2012/01/15/oop-interface-vs-abstract-class/angry-bird-fly/)



# JaVa IO



Luồng nhị phân dẫn đầu bởi 2 class **InputStream**và **OutputStream**dưới 2 class này là rất nhiều các class con , nếu so sánh về lực lượng theo cách này mà nói thì họ hàng luồng nhị phân (Binary Stream) nhiều và phức tạp hơn hẳn so với họ hàng luồng ký tự (Charactor Stream).Với **JDK1.5** trong luồng nhị phân có 2 class được khuyến cáo là không nên sử dụng là **LineNumberInputStream**và **StringBufferInputStream**vì đã bị lỗi thời .



2- Class InputStream & OutputStream

Class **InputStream**là một class trừu tượng vì vậy bạn không thể khởi tạo đối tượng **InputStream**thông qua chính class**InputStream**. Tuy nhiên class này rẽ ra nhiều nhánh thông qua các class con thừa kế nó. Tùy vào các tình huống bạn có thể tạo đối tượng **InputStream**từ các Constructor của các class con.

trong java IO nguyên tắc đọc một file đó là . Đầu tiên là bạn phải tạo ra một file và cung cấp đường dẫn file đó cho java , sau đó từ file này java mới tạo ra một luồng Stream ,cuối cùng từ luồng stream đó mình mới dùng một lớp nào đó để xử lí trên các luồng này (tùy vào mục đích và kiểu dữ liệu đc truyền).

**ObjectInputStream**, **ObjectOutputStream** cho phép bạn đọc hoặc ghi một Object vào luồng. Các Object này phải là kiểu ***Serializable***(Nghĩa là có thể sắp hàng).(phải implemnet serializable)

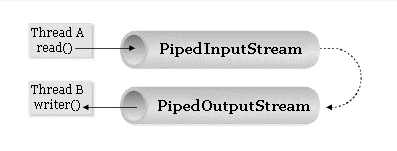
**ByteArrayInputStream**bao bọc một mảng các **byte** (byte[] buf) và thông qua **ByteArrayInputStream**truy cập phần tử mảng .. 

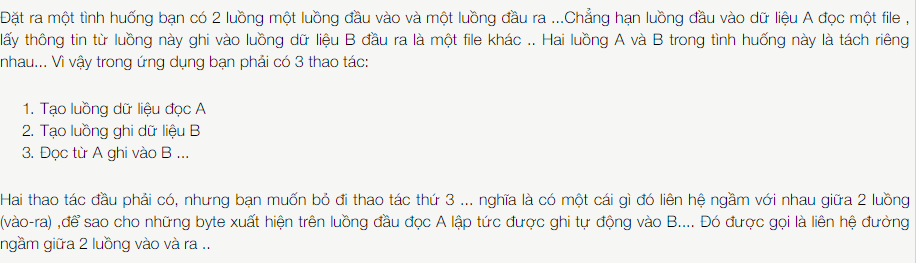
**ByteArrayOutputStream**là một luồng các **byte**, bên trong đối tượng này chứa một mảng các **byte** (byte[] buf) có khả năng tự tăng kích cỡ khi số **byte** của luồng tăng lên.Mỗi khi luồng được ghi vào các **byte** thì chính là gán tiếp **byte** đó vào các vị trí mảng chưa được gán ..

 Khi mảng đầy phần tử thì chương trình tạo mảng mới có độ dài lớn hơn và copy các phần tử của mảng cũ vào ...(Đó là cách tự lớn lên của mảng các **byte**như đã nói trên)

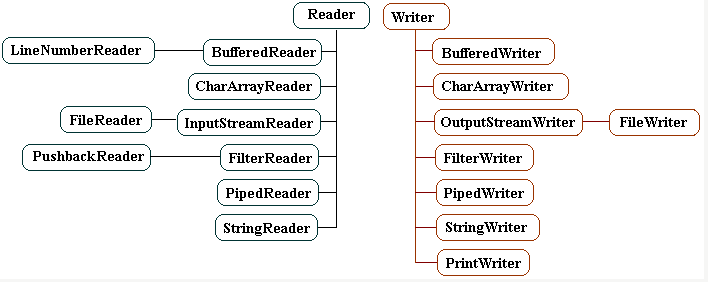
Thông thường bạn đã quen thuộc với việc đọc một file nào đó và thu được một luồng đầu vào .Nhưng trong thực tế đôi khi bạn cần đọc từ nhiều file và lấy các dữ liệu đó ghép với nhau để ghi thành 1 file khác chẳng hạn .Vậy là ý tưởng ghép nhiều luồng đầu vào với nhau để thành một luồng lớn hơn nối đuôi nhau . Chúng ta đang nói đến class **java.io.SequenceInputStream**. Khái niệm này không có tương ứng cho luồng đầu ra ...nghĩa là luôn chỉ có 1 luồng đầu ra

**Luồng dữ liệu đường ngầm nhị phân (PipedInputStream,PipedOutputStream)**

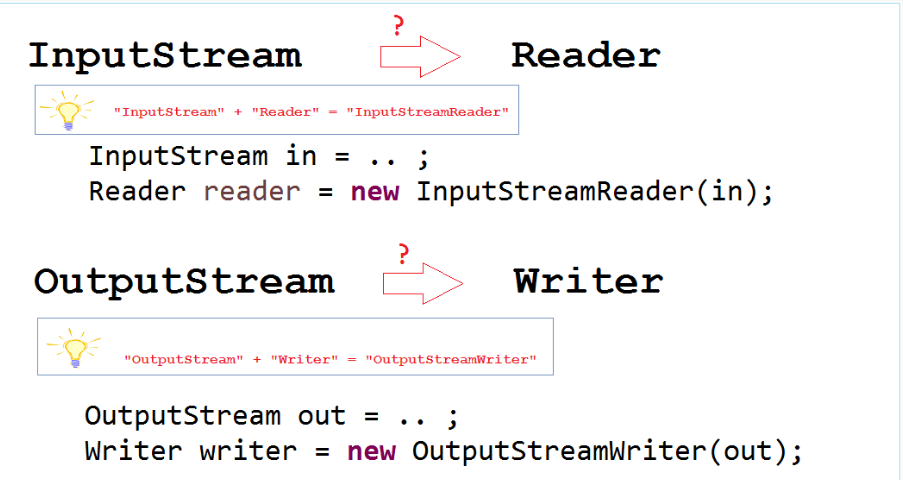




**I/O trong JaVa sử dụng luồng kí tự**

****

**Chuyển đổi từ luồng kí tự sang luồng nhị phân**

****

**...Tham khảo thêm tại:** **http://acegik.net/blog/java/concurrency/lap-trinh-cocurrency-trong-java-bang-multithreading.html**

