Có 2 loại biến: biến nguyên thủy và biến tham chiếu - object refefences

Quy tắc đặt tên biến : …

**Working with Primitive Data Types:**

Có 8 loại dữ liệu nguyên thủy:

- boolean: 1bit

- byte: signed 8-bit integer

- short: signed 16 bit

- char: unsigned 16 bit integer - sử dụng để đại diện 1 ký tự bàn phím

- int: signed 32 bit integer

- long: signed 64 bit integer

- float: signed floating-point number 32 bit

- double: signed floating-point number 64 bit

**Khai báo và khởi tạo biến nguyên thủy**

**Truy cập biến - Accessing Variables**

- Biến dùng để giữ và giao tác giữ liệu. Đối với cả hai tác vụ này, bạn cần truy cập vào biến.

- Một biến chỉ có thể được truy cập trong phạm vi của nó, đó là khu vực trong chương trình mà từ đó nó có thể được gọi một cách hợp pháp. Từ quan điểm của phạm vi, các biến có thể được phân thành ba loại:

- local variables: Các biến dc khai báo trong hàm. Phạm vi của nó là trong hàm đó. Còn dc gọi là stack variables vì dc lưu tại stack

- Instance Variables - biến thể hiện: Khai báo ở trong class nhưng ko nằm trong hàm. Phạm vi của nó là lớp dc khai báo. Được lưu tại bộ nhớ heap. Lưu trữ ở mức Instance.

- Static Variables - biến tĩnh: khai báo trong class với từ khóa static. phạm vi trong class dc khai báo. Lưu trữ ở mức class.

**Cách sử dụng**

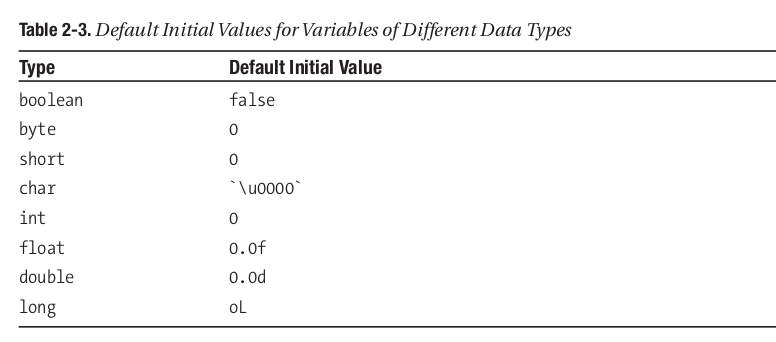
double: chỉ dùng 2 giá trị true hoặc false

char: chỉ lưu giá trị ký tự hoặc giá trị unicode của giá trị đó

int: số nguyên có thể nhập vào ở các hệ cơ số khác nhau: hệ 10, hệ 2, hệ 8, hệ 16

Số phảy động – floating-point:

**Giá trị mặc định:**

****

**Working with Nonprimitive Data Types**

Các dữ liệu không nguyên thủy là các object

Khi khai báo biến với dữ liệu không nguyên thủy, biến đó sẽ trỏ về vùng nhớ của object đó. Vì vậy, nó dc gọi là reference variables or object variable. Object variable có thể lưu trên stack hoặc heap tùy thuộc vào vị trí nó khai báo. Tuy nhiên object luôn dc lưu trữ trên heap. Do biến object chỉ dùng để trỏ vào bộ nhớ chứa object chứ ko thực sự là object.

**Object**

**Arrays**

**Enum**

**Understanding Operations on Data**

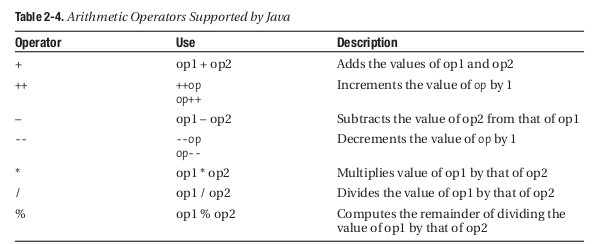
Dựa vào số lương toán hạng, chia các phép toán thành:

- phép toán 1 ngôi: như ++, –

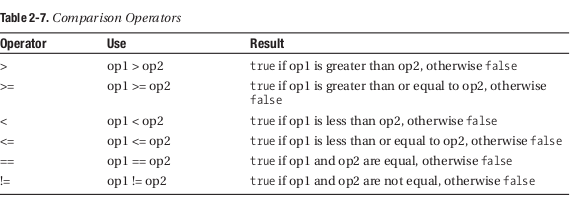
- phép toans 2 ngôi

- phép toán 3 ngôi: ?:

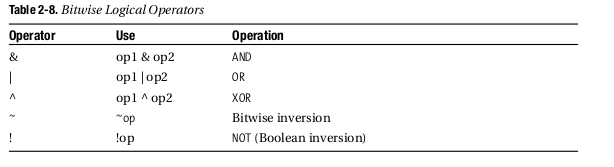
**Phép toán số học:**



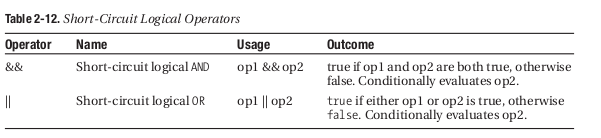
**Phép toán so sánh / điều kiện**



**Phép toán logic**



**Short-Circuit Logical Operators**



**STRING**

**String Class** extends CharSequence

**Constructor**

Có 2 cách khởi tạo một chuỗi:

-Khởi tạo bằng toán tử new:

string str = new String(“abc”)

hoặc

char[] chars = {‘a’,’b’,’c’}

string str = new String(chars)

Khí khởi tạo bằng cách này, sẽ có 2 object dc tạo, trong đó có 1 string object và biến str sẽ trỏ vào string object này

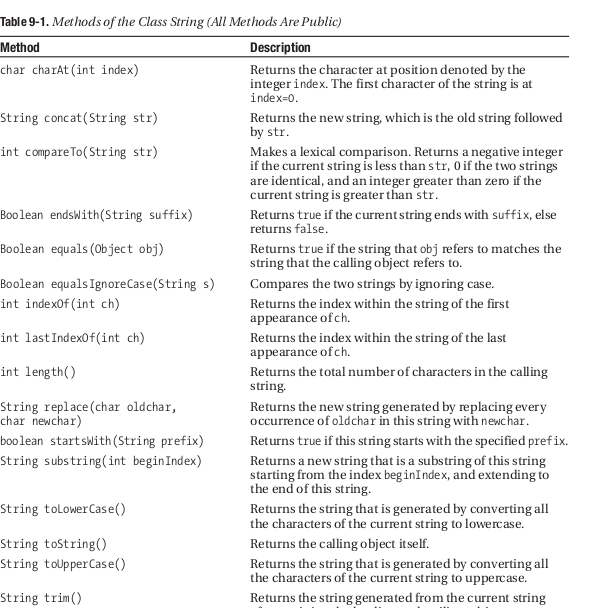
- Không dùng toán tử new

String str = “abc”

Chỉ có 1 object được tạo.

Khi khao báo theo cách trên, JVM sẽ tìm trong bộ nhớ heap String “abc” và gán biến str vào bộ nhớ đó.

**Method String**

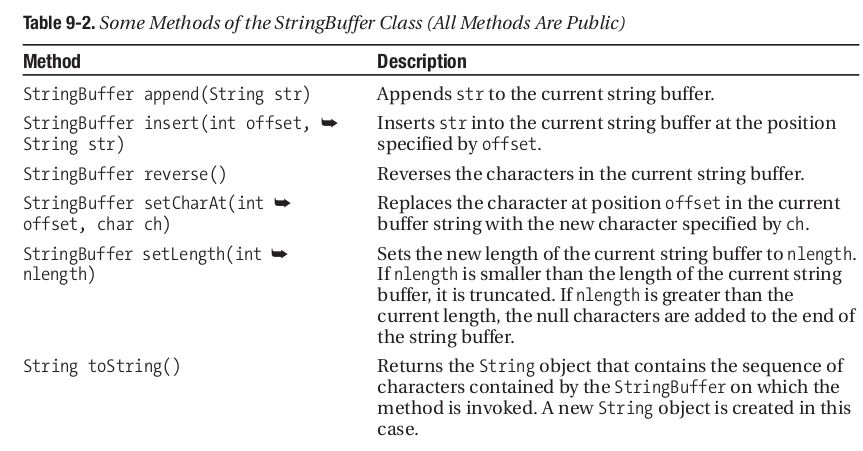


**StringBuffer Class**

StringBuffer được sử dụng khi bạn cần sửa mọto chuỗi nhiều lần. Nó cho phép các method chỉnh sửa chuỗi ngay trên chính object đó.