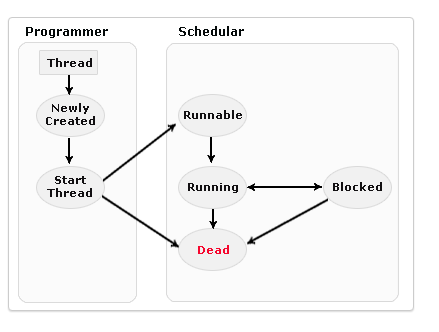
# Multithread java

Thread là đơn vị nhỏ nhất của mã thực thi mà đoạn mã đó thực hiện một nhiệm vụ cụ thể.

Một ứng dụng có thể được chia nhỏ thành nhiều nhiệm vụ và mỗi nhiệm vụ có thể được giao cho một thread.

Với cơ chế multithreading ứng dụng của bạn có thể thực thi đồng thời nhiều dòng lệnh cùng lúc. Có nghĩa là bạn có thể làm nhiều công việc đồng thời trong cùng một ứng dụng của bạn. Có thể hiểu một cách hết sức đơn giản : hệ điều hành với cơ chế đa nhiệm cho phép nhiều ứng dụng chạy cùng lúc. Thì với cơ chế đa luồng, mỗi ứng dụng của bạn có thể thực hiện được nhiều công việc đồng thời.  
Vòng đời của một thread

**Cụ thể**

****

* New: Khi một thread được tạo ra (bằng toán tử new) nó sẽ ở trạng thái New, là trạng thái đầu tiên trong vòng đời của nó. Thread giữ ở trạng thái này cho đến khi nó được chương trình kích hoạt thông qua lời gọi phương thức start().
* Runnable: Sau khi được chương trình kích hoạt, thread chuyển sang trạng thái runnable. Ở trạng thái này, thread đang thực thi các lệnh của nó.
* Waiting: Thread ở trạng thái waiting (chờ đợi) khi nó bị tạm thời ngừng thực thi và chờ trong khi một thread khác được thực thi. Thread này quay trở lại trạng thái runnable khi nhận được tín hiệu cho phép được thực thi tiếp.
* Blocked: Thread rơi vào trạng thái blocked khi nó thực thi đến lệnh/khối lệnh được đồng bộ (synchronized statement/block) mà lệnh/khối lệnh này đang được một thread khác thực thi. Thread này quay lại trạng thái runnable khi nó đến lượt thực thi khối lệnh được đồng bộ.
* Dead: Thread chuyển từ trạng thái runnable sang trạng thái terminated khi nó thực hiện xong công việc. Dead là trạng thái cuối cùng trong vòng đời của thread. Những thread ở trạng thái này sẽ bị chương trình kết thúc và loại bỏ.

**Có 2 cách khởi tạo thread**

- Bạn có thể implement Runnable interface.  
- Bạn có thể extend class Thread mặc định trong java rồi override lại phương thức run()

## Các phươg thức được sử dụng trong thread

1. **public void run():**Được sử dụng để thực hiện hành động cho một thread..
2. **public void start():**Bắt đầu thực hiện thread. JVM gọi phương thức run() trên thread.
3. **public void sleep(long miliseconds):**Làm cho thread hiện tại tạm ngừng thực thi cho số mili giây quy định.
4. **public void join():**Đợi cho một thread chết.ví dụ có 3 thread:thread1;thread2;thread3 khi sử dụng thread1.join() thì đợi thread 1 chạy xong rồi thread2 và 3 mới đc chạy
5. **public void join(long miliseconds):**Đợi cho một thread chết với các mili giây quy định.ví dụ thread1.join(1500) thì sau 1500ms thread1s duoc chay thi các thread còn lại mới đc chạy
6. **public int getPriority():**Trả về mức độ ưu tiên của thread.
7. **public int setPriority(int priority):**Gán mức độ ưu tiên của thread
8. **public String getName():**Trả về tên của thread.
9. **public void setName(String name):**Thay đổi tên của thread.
10. **public Thread currentThread():**Trả về tham chiếu của thread đang được thi hành.//giống this trong class
11. **public int getId():**Trả về id của thread.
12. **public Thread.State getState():**Trả về trạng thái của thread.
13. **public boolean isAlive():**Kiểm tra nếu thread còn sống.
14. **public void yield():**Làm cho các đối tượng thread đang thực thi để tạm thời tạm dừng và cho phép các thread khác để thực thi.
15. **public void suspend():**Được sử dụng để hoãn lại các thread (không dùng nữa).
16. **public void resume():**Được sử dụng để tiếp tục các thread đang bị hoãn (không dùng nữa).
17. **public void stop():**Được sử dụng để dừng thread (không dùng nữa).
18. **public boolean isDaemon():**Kiểm tra nếu thread là một luồng hiểm.
19. **public void setDaemon(boolean b):**Đánh dấu thread là luồng hiểm hoặc luồng người dùng.
20. **public void interrupt():**Ngắt thread.
21. **public boolean isInterrupted():**Kiểm tra nếu thread đã bị ngắt.
22. **public static boolean interrupted():**Kiểm tra nếu thread hiện tại đã bị ngắt.

## Thread piority

Độ ưu tiên của thread,trong java quy định độ ưu tiên của thread từ 1-10,độ ưu tiên càng lớn thì càng có cơ hội được chạy trước(ko chắc chắn,còn tùy thuộc vào việc thực thi các thread và phụ thuộc nhiều vào từng platform.)

## synchronized trong java.

Khi chúng ta bắt đầu 2 hay nhiều Thread trong cùng 1 chương trình, có thể xảy ra tình huống nhiều Thread cố gắng truy cập vào cùng 1 file hay 1 đối tượng nhất định gây ra tình trạng xung đột dữ liệu, mất dữ liệu.

Ví dụ, nếu nhiều luồng cố gắng để ghi vào cùng một tập tin, vì vậy tập tin có thể bị hỏng dữ liệu vì một trong những Thread có thể ghi đè dữ liệu trong khi một thread khác cũng mở cùng một tập tin, cùng lúc đó một thread khác nữa có thể được đóng tập tin đó.

Ngôn ngữ lập trình Java cung cấp một cách rất tiện lợi cho việc tạo ra Thread và đồng bộ hóa các nhiệm vụ bằng cách sử dụng các khối **synchronized** (đồng bộ). Bạn giữ các nguồn tài nguyên được chia sẻ trong khối này.

Một khối Synchronized block đánh dấu một phương thức hay một khối mã là được đồng bộ. Sử dụng khối đồng bộ trong Java có thể tránh xảy ra xung đột.

Có 2 cách là :

**1.Đồng bộ Object**

**2. Đồng bộ phương thức**

* Cách 1 hay được sử dụng khi ta có 1 **Object** chung và gọi các phương thức bên ngoài để cùng xử lý, tính toán trên **Object** dữ liệu chung
* Cách 2 hay được sử dụng khi trong 1 **Class** có nhiều hàm cùng sử dụng 1 **Object** để xử lý và **Class** này được gọi thực thi ở 1 phương thức khác.

Và một điều quan trọng cũng như cơ bản cần nhớ là đồng bộ sẽ chỉ chơi với đồng bộ (synchronized). Nếu một threadA muốn đồng bộ dữ liệu với một thread B thì cả hai thằng đều phải dùng các câu lệnh(hay method) có từ khóa synchronized.(vd trong hàm run của cả 2 thread đều sử dụng 1 method có từ khóa synchonize)

Điều quan trọng thứ 2 là khi synchronized(object) tức là object đó đang bị khóa bởi luồng(tạm gọi ThreadA) chứa câu lệnh đó. Và bất kỳ thread nào làm việc gì liên quan đến synchronized đối với object này, mà trong thời gian thằng ThreadA đang lock object đó đều phải đợi cho đến khi thằng ThreadA xong việc với object.

## Concurency in java

Tính toán đồng thời là quá trình xử lý nhiều nhiệm vụ được diễn ra trong cùng một khoảng thời gian. Khái niệm này trái ngược với tính toán tuần tự (Sequentially computing) là quá trình xử lý chỉ một nhiệm vụ trong mọi thời điểm, xong việc này mới bắt đầu xử lý việc tiếp theo. Một khái niệm khác có liên quan cũng cần phân biệt với Tính toán đồng thời, đó là khái niệm Tính toán song song (Parallel computing). Mặc dù cả hai khái niệm đều có ý nghĩa là xử lý nhiều nhiệm vụ cùng một lúc, tuy nhiên trong tính toán song song, hai lệnh thực thi cho hai nhiệm vụ khác nhau hoàn toàn độc lập với nhau và có thể xảy ra vào cùng thời điểm, trên hai máy tính hoặc bộ xử lý riêng biệt. Tính toán song song không thể thực hiện trên một máy tính chỉ có bộ xử lý đơn lõi được. Còn trong Tính toán đồng thời, về mặt tổng thể các nhiệm vụ diễn ra cùng nhau trong một khoảng thời gian chung, nhưng các lệnh xử lý các nhiệm vụ có khả năng phải thực thi xen kẽ nhau liên tiếp trên cùng một bộ vi xử lý đơn lõi trong một máy tính. Nói một cách đơn giản, Tính toàn đồng thời thực hiện nhiều nhiệm vụ cùng lúc, bằng những câu lệnh thực thi tuần tự.

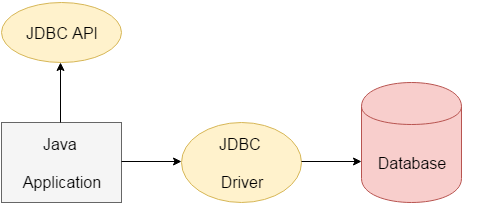
=> nghĩa là các thread được thực hiện 1 cách xen kẽ chứ không phải chúng chạy đồng thời,khi 1 thread tạm ngừng hoạt động thì thread còn lại sẽ chạy,(kiểu tối ưu thời gian bằng cách khi 1 thread tạm ngừng thì sẽ delay 1 khoảng thời gian,trong thời gian đó thì thread khác sẽ chạy)

Trong [Java](http://en.wikipedia.org/wiki/Java_concurrency), lập trình đồng thời được thiết kế theo mô hình Multithreading (xử lý đa luồng), trong đó chương trình bao gồm nhiều Thread (luồng), mỗi Thread chạy theo các bước thực thi lệnh của riêng mình, nhưng tất cả các luồng đều có khả năng truy cập được bất kỳ đối tượng và tài nguyên của chương trình. Người lập trình phải chịu trách nhiệm trong việc phân bổ và phối hợp chính xác giữa các Thread trong việc truy cập (đọc/ghi) các đối tượng và tài nguyên chung.

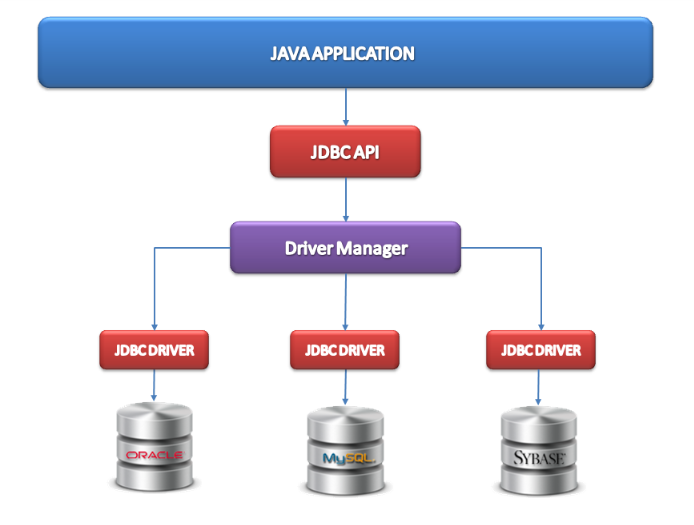
Mỗi chương trình Java được chạy trên nền một máy ảo Java ([Java Virtual Machine](http://en.wikipedia.org/wiki/Java_virtual_machine)). Máy ảo Java là dạng Máy ảo tiến trình (Process Virtual Machine), do đó mỗi chương trình Java là một tiến trình của hệ điều hành. Hàm main() trong chương trình Java sẽ tạo ra [Thread](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html) mặc định. Một chương trình Java đơn giản chỉ có một Thread được tạo ra từ hàm main(). Máy ảo Java cho phép chương trình Java có thể tạo ra nhiều Thread khác. Nếu muốn xử lý đồng thời nhiều nhiệm vụ, người lập trình phải viết mã lệnh để hàm main() gọi đến để tạo ra các Thread khác.