Constructor giống như String

**StringBuffer Method**



**StringBuilder Class**

Tương tự StringBuffer, StringBuilder tạo ra object string có thể thay đổi được. Điểm khác biệt giữa StringBuffer và StringBuilder là StringBuffer đồng bộ, còn StringBuilder không đồng bộ. Ngoài ra, StringBuilder hiệu quả hơn StringBuffer.

Đồng bộ là không thể có 2 luồng cùng truy cập vào method của nó đồng thời.

Không đồng bộ thì ngược lại.

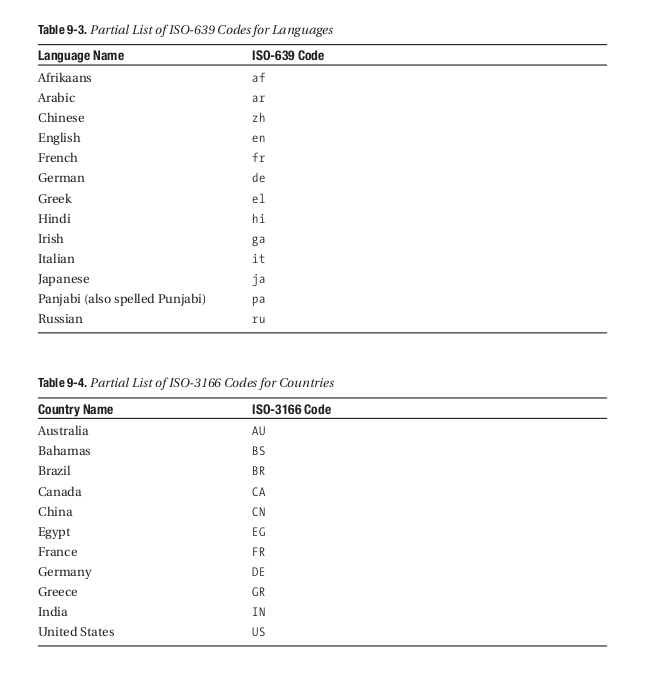
**Formatting and Parsing for the World**

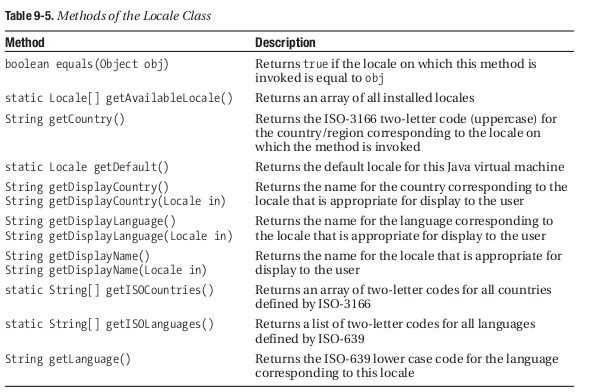
**Locale Class**

Giúp hiển thị thông tin cụ thể theo từng miền địa phương mà không cần sử dụng mà không phải sử dụng hard code.

Constructor: Locale(String language) || Locale(String language, String country)

Tham số language là mã ngôn ngữ được chỉ định bởi ISO-639, country là mã quốc gia dc chỉ định bới ISO-3166.

Bằng cách truyền vào 1 hoặc 2 tham số, ta sẽ tạo ra đối tượng về một miền địa phương. Sau đó có thể sử dụng đối tượng này để lấy thông tin về địa phương đó.

****

**Formatting Numbers and Currencies**

**NumberFormat Class** giúp ta có thể biến đổi số cho bất kỳ địa phương nào. Nó cung cấp một số phương thức thuận tiện để tạo đối tượng NumberFormater tương ứng với địa phương mặc định hoặc cụ thể

Ví dụ:

NumberFormat.getInstance();

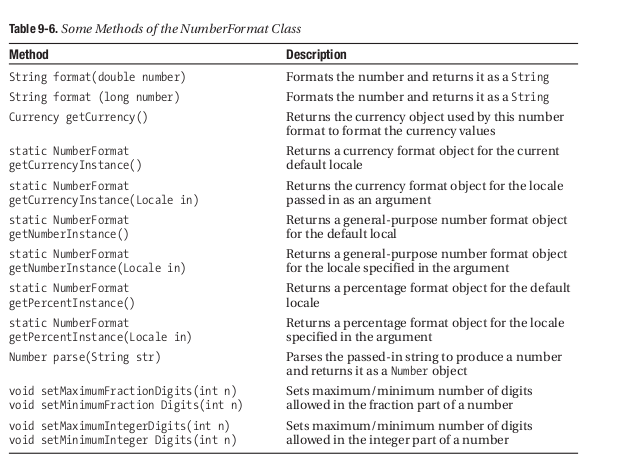
NumberFormat.getCurrencyInstance();

NumberFormat.getPercentInstance();

NumberFormat.getInstance(myLocale);

NumberFormat.getCurrencyInstance(myLocale);

NumberFormat.getPercentInstance(myLocale);

**Các bước tạo fomater đặc biệt:**

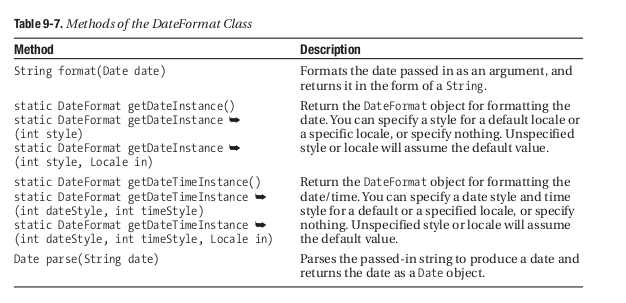
1. Tạo object địa phương Locale

2. Tạo NumberFormater thích hợp với locale tạo ở bước 1

3. Format số bằng phương thức format của NumberFormat

**Formatting Dates**

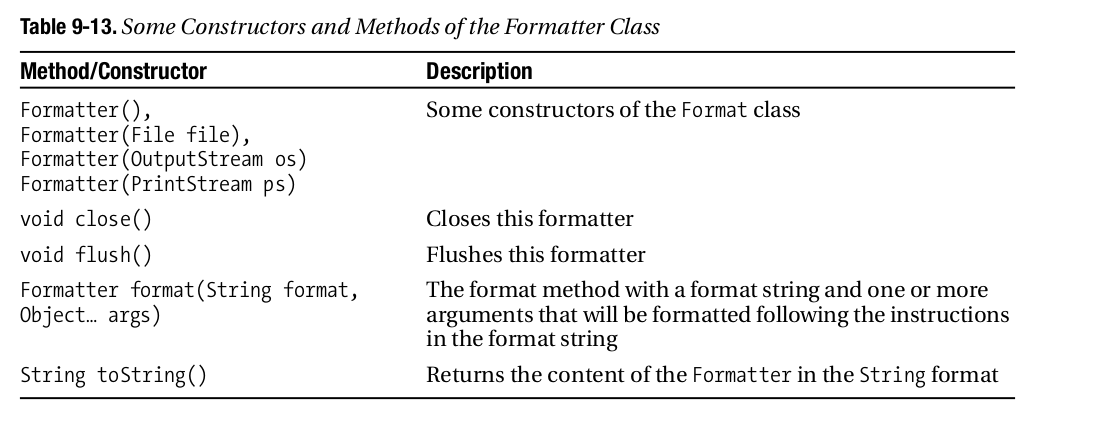
DateFormat cung cấp các phương thức để định dạng ngày tháng theo các địa phương khác nhau.

**Formatting and Parsing Streams**

Định dạng và phân tích cú pháp như 2 mặt của một đồng tiền. Ví dụ, ở bên gửi, ứng dụng gửi một dữ liệu ở một định dạng nhất định, và ở phía nhận, nó sẽ phân tích thành thông tin hữu ích.