/\*Phần này mới tạo đc demo cho thread conccurency,sleeping,synchonize trong thread\*/

# Java JDBC

****

JDBC là viết tắt của **Java Database Connectivity**, nó là ứng dụng mã nguồn mở cho Java, giúp ứng dụng Java thực hiện kết nối, làm việc với CSDL.  
Nó cho phép ta thực hiện các thao tác chuy xuất, update dữ liệu với CSDL quan hệ bằng việc sử dụng các câu lệnh SQL.



Ứng dụng Java sử dụng JDBC làm việc với CSDL thông qua trình tự 7 bước như sau.

1. Tạo kết nối đến database
2. Gửi SQL query đến database sử dụng JDBC driver tương ứng
3. JDBC driver kết nối đến database
4. Thực thi câu lệnh query để lấy kết quả trả về (số bản ghi lấy được, số bản ghi được update/delete)
5. Gửi dữ liệu đến ứng dụng thông qua Driver Manager
6. Xử lý dữ liệu trả về
7. Đóng (giải phóng) kế nối đến database

# Regular expressions trong Java

Regular Expression hay còn gọi là biểu thức chính quy được dùng để xử lý chuỗi nâng cao thông qua biểu thức riêng của nó, những biểu thức này sẽ có những nguyên tắc riêng và bạn phải tuân theo nguyên tắc đó thì biểu thức của bạn mới hoạt động được. Ngoài cái tên gọi Regular Expression ra thì nó còn có thể viết tắt thành Regex.

Nguyên tắc hoạt động của biểu thức Regex là so khớp dựa vào khuôn mẫu, khuôn mẫu được xây dựng từ các quy tắc căn bản của biểu thức Regex.  
Các ký tự thường dùng trong Regex:

1 . Khớp (match) với bất kỳ ký tự nào

2 ^regex Biểu thức chính quy phải khớp tại điểm bắt đầu

3 regex$ Biểu thức chính quy phải khớp ở cuối dòng.

4 [abc] Thiết lập định nghĩa, có thể khớp với a hoặc b hoặc c.

5 [abc][vz] Thiết lập định nghĩa, có thể khớp với a hoặc b hoặc c theo sau là v hay z.

6 [^abc] Khi dấu ^ xuất hiện như là nhân vật đầu tiên trong dấu ngoặc vuông, nó phủ nhận mô hình. Điều này có thể khớp với bất kỳ ký tự nào ngoại trừ a hoặc b hoặc c.

7 [a-d1-7] Phạm vi: phù hợp với một chuỗi giữa a và điểm d và con số từ 1 đến7

8 X|Z Tìm X hoặc Z.

9 XZ Tìm X và theo sau là Z.

10 $ Kiểm tra kết thúc dòng.

11 \d Số bất kỳ, viết ngắn gọn cho [0-9]

12 \D Ký tự không phải là số, viết ngắn gon cho [^0-9]

13 \s Ký tự khoảng trắng, viết ngắn gọn cho [ \t\n\x0b\r\f]

14 \S Ký tự không phải khoản trắng, viết ngắn gọn cho [^\s]

15 \w Ký tự chữ, viết ngắn gọn cho [a-zA-Z\_0-9]

16 \W Ký tự không phải chữ, viết ngắn gọn cho [^\w]

17 \S+ Một số ký tự không phải khoảng trắng (Một hoặc nhiều)

18 \b Ký tự thuộc a-z hoặc A-Z hoặc 0-9 hoặc \_, viết ngắn gọn cho [a-zA-Z0-9\_].

19 \* Xuất hiện 0 hoặc nhiều lần, viết ngắn gọn cho {0,}

20 + Xuất hiện 1 hoặc nhiều lần, viết ngắn gọn cho {1,}

21 ? Xuất hiện 0 hoặc 1 lần, ? viết ngắn gọn cho {0,1}.

22 {X} Xuất hiện X lần, {}

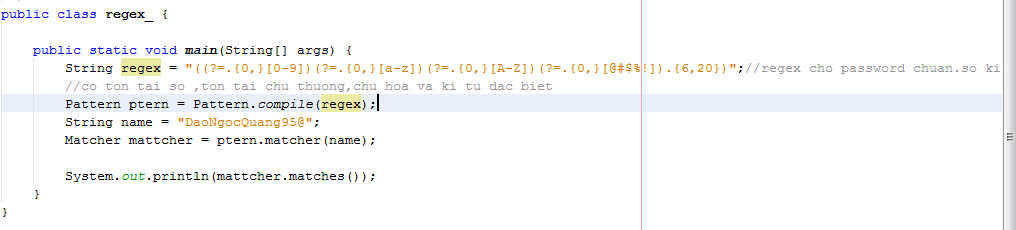
23 {X,Y} Xuất hiện trong khoảng X tới Y lần.

24 \*? \* có nghĩa là xuất hiện 0 hoặc nhiều lần, thêm ? phía sau nghĩa là tìm kiếm khớp nhỏ nhất.

**Sử dụng String.matches(String)**

**String.matches(String)** được dùng để kiểm tra chuỗi từ người dùng có phù hợp với biểu thức do chúng ta viết với **regex**hay không**=> trả về true hoặc fails**

* **Pattern**là một đối tượng mẫu, một phiên bản đã được biên dịch của một biểu thức chính quy. Nó không có constructor public, và chúng ta sẽ sử dụng method tĩnh **compile(String)** để tạo đối tượng, với tham số là biểu thức chính quy.
* **Matcher**là một phương tiện để so khớp chuỗi dữ liệu đầu vào với đối tượng **Pattern**đã được tạo ra ở trên. Class này không có cấu tử public, và chúng ta lấy đối tượng này thông qua method **matcher(String)** của đối tượng **Pattern**. Với tham số đầu vào **String** là văn bản cần kiểm tra.
* **PatternSyntaxException**sẽ bị ném ra nếu biểu thức chính quy có ngữ pháp không chính xác.



Viết cái regex như sau:

(.{6-20}) quy định 6-20 kí tự

lần lượt thêm các đoạn (?=.{0,}[0-9]) quy định kí tự nào được thêm vào regex và độ dài của chúng,ở đây {0,} có nghĩa là ít nhất 1 đoạn chuỗi là số thuộc [0-9];?= có nghĩa là có ít nhất 1 đoạn chuỗi như thế.

tương tự với các đoạn khác

1 vài ví dụ khác về regex

**^[ \s]+|[ \s]+$ : bỏ nút space trong String**

**^[\_A-Za-z0-9-]+(\\.[\_A-Za-z0-9-]+)\*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)\*(\\.[A-Za-z]{2,})$ :kiểm tra mail đúng định dạng hay ko: cụ thể:**

**^:bắt đầu,$ kết thúc,**

[A-Za-z0-9]+ tương đương với [A-Za-z0-9]{1,} :ít nhất 1 kí tự//bắt buộc phải có

(\\. dấu chấm bình thường

[A-Za-z0-9]+ tương tự như trên

)\* có thể có hoặc không tương đường với {0,}

@ chuỗi phải có kí tự @

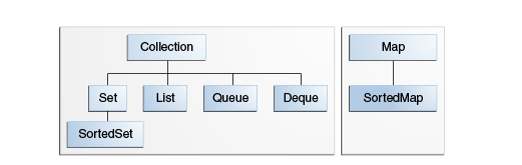
2 cái group sau tương tự

gruop cuối là đuôi . nên cần quy định số kí tự ít nhất là 2 :cụ thể {2,}

tham khảo thêm tại: http://laptrinhandroid.vn/bieu-thuc-chinh-quy-regular-expressions-trong-java/

# Collection and generic trong java

**1.Phân loại**

****

-Một danh sách ( *List*) là một danh sách tuần tự các đối tượng, nơi mà các đối tượng giống nhau có thể xuất hiện một hoặc nhiều lần. Ví dụ: [1, 7, 1, 3, 1, 1, 1, 5]. Và bạn có thể nói về phần tứ "thứ N" trong danh sách. Bạn có thể thêm một phần tử vào bất kỳ một vị trí nào trong danh sách, thay đổi một phần tử nào tại một vị trí nào đó trong danh sách, hoặc xóa một phần tử tại một vị trí bất kỳ trong danh sách.

-Hàng đợi ( ***Queue***) cũng là một tập hợp tuần tự, nhưng bạn chỉ có thể chạm vào phần tử đứng ở đầu hàng đợi. Tất cả các phần tử được trèn vào phía cuối của hàng đợi và xóa phần tử đầu tiên của hàng đợi. Bạn có thể biết được có bao nhiêu phần tử trong hàng đợi, nhưng bạn không thể tìm ra hoặc nói về phần tử thứ N, bạn chỉ có thể thấy nó khi nó đứng lên đầu tiên của hàng đợi.

***-Set***là một tập hợp không tuần tự, và nó không cho phép trùng lặp. Bạn không thể nói về phần tử thứ N thậm chí là phần tử đầu tiên, vì nó không có sự tuần tự. Bạn có thể thêm hoặc xóa các phần tử, và có thể tìm ra nếu thực sự nó tồn tại (Ví dụ "7 có nằm trong tập hợp này không?",nó tương tự tập hợp trong số học ấy).

## Collection Interface

Collection Interface định nghĩa những phương thức cơ bản khi làm việc với tập hợp, đây là gốc cũng là nền móng để từ đó xây dựng lên cả bộ thư viện Java Collection Framework. Collection Interface được kế thừa từ Iterable Interface nên các bạn có thể dễ dàng duyệt qua từng phần tử thông qua việc sử dụng Iterator.

## Set Interface

Set (tập hợp) là kiểu dữ liệu mà bên trong nó mỗi phần tử chỉ xuất hiện duy nhất một lần (tương tự như tập hợp trong toán học vậy) và Set Interface cung cấp các phương thức để tương tác với set. Set Interface được kế thừa từ Collection Interface nên nó cũng có đầy đủ các phương thức của Collection Interface.  
Một số class thực thi Set Interface thường gặp:

* **TreeSet**: là 1 class thực thi giao diện Set Interface, trong đó các phần tử trong set đã được sắp xếp.//chú ý xem phần tử có compare đc ko,nếu ko thì phải imp comparable interface
* **HashSet**: là 1 class implement Set Interface, mà các phần tử được lưu trữ dưới dạng bảng băm (hash table),nếu cần quan tâm đến tốc độ thì nên xài cái này
* **EnumSet**: là 1 class dạng set như 2 class ở trên, tuy nhiên khác với 2 class trên là các phần tử trong set là các enum .

## List Interface

List (danh sách) là cấu trúc dữ liệu tuyến tính trong đó các phần tử được sắp xếp theo một thứ tự xác định. List Interface định nghĩa các phương thức để tương tác với list cũng như các phần tử bên trong list. Tương tự như Set Interface, List Interface cũng được kế thừa và có đầy đủ các phương thức của Collection Interface.

Một số class thực thi List Interface thường sử dụng:

* **ArrayList**: là 1 class dạng list được implement dựa trên mảng có kích thước thay đổi được.
* **LinkedList**: là một class dạng list hoạt động trên cơ sở của cấu trúc dữ liệu danh sách liên kết đôi (double-linked list)
* **Vector**: là 1 class thực thi giao diện List Interface, có cách thực lưu trữ như mảng tuy nhiên có kích thước thay đổi được, khá là tương tự với ArrayList, tuy nhiên điểm khác biệt là Vector là synchronized, hay là đồng bộ, có thể hoạt động đa luồng mà không cần gọi synchronize một cách tường minh
* **Stack:** cũng là 1 class dạng list, Stack có cách hoạt động dựa trên cơ sở của cấu trúc dữ liệu ngăn xếp (stack) với kiểu vào ra LIFO (last-in-first-out hay vào sau ra trước) nổi tiếng.