**Formatting Streams**

Lớp Formatter có phương thức format(). Ta có thẻ dùng Formater để định dạng dữ liệu và xuất ra các thiết bị I/O.

Phương thức forrmat() có định dạng sau đây:

format(<format specifier>, <argument>)

argument là data chúng ta cần định dạng

format specifier cung cấp cách để ta định dạng dữ liệu, nó có dạng như sau:

%[<argumentIndex>$][<flags>][<width>][.<precision>] <type>

Mô tả:

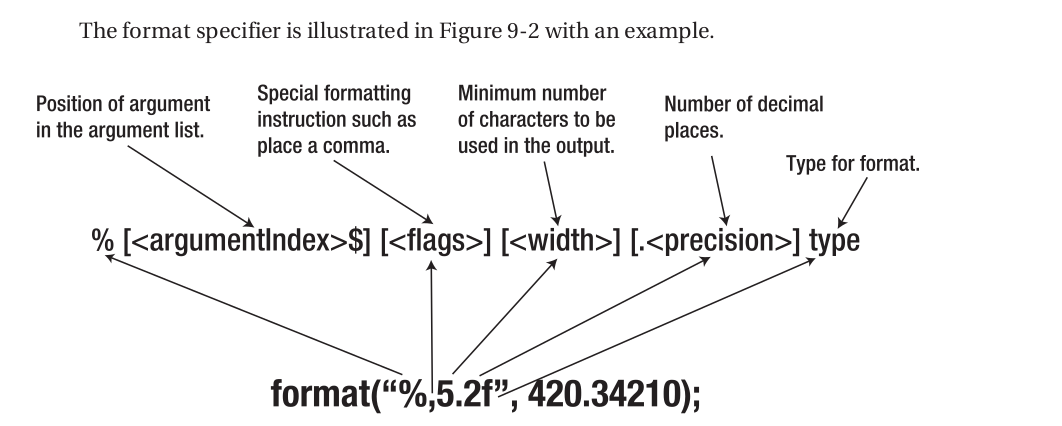
<argumentIndex>: chỉ định vị trí của đối số trong danh sách đối số trong trường hợp có nhiều hơn một đối số. Nếu chỉ có một đối số, bạn không cần chỉ định phần tử này.

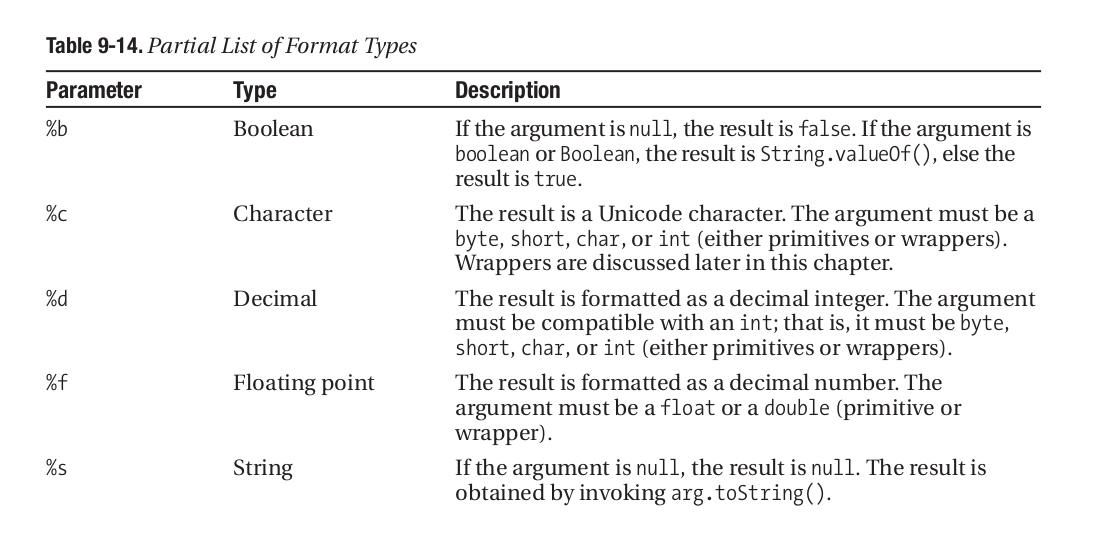
<flags>: Là một tập hợp các ký tự xác định các định dạng đặc biệt của đầu ra. Có thể bỏ qua nếu không cần thêm thông số kỹ thuật.

<width>: Chỉ định số lượng ký tự tối thiểu được ghi vào đầu ra. Hãy nhớ rằng đó là số lượng tối thiểu. Ví dụ: nếu một số có chiều rộng bảy ký tự và <width> có giá trị 5, toàn bộ số vẫn sẽ được ghi.

<precision>: Chỉ định độ chính xác về số lượng vị trí thập phân. Ví dụ: giá trị float 23.4321 sẽ được chuyển đổi thành 23.43 với giá trị <precision> 2.

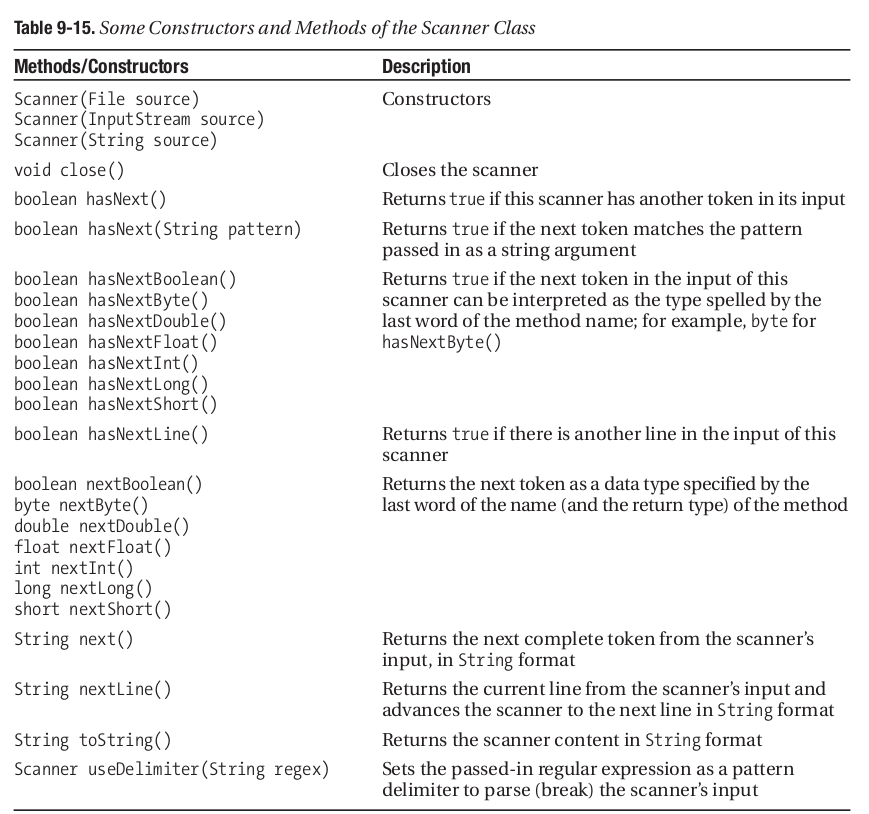
<type>: Chỉ định loại định dạng liên quan đến loại dữ liệu, chẳng hạn như int, double và float. Các loại này được liệt kê trong Bảng 9-14. Lưu ý rằng một loại định dạng có liên quan đến loại đối số.