## Queue Interface

Queue (hàng đợi) là kiểu dữ liệu nổi tiếng với kiểu vào ra FIFO (first-in-first-out hay vào trước ra trước), tuy nhiên với Queue Interface thì queue không chỉ còn dừng lại ở mức đơn giản như vậy mà nó cũng cấp cho bạn các phương thức để xây dựng các queue phức tạp hơn nhiều như priority queue (queue có ưu tiên), deque (queue 2 chiều), … Và cũng giống như 2 interface trước, Queue Interface cũng kế thừa và mang đầy đủ phương thức từ Collection Interface.  
Một số class về Queue thường sử dụng:

* **LinkedList**: chính là LinkedList mình đã nói ở phần List
* **PriorityQueue. Hàng đợi này lưu trữ các phần tử trong nội bộ theo trật tự tự nhiên của phần tử (nếu các phần tử đó so sánh được với nhau - thi hành *Comparable*) hoặc một bộ so sánh *Comparator* được sét đặt cho PriorityQueue.lưu ý:các phần tử trong quece phải so sánh đc nếu ko là ăn ngoại lệ,có thể implement interface comparable để compare kiểu dữ liệu trong class chứa các object cần so sánh**
* **ArrayDeque**: là 1 dạng deque (queue 2 chiều) được implement dựa trên mảng

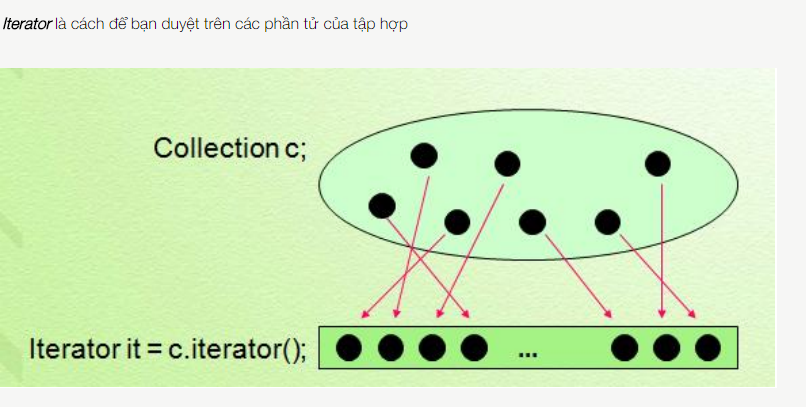
## Map Interface

Map (đồ thị/ánh xạ) là kiểu dữ liệu cho phép ta quản lý dữ liệu theo dạng cặp key-value, trong đó key là duy nhất và tương ứng với 1 key là một giá trị value. Map Interface cung cấp cho ta các phương thức để tương tác với kiểu dữ liệu như vậy. Không giống như các interface ở trên, Map Interface không kế thừa từ Collection Interface mà đây là 1 interface độc lập với các phương thức của riêng mình.  
Dưới đây là một số class về Map cần chú ý:

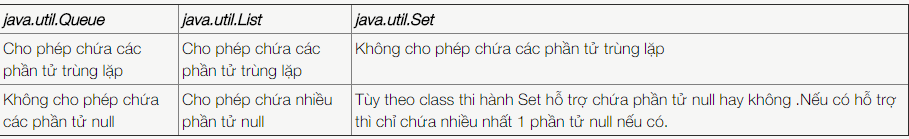
* **TreeMap**: là class thực thi giao diện Map Interface với dạng cây đỏ đen (Red-Black tree) trong đó các key đã được sắp xếp,do mặc định các node của cây đỏ đen thuộc loại có thể so sánh đc
* **HashMap**: là class thực thi giao diện Map Interface với các key được lưu trữ dưới dạng bảng băm.(có generic)
* **EnumMap**: cũng là 1 Map class nữa, tuy nhiên các key trong Map lại là các enum chứ không phải object như các dạng Map class ở trên.
* **WeakHashMap**: tương tự như HashMap tuy nhiên có 1 điểm khác biệt đáng chú ý là các key trong Map chỉ là các Weak reference (hay Weak key), có nghĩa là khi phần tử sẽ bị xóa khi key được giải phóng hay không còn một biến nào tham chiếu đến key nữa

## Iterator trong java

Cách Iterator hoạt động:



Sự khác biệt giữa cách kiểu dữ liệu cơ bản trong Collection:



**lưu ý:các phần tử trong các kiểu dữ liệu sắp xếp phải so sánh đc nếu ko là ăn ngoại lệ,có thể implement interface comparetator để override hàm compare trong class chứa các object cần so sánh**

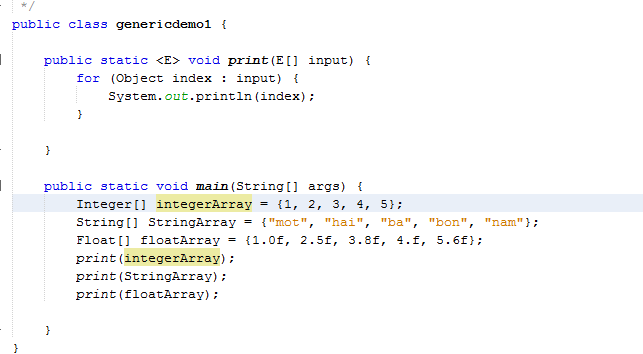
## Generic là gì?

Generics là cách thức lập trình khái quát cho phép một object hoạt động với nhiều kiểu dữ liệu khác nhau.

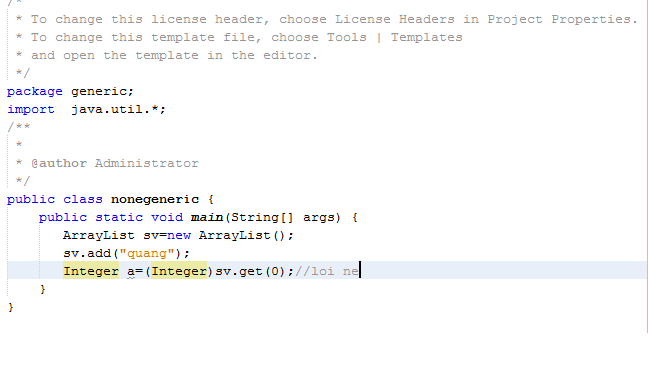
Một ưu điểm mà người ta thường nhắc tới chính là dùng Generic có thể hạn chế được các lỗi trong ép kiểu.

3 ví dụ về cách sử dụng generic.

1.Sử dụng Generic để sử dụng 1 hàm cho nhiều kiểu dữ liệu khác nhau

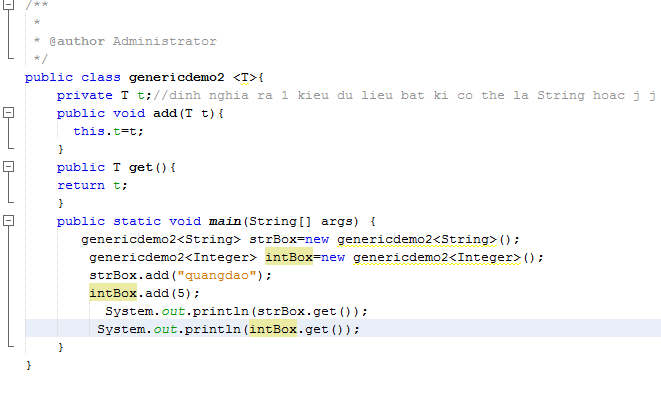


2.Sử dụng generic để hạn chế các lỗi trong ép kiểu



Như trên thì có lỗi cũng ko báo.

Dùng generic định nghĩa lớp chung



# Annotation and Enum

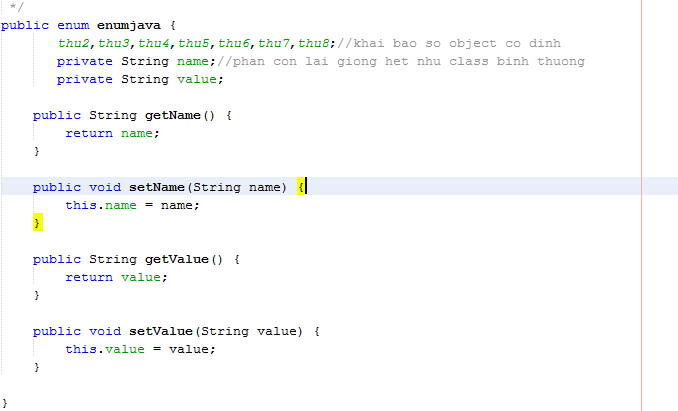
**enum**trong Java là một từ khóa, một tính năng được sử dụng để đại diện cho biến cố định ví dụ:Các ngày trong tuần,các mùa trong năm...

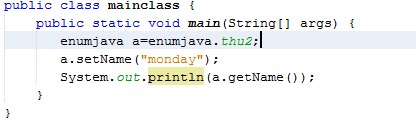
Enum trong Java là 1 class đặc biệt ,từ class đó chỉ tạo được số object cố định,ko thêm xóa được số object nhưng có thể thay đổi được thuộc tính của obj

vd public enum enumjava{

thu2,thu3,thu4;//tương đương với khai báo và khởi tạo các object luôn,ko đc khai báo thêm object nữa

}





Do enum chỉ cho phép khởi tạo số obj từ 1 class cố định => việc sử dụng hàm khởi tạo trong enum cũng hơi khác,khi khai báo :

public enum enumjava {

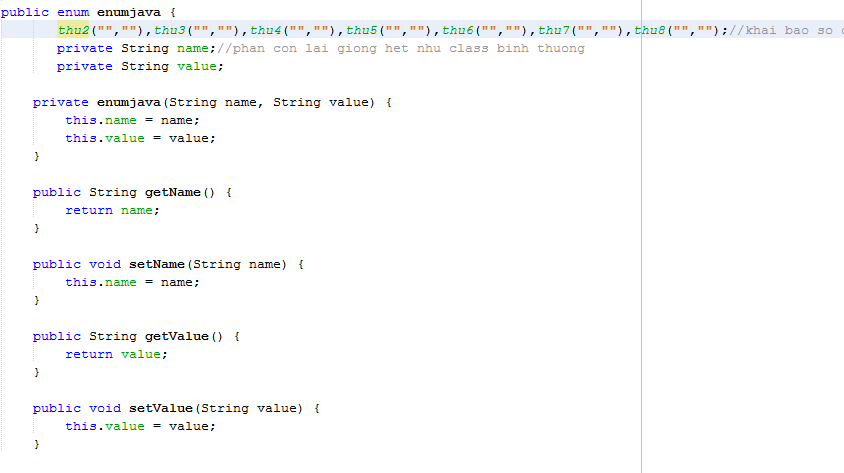
thu2,thu3,thu4,thu5,thu6,thu7,thu8;

Và chưa có hàm khởi tạo thì tương đương với

public enum enumjava {

thu2(),thu3(),thu4(),thu5(),thu6,thu7,thu8;

=>Khi có hàm khởi tạo thì cú pháp sẽ là:



1 tác dụng của enum trong java là có thể sử dụng switch -case:

Trong java chỉ có 2 kiểu có thể sử dụng đc switch-case là int và enum;tất nhiên khi sử dụng enum thì value của các case phải là các obj được khai báo và khởi tạo sẵn.

**Annotation trong java**

Java Annotations (Chú thích) cho phép chúng ta thêm vào thông tin siêu dữ liệu (metadata information) vào mã nguồn của chúng ta, mặc dù nó không phải là một phần của chương trình.