****

**Parsing Streams**

Văn bản đầu vào cần được phân tích cú pháp có thể được chuyển đến hàm tạo của Trình quét dưới dạng Chuỗi, Tệp hoặc InputStream. Các mã thông báo riêng lẻ có thể được chuyển đổi thành các giá trị thuộc các loại khác nhau bằng cách sử dụng phương pháp tiếp theo phù hợp

Constructor:



**Wrapping the Primitives**

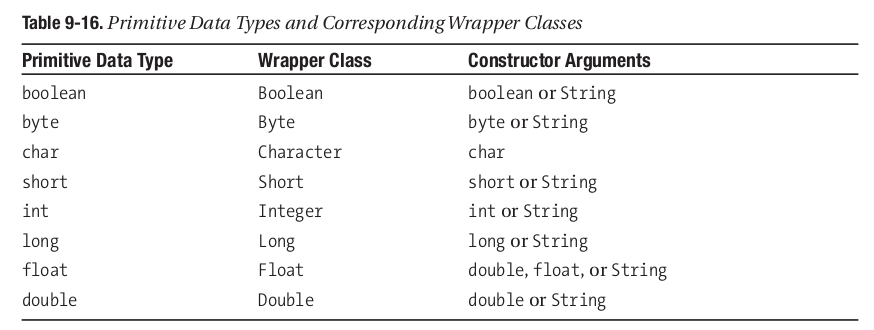
Java là một ngôn ngữ hướng đối tượng và như họ nói, mọi thứ trong Java là một đối tượng. Nhưng, còn dữ liệu nguyên thủy thì sao? Dữ liệu nguyên thủy là loại bỏ trong thế giới của các đối tượng; nghĩa là, chúng không thể tham gia vào các hoạt động của đối tượng, chẳng hạn như được trả về từ một phương thức như một đối tượng và được thêm vào Collection các đối tượng (Collection là một nhóm các mục dữ liệu, được thảo luận thêm trong Chương 10). Là một giải pháp cho vấn đề này, Java cho phép bạn bao gồm các dữ liệu nguyên thủy trong các đối tượng bằng cách sử dụng cái được gọi là các Wrapper class.

**Creating Objects of Wrapper Classes**

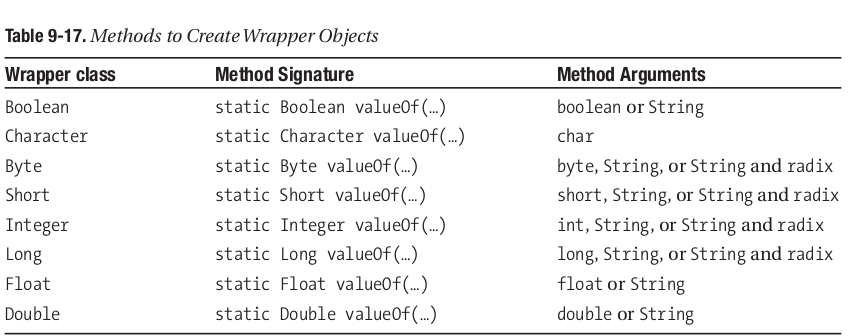
Tương ứng với mỗi kiểu dữ liệu nguyên thủy trong Java là một lớp được gọi là lớp bao bọc. Lớp này gói gọn một giá trị cho kiểu dữ liệu nguyên thủy. Ví dụ, lớp Integer sẽ giữ một giá trị nguyên. Đối tượng Wrapper của lớp trình bao bọc có thể được tạo theo một trong hai cách: bằng cách khởi tạo lớp bao bọc với toán tử new hoặc bằng cách gọi một phương thức tĩnh trên lớp bao bọc. Chúng ta khám phá điều này tiếp theo.

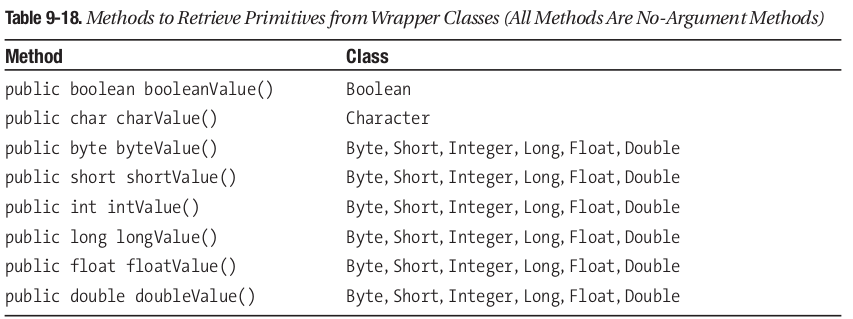
Sử dụng toán tử new:

Tên lớp wrapper tương ứng với các dữ liệu và các kiểu đối số nó có thể nhận:

Sử dụng static method:

Ta có thể sử dụng phương thức valueOf() của tất cả wrapper để tạo ra đối tượng tương ứng:

**Methods to Extract the Wrapped Values**

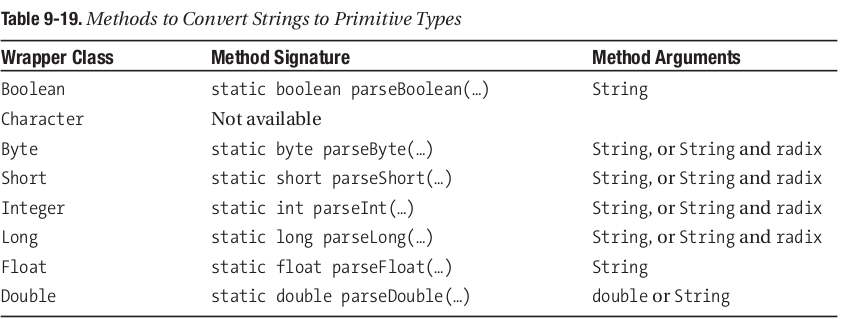
**The Instant Use of Wrapper Classes**

Nếu bạn không cần lưu trữ một giá trị trong trình bao bọc mà chỉ muốn thực hiện thao tác nhanh trên nó, chẳng hạn như chuyển đổi loại, bạn có thể thực hiện bằng cách sử dụng một phương thức tĩnh thích hợp của lớp trình bao bọc thích hợp. Ví dụ, tất cả các lớp trình bao bọc trừ Ký tự cung cấp một phương thức tĩnh có cú pháp sau:

static <type> parse<Type>(String s)

<type> là các kiểu dữ liệu nguyên thủy trừ character, <Type> giống như <type> với chữ cái đầu viết hoa.

Ví dụ: static int parseInt(“123”);

****

**The Mother of All Classes: Object**

Mọi thứ trong Java, ngoại trừ nguyên thủy, là một đối tượng. Như bạn đã biết từ chương trước, bạn có thể làm cho ngay cả những người nguyên thủy cũng hành xử giống như các đối tượng bằng cách sử dụng các Wrapper class. Java chung cấp class Object mà từ đó các lớp khác được dẫn xuất ngầm định hoặc rõ ràng.

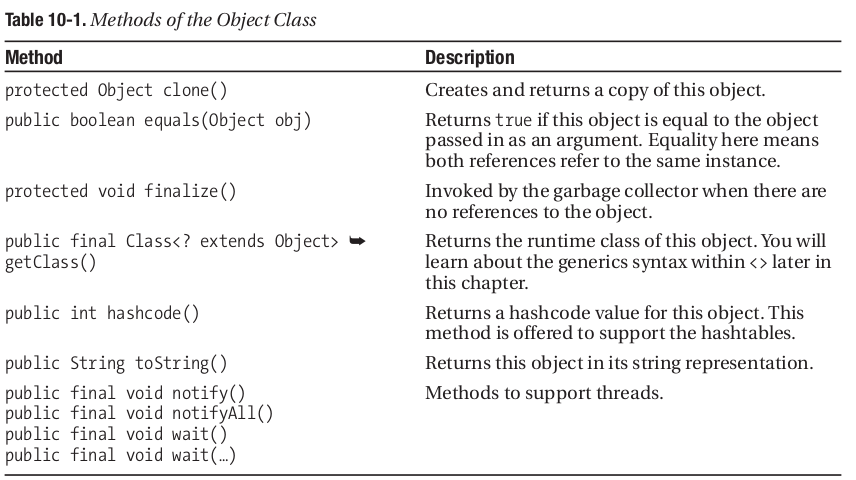
**Object Class**

Java hỗ trợ kế thừa đơn; đó là một lớp không thể là một lớp con của nhiều hơn một lớp. Điều đó có nghĩa là cây phân cấp lớp trong Java phải có một gốc được đại diện bởi một lớp duy nhất. Lớp Object là gốc mà từ đó tất cả các lớp khác được dẫn xuất trực tiếp hoặc gián tiếp. Kết quả là, các lớp mà Java cung cấp, chẳng hạn như các lớp trong các gói khác nhau của nó và mọi lớp bạn viết sẽ tự động kế thừa các phương thức của lớp Object ngay cả khi bạn không mở rộng Object trong khai báo lớp.

Tác dụng của Object class:

- Object class có thể dùng để khai báo trước một đối tượng khi mà ta chưa chắc chắn đó là đối tượng nào. Sau từng tình huống cụ thể, ta có thể ép kiểu vầ class tương ứng.

- Đây llaf lớp cơ sở, cung cấp những giá trị chung nhất cung cấp cho các lớp kế thừa nó, các phương thức chung là class Object cung cấp:

****

**The equals() Method**

Nếu mọi thứ trong Java là một đối tượng, bạn sẽ có thể xác định xem hai đối tượng có bằng nhau không. Kiểm tra đẳng thức là một hoạt động quan trọng trong lập trình. Ví dụ, một số cấu trúc dữ liệu không cho phép lưu trữ trùng lặp.

Tuy nhiên, phép so sáng == không giúp so sánh 2 đối tượng có giống nhau hay không mà chỉ cho biết 2 biến x và y có trỏ về cùng một object hay không.

Để tạo điều kiện cho kiểm tra đẳng thức, lớp Object cung cấp phương thức equals (…) như sau:

public boolean equals(Object obj)

Việc thực hiện phương thức equals (...) trong lớp Object rất mơ hồ; nó chỉ sử dụng toán tử == để so sánh. Ví dụ, hãy xem xét hai tham chiếu đối tượng x và y. Hàm x.equals (y) sẽ chỉ trả về true nếu x và y tham chiếu đến cùng một đối tượng. Tuy nhiên, bạn có thể ghi đè phương thức này trong lớp mà bạn viết và cung cấp cho nó một ý nghĩa sâu sắc hơn.

Việc triển khai hợp lệ phương thức equals(...) phải tôn trọng nguyên tắc sau:

- Đây là hàm phản xạ: đối với mọi tham chiếu x không null, x.equals (x) sẽ trả về true.

- Đây là phương thức đối xứng: với 2 thâm chiếu x và y không null, x.equals(y) trả về true khi và chỉ khi y.equals(x) cũng trả về true

- Phương thức có tính chất bắc cầu: với x,y,z không null. Nếu x.equals(y) true, y.equals(z) true thì x.equals(z) cũng phải true

- Phương thức có tính bất biến: với x,y không null, sau nhiều lần sử dụng, x.equals(y) luôn cùng trả về một giá trị. hàm này ko dc thay đổi bất kỳ thuộc tính nà của x và y.

- Có thể sử dụng với tham chiếu null.

**The toString() Method**

Trong Object class, hàm toString() sẽ trả về <Tên class>@<Hashcode in hex>

Tuy nhiên, ta có thể ghi đè hàm này ở các class cụ thể để tùy biến phù hợp hơn. Ví dụ, hàm toString của class String sẽ trả về chính đối tượng gọi nó.

**Understanding Collections**

Khi viết một ứng dụng, bạn thường xử lý nhiều mục dữ liệu liên quan đến nhau, chẳng hạn như các phòng ban trong một công ty, nhân viên trong một bộ phận, v.v. Để đối phó với một nhóm các mục dữ liệu, Java cung cấp các Collections Collection là một đối tượng tổ chức nhiều mục dữ liệu thành một nhóm. Như đã nói trước đó, trong một vấn đề trong thế giới thực, các Collection thường đại diện cho nhóm vật phẩm dữ liệu có liên quan với nhau. Collection được sử dụng để thao tác dữ liệu (thực hiện các thao tác trên nó), lưu trữ và truy xuất dữ liệu và truyền dữ liệu giữa các phương thức.

Trong khoa học máy tính, chúng tôi định nghĩa một kiểu dữ liệu trừu tượng (ADT) là một tập hợp các đối tượng bao gồm tappj hợp các phương thức có thể thực thi ở đối tượng đó. Java đưa ra một mô hình hợp nhất gọi là Collections framwork tạo ra một Collection như một implementation của một ADT. Collections framwork bao gồm 3 yếu tố:

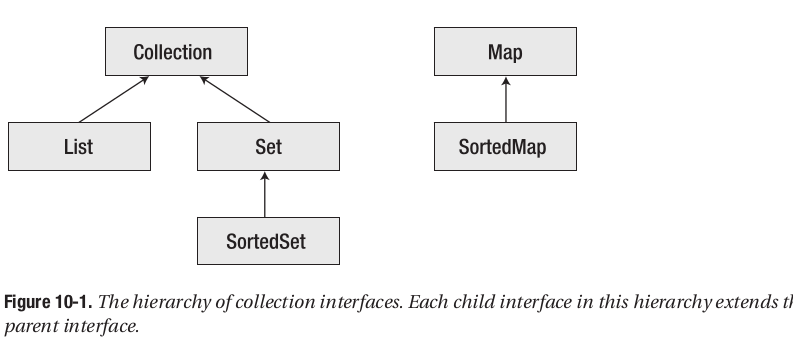
- Interface: Cho phép sử dụng collections với các thao tác triển khai đọc lập

- Implementation: việc triển khai cụ thể các phương thức của Collections Interface

- Algorithms: Các phương thức thực hiện các hoạt động như tìm kiếm và sắp xếp trên các đối tượng trong cấu trúc dữ liệu.

**The Collections Interfaces**

Gốc - cha của hệ thống phân cấp cảu collections interfaces là Collection interface, được gọi là super-interface cả collections. Ngoài ra còn một kiểu collections khác là map với super-interface là Map. Map không extends Collection.

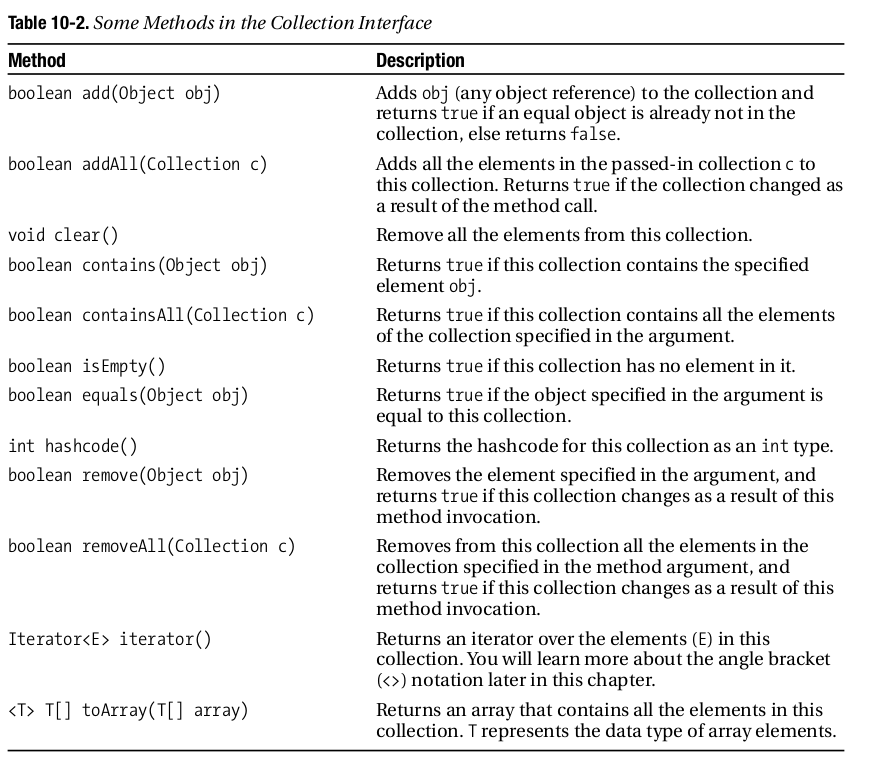
The List and Set interfaces extend từ Collection interface, không implementation trực tiếp từ Collection interface. Đặc điểm của List và Set và Map:

- List: An ordered collection of data items; Bận biết chính xác vị trí các item trong list. Có thể chứa các item trùng lặp. ArrayList, LinkedList, and Vector are the classes that implement the List interface.