Trong java có 3 Annotations mặc định:@Deprecated, @Override và @SuppressWarnings

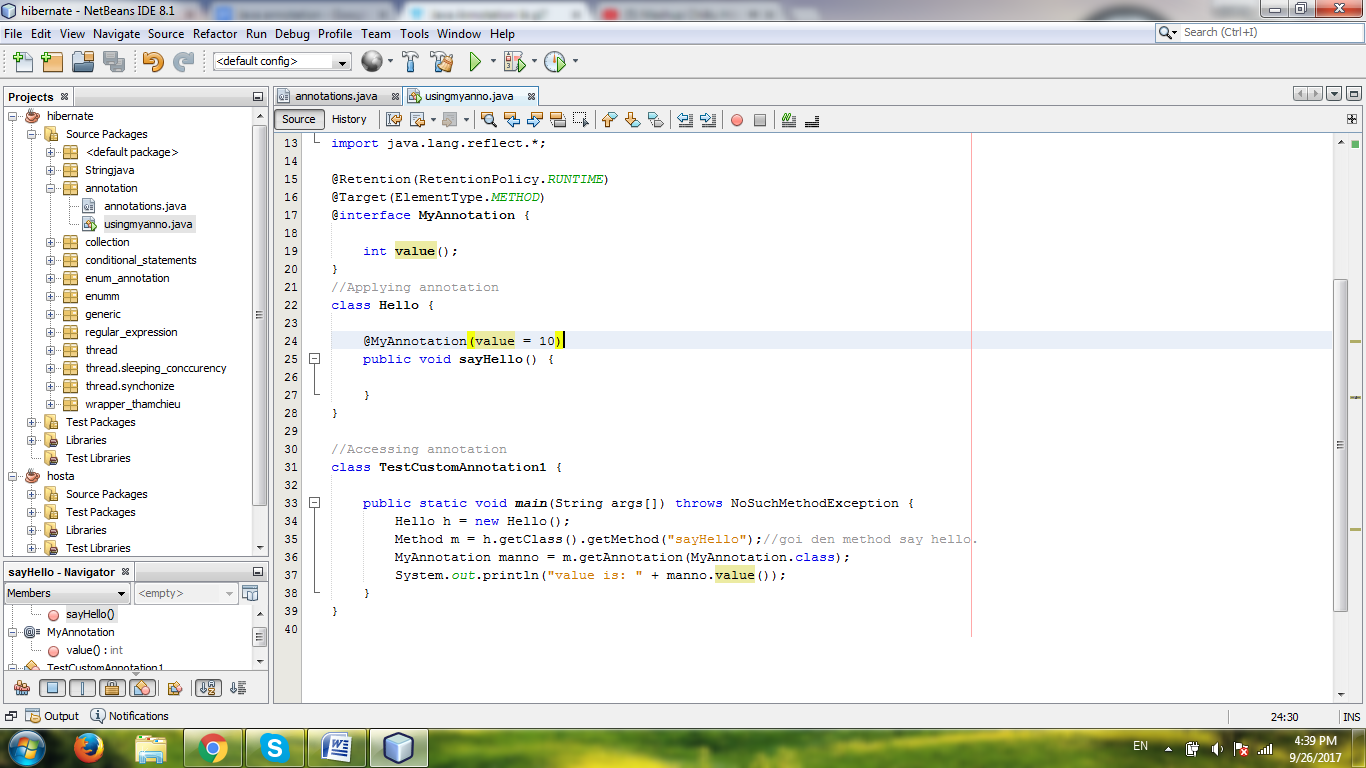
@Deprecated :Đây là một Annotation dùng để chú thích một cái gì đó bị lỗi thời, tốt nhất không nên sử dụng nữa, chẳng hạn như class, hoặc method.

Annotation **@Override** được sử dụng cho các method ghi đè của method trong một class cha (superclass). Nếu method này không hợp lệ với một method trong class cha, trình biên dịch sẽ thông báo cho bạn một lỗi.

Chú thích @SuppressWarnings làm cho các trình biên dịch thôi không cảnh báo một vấn đề của method nào đó. Ví dụ, nếu trong một method có gọi tới một method khác đã lỗi thời, hoặc bên trong method có một ép kiểu không an toàn, trình biên dịch có thể tạo ra một cảnh báo. Bạn có thể tắt các cảnh báo này bằng cách chú thích method này bằng @SuppressWarnings.

//có thể tự tạo các custom annotations:

Đơn giản như sau:



Có thể lấy được giá trị value như trong hình

Có thể nhận thấy Annotations gần giống như 1 interface,cú pháp @interface Annotation\_name{

public int value();

........

}

@Retention & @Target là 2 annotation sẵn có của Java.

**@Retention**: Dùng để chú thích mức độ tồn tại của một annotation nào đó.

Cụ thể có 3 mức nhận thức tồn tại của vật được chú thích:

1. **RetentionPolicy.SOURCE**: Tồn tại trên code nguồn, và không được bộ dịch (compiler) nhận ra.
2. **RetentionPolicy.CLASS**: Mức tồn tại được bộ dịch nhận ra, nhưng không được nhận biết bởi máy ảo tại thời điểm chạy (Runtime).
3. **RetentionPolicyRUNTIME**: Mức tồn tại lớn nhất, được bộ dịch (compiler) nhận biết, và máy ảo thời điểm chạy cũng nhận ra sự tồn tại của nó.

**@Target**: Dùng để chú thích cho một annotation khác, và annotation đó sẽ được sử dụng trong phạm vi nào.

1. **ElementType.TYPE** - Gắn trên khai báo Class, interface, enum, annotation.
2. **ElementType.FIELD** - Gắn trên khai báo trường (field), bao gồm cả các hằng số enum.
3. **ElementType.METHOD** - Gắn trên khai báo method.
4. **ElementType.PARAMETER** - Gắn trên khai báo parameter
5. **ElementType.CONSTRUCTOR** - Gắn trên khai báo cấu tử
6. **ElementType.LOCAL\_VARIABLE** - Gắn trên biến địa phương.
7. **ElementType.ANNOTATION\_TYPE** - Gắn trên khai báo Annotation
8. **ElementType.PACKAGE** - Gắn trên khai báo package.

/\*

Nạp chồng phương thức (method overrloading) xảy ra khi trong cùng một lớp có nhiều hơn hai phương thức có cùng tên, nhưng khác tham số (số lượng hoặc kiểu). Mặt khác, ghi đè phương thức (method overriding) được dùng trong trường hợp một lớp con muốn định nghĩa lại phương thức đã có ở  lớp cha. Phương thức ghi đè (Overridden methods) phải có tên, số lượng tham số, và kiểu trả về giống với lớp cha. Overriding method cũng không giới hạn việc truy cập vào phương thức nó đã ghi đè.\*/

Java cung cấp một interface là Comparable chỉ chứa một phuơng thức compareTo. Phương thức này so sánh 2 đối tuợng, để sắp đặt trật tự giữa chúng. Đặt biệt nó trả về một số dương, 0 hoặc số âm để cho biết đối tuợng nhận vào sẽ nhỏ hơn, bằng hay lớn hơn nó. Java cung cấp interface Comparator chứa 2 phuơng thức là compare và equals. Phuơng thức compare nhận vào 2 tham số và xác định trật tự (order) giữa chúng. Hàm trả về số duơng, 0 hoặc số âm để biểu thị tham số đầu tiên sẽ nhỏ, bằng hay lớn hơn tham số thứ 2. Phương thức equals yêu cầu một đối số là một đối tuợng và giúp xác định đối tuợng truyền vào có phải là một comparator. Hàm trả về true nếu đối tuợng truyền vào là một comparator và đuợc sắp xếp như một comparator.

Hashset k có thứ tự,treeSet có.