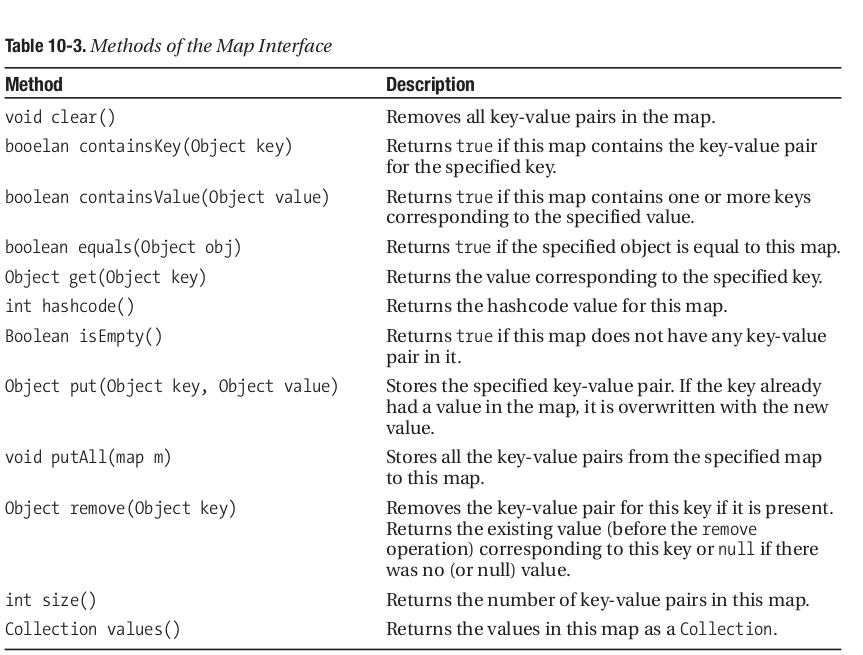
- Map: một object được có key và value được map với nhau. Mỗi key được ánh xạ đến một value. Các key không thể trùng lặp nhưng các value có thể trùng nhau. HashMap và HashTable là cá class implement Map interface.

- Set: là một Collection chưa các item không trình nhau. Các lớp implement từ Set interface là HashSet và LinkedHashSet.

Một số phương thức của Collection interface:

****

Phương thức của Map interface:

****

**Implementations of Collections Interfaces**

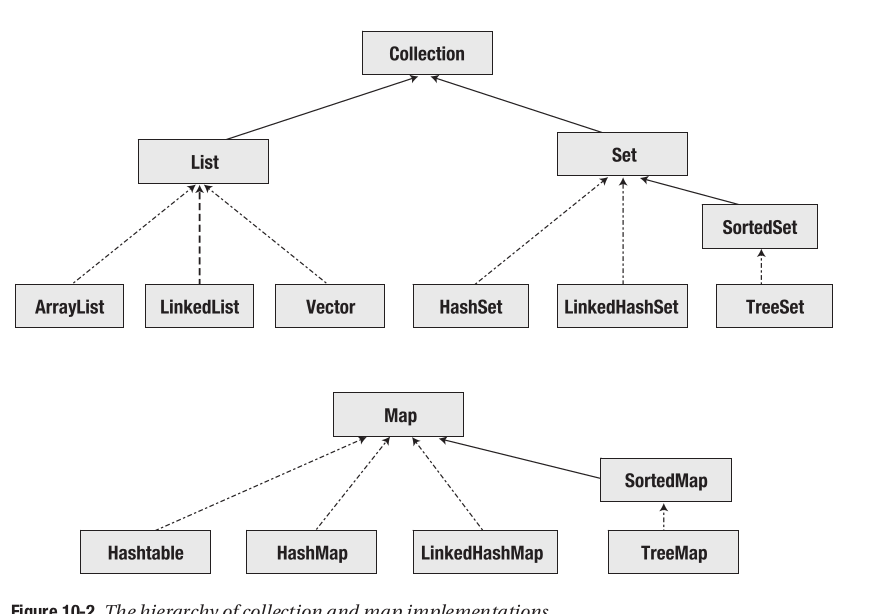
Implementations of Collections Interfaces là các class implement các interfaces. Mục đích của bộ sưu tập là nhóm các object có liên quan đến nhau. Các object có thể nhóm theo nhiều các khác nhau tùy thuộc vào yêu cầu ứng dụng. Đó là lý do tại sao chúng ta có một số ADT và cấu trúc dữ liệu tương ứng và đó cũng là lý do Java cung cấp một số giao diện bộ sưu tập và một số triển khai của một giao diện nhất định.

Các đặc điểm của implementations:

- Performance: là thời gian thực hiện bởi thao tác cụ thể được thực hiện trên dữ liệu. Ví dụ như tìm kiếm, chèn, xóa. Tìm kiếm, chèn, xóa thường có hiệu suất đối lập. Nếu tìm kiếm nhanh thì việc chèn xóa có thể chậm và ngược lại.

- Ordered/ Sorted (Đánh chỉ số/ sắp xếp): Một cấu trúc dữ liệu dc gọi là ordered nếu các dữ liệu được đánh chỉ mục theo một cách nào đó. nghĩa là ta có thể truy xuất đến dữ liệu theo chỉ mục của chúng. Cấu trúc dữ liệu dc gọi là sorted nếu nó đc sắp xếp theo bản chất của cá dối tượng, như sự tăng dần của giá trị. Vì vậy, theo định nghĩa, cấu trúc dữ liệu được sắp xếp là cấu trúc dữ liệu theo thứ tự, nhưng điều ngược lại không nhất thiết là đúng

- Synchronized (Đồng bộ hóa): Một số implementations được đồng bộ hóa, chúng có thể hỗ trowj làm việc đa luồng an toàn. Các implementation khác sẽ ko được hỗ trợ. Tuy nhiên, đây không phải là một yếu tố lớn trong việc chọn triển khai, bởi vì ngay cả khi triển khai bạn chọn không đồng bộ hóa, bạn luôn có thể sử dụng các phương thức đồng bộ hóa từ lớp Bộ sưu tập.



**Implementation of List interface**

- ArrayList: ArrayList giống như một mảng có thể tăng số lượng phần tử. Giống như mảng, các phép truy cập vào item cụ thể trong list với thời gian cố định O(1) nhưng các phép chèn và xóa với thời gian tuyến tính O(n). Vì vậy sử dụng ArrayList khi có nhiều phép truy cập ngẫu nhiên phân tử mảng nhưng ít thao tác chèn hoặc xóa.

- Vector: giống như ArrayList nhưng hỗ trợ đa luồng.

**Implementations the Set Interface**

**-** HashSet: cung cấp cách truy cập phần tử ngẫu nhiên nhanh hơn TreeSet. Tuy nhiên không đảm bảo các phần tử được đánh chỉ số và không hỗ trợ đa luồng.

- TreeSet: các phần tử được sắp xếp nhưng hiều suất tru cập trong nhanh bằng HashSet. Không hỗ trợ đồng bộ hóa.

- LinkedHashset: LinkedHashset giống như một Hashset duy trì một danh sách liên kết đôi, chạy qua tất cả các mục dữ liệu. Nó được đánh chỉ số bằng cách chèn nhưng không dc sắp xếp. Không hỗ trợ đồng bộ hóa đa luồng.

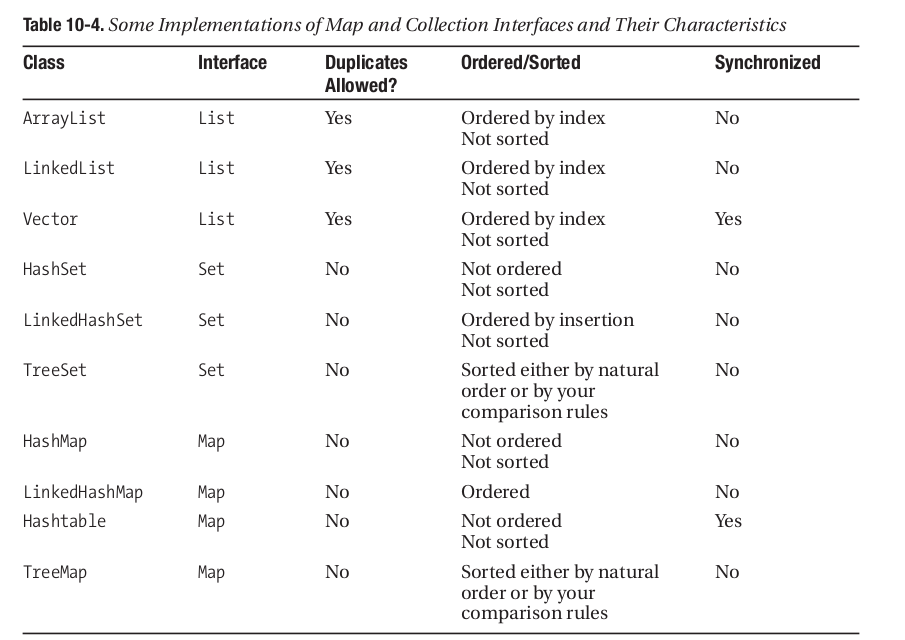
**Implementations the Map Interface**

Nó không đảm bảo các dữ liệu dc đánh chỉ số, các đối tượng làm khóa phải có phương thức hashCode() và equals() để có thể truy xuất thành công các đối tượng từ hàm băm. Nó hỗ trợ đồng bộ hóa đa luồng.

- HashMap: mplementation này dựa trên cấu trúc dữ liệu có thể băm, có thể dùng object null, không được đnahs chỉ số, không hỗ trợ đồng bộ hóa.

- LinkedHashMap: Việc triển khai này khác với HashMap ở chỗ nó duy trì một danh sách liên kết gấp đôi chạy qua tất cả các mục của nó. Hơn nữa, nó xác định thứ tự, thường là thứ tự của các keys được chèn vào Map. Sử dụng danh sách này nếu thứ tự không xác định được cung cấp bởi Hashtable và HashMap không phù hợp với ứng dụng của bạn

- SortedMap: TreeMap thực hiện giao diện SortedMap. Nó đảm bảo rằng bản đồ sẽ theo thứ tự khóa tăng dần (nghĩa là đã được sắp xếp). Việc thực hiện này không đồng bộ.



**The hashCode() method**

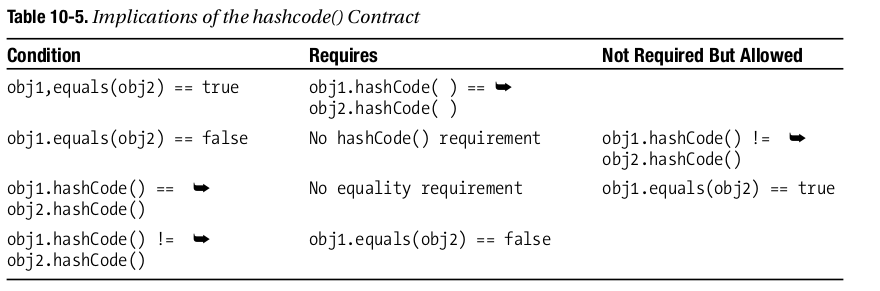
Mã băm là một giá trị nguyên được gắn vào một đối tượng, một mặt, chỉ định tính duy nhất của một đối tượng và mặt khác, giúp lưu trữ một đối tượng trong cấu trúc dữ liệu và truy xuất nó. Để tính toán mã băm cho một đối tượng, lớp Object cung cấp phương thức hashCode(), có cú pháp sau: public int hashCode()

Ta có thể ghi đè hàm hashCode nhưng phải tuân theo nguyên tắc sau:

- Hàm này không thay đổi các giá trị của Object, theo đó, sau nhiều lần chạy, hashCode của một object không thay đổi.

- Nếu 2 object bằng nhau qua phương thức equals() thì hashCode của nó phải giống nhau. Nhưng 2 object không bằng nhau có thể có hashCode giống nhau

- Hai object khác hashCode thì nó phải khác nhau qua phương thức equals.



**Understanding Generics**

Khả năng sử dụng lại mã là một đặc tính quan trọng của bất kỳ ngôn ngữ hướng đối tượng nào. Ví dụ, kế thừa lớp cung cấp khả năng sử dụng lại của một lớp. J2SE 5.0 đưa khả năng sử dụng lại lên cấp độ tiếp theo bằng cách giới thiệu một tính năng rất mạnh gọi là Generics Programing, cho phép bạn viết mã có thể được sử dụng lại cho các loại đối tượng khác nhau. Trong phần này, chúng tôi khám phá lập trình chung từ quan điểm của các Collections.

**Generic Collections**

Khi truy xuất một đối tượng từ Collection, ta cần ép kiểu chúng về đúng loại. Với cách làm này, bạn có thể sẽ ko gặp lỗi biên dịch nhưng sẽ có thể gặp lỗi khi chạy. Chúng ta luôn muốn biết được lỗi ngay khi dịch chứ không phải khi chạy. Giải pháp cho vấn đề này được cung cấp trong J2SE 5.0 giúp ta khai báo kiểu dữ liệu nằm trong bộ sưu tập, và trình biên dịch có thể kiểm tra dữ liệu ta truy xuất / chèn vào có đúng kiểu hay không.

Ngoài ra, việc khai báo này sẽ tránh việc trền các kiểu dữ liệu khác nhau vào collection.

**Generic Programming**

Câu trả lời là lớp ArrayList đã được khai báo trong gói java.util theo cách mà bạn có thể đề cập đến kiểu dữ liệu của nó cùng với tên: public class ArrayList<E> với<E> là bất kỳ class nào.

Để làm quen với generic programming, ta cần làm quen với các ký hiệu <E>, kí tự đại diện. Các ví dụ về generic programming:

public interface List<E> extends Collection<E>

Có nghĩa là List extends từ collection và List là generic collection, nó có thể là List của <E> với E là bất kỳ kiểu đối tượng nào.

<T> T[] toArray(T[] a)

hàm trên sẽ trả về một mảng chwuas các object của collection, Kiểu dữ liệu trả về của nó chưa biết trước, có thẻe là một kiểu <T> bất kỳ.

HashSet(Collection<? extends E> c)

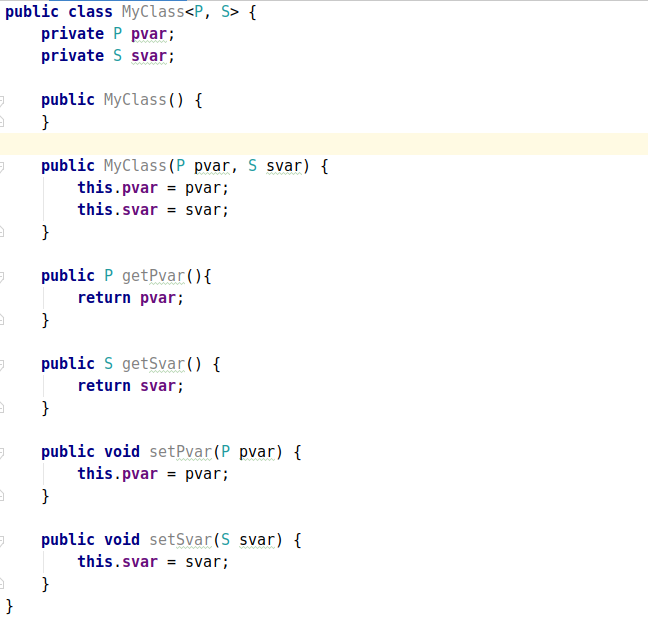
Contructor tren sẽ tạo ra HashSet object từ một Collection nào đó. Collection<? extends E> có nghĩa là object của collection phải có kiểu tương thích với <E>.

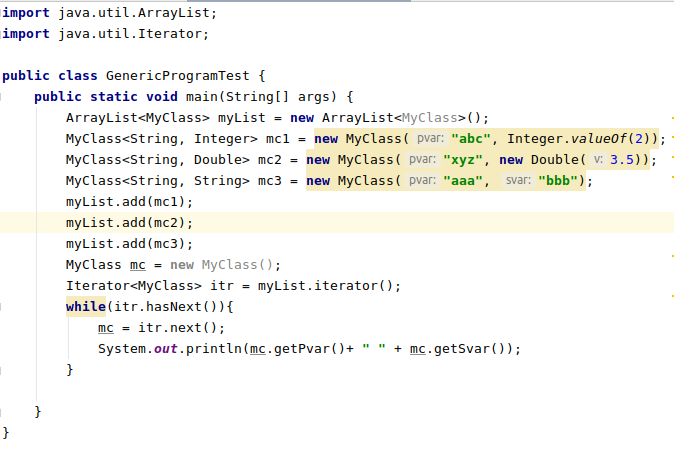
Ta cũng có thể tạo ra lớp Generic với 1 hoặc nhiều generic type:

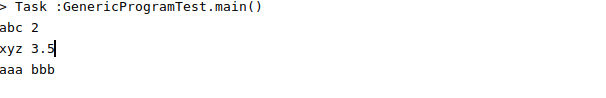
public Class<P, S> (){

Bạn có thể sử dụng P, S type trong class và không cần biết chính xác nó là gì. Kiểu này sẽ dc định nghĩa khi ta khai báo về sau.

}



Output:



**Understanding Autoboxing**

Các kiểu dữ liệu nguyên thủy không thể tham gia vào các hoạt động hướng đối tượng. Để giải quyết vấn đề này, ta dùng các lớp Wrapper để bọc giá trị nguyên thủy vào, quá trình này gọi là boxing, khi cần lấy giá trị nguyên thủy, ta dùng các hàm – unboxing. Nếu có rất nhiều quá trình boxing và unboxing diễn ra, chương trình trông sẽ lọno xộn. Vì vậy J2SE 5.0 đưa ra một cách tự động autoboxing và autounboxing.

Ví dụ nếu code bình thường:

public Double areaOfASquare(Double side){

double d = side.doubleValue();

double a = d\*d;

return new Double(a);

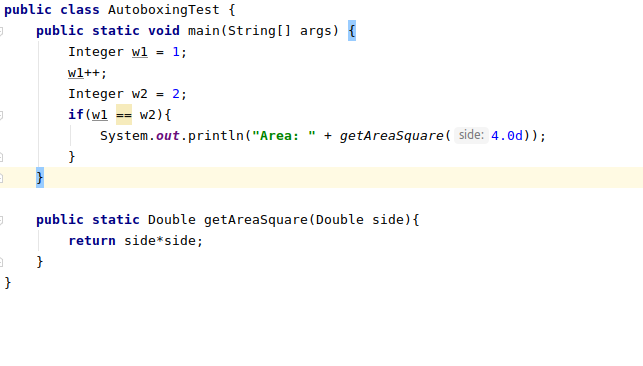
}

Sử dụng autoboxing:

public Double areaOfASquare(Double side){

return side\*side;

}



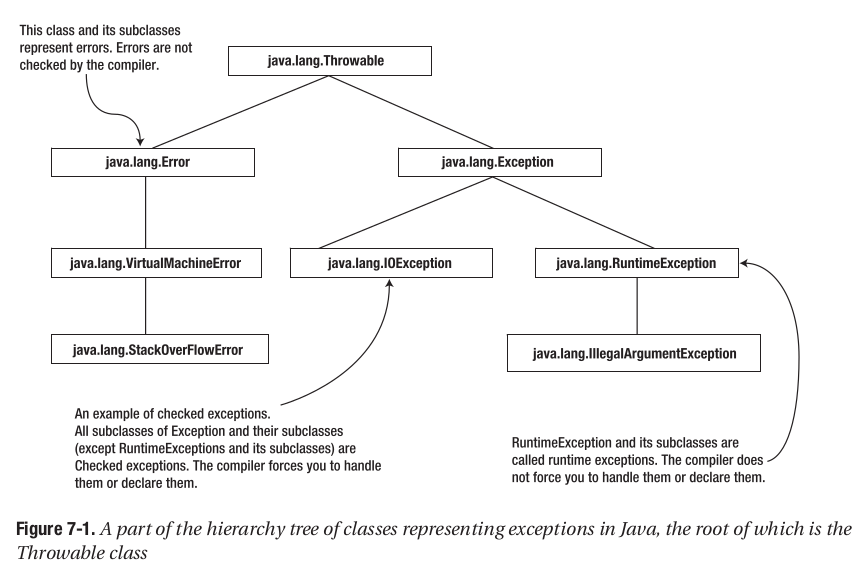
Output: 16.0

**Exceptions in Java**

**Understanding Exception in Java**

Exception là lỗi xảy ra khi chương trình chạy. Khi exception xảy ra, exception sẽ gọi là bị bắt, nó sẽ thay đổi luồng chạy của chương trình.

**Exception Tree**



Lớp throwable, là gốc của exception tree, là lớp cơ sở của tất cả các ngoại lệ trong Java. Hai lớp con của throwable, có nguồn gốc trực tiếp từ throwable, là Error và Exception.

Error (được biểu thị bằng các lớp con LỗiError xảy ra trong máy ảo Java (JVM) và không phải trong chính ứng dụng. Một lỗi thường đại diện cho một vấn đề bất thường mà từ đó rất khó để phục hồi. Những vấn đề này thường bắt nguồn từ môi trường thực thi của ứng dụng (ví dụ: hết bộ nhớ). Vì không thể (hoặc khó nhất là tốt nhất) để phục hồi từ các lỗi trong thời gian chạy, nên bạn, lập trình viên, không bắt buộc phải xử lý chúng.

Các ngoại lệ (đại diện bởi các lớp con của Ngoại lệ), mặt khác, thường bắt nguồn từ bên trong ứng dụng. Chúng cũng có thể liên quan đến cách thức mà chương trình tương tác với môi trường (ví dụ, đề cập đến một máy từ xa không tồn tại). Các ngoại lệ, nếu được xử lý, thường sẽ được xử lý trong chương trình, nhưng bạn không bắt buộc phải xử lý tất cả chúng. Một số ví dụ về các trường hợp ngoại lệ được chia cho 0, trong các mảng biên và các vấn đề đầu vào / đầu ra của tệp. Lớp Exception có vô số các lớp con, có thể được nhóm thành hai loại chính: ngoại lệ được kiểm tra và ngoại lệ thời gian chạy.

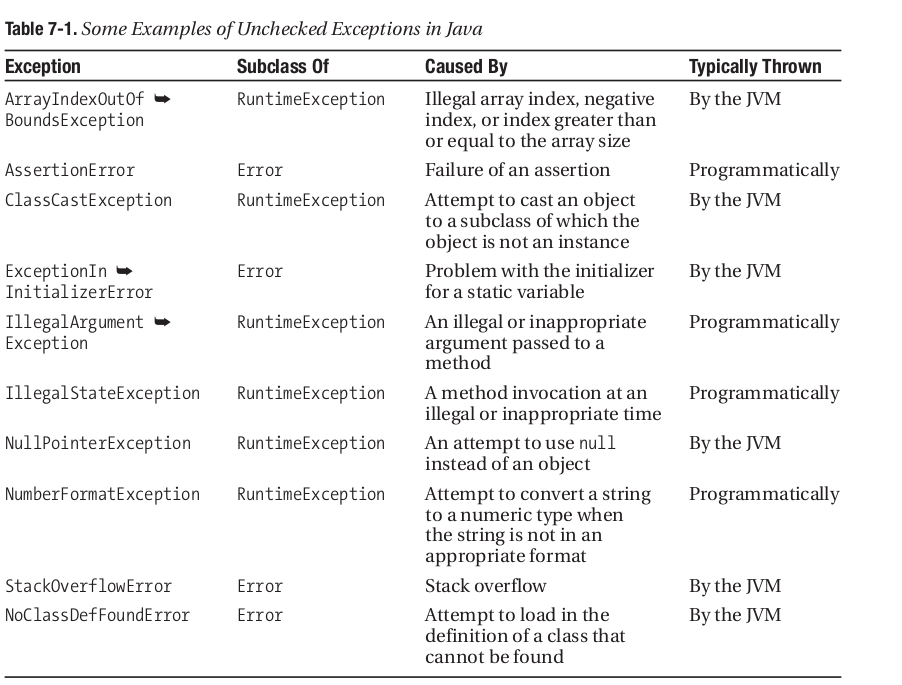
**Checked Exception**

Là các ngoại lệ được chương trình dịch phát hiện. Giúp giảm nguy cơ lỗi khi chạy. Chương trình dịch dự đáon chúng sẽ xảy ra và ta có thể làm 1 trong 2 cách: Ném (throw) và catch chúng. Khi catch lỗi, ta có thể xử lý trường hợp lỗi ở thân hàm.

Điều kiện tạo ra checked exception nằm ngoài sự kiểm soát của chương trình và do đó chúng có thể xảy ra ở một chương trình viết đúng.

**Runtime Exceptions**

Runtime Exception xảy ra do lỗi chương trình. Lập trình viên không phải bắt các ngoại lệ này, bởi vì nếu code đúng, lỗi này sẽ không xảy ra. Tất cả chúng được throw bởi JVM.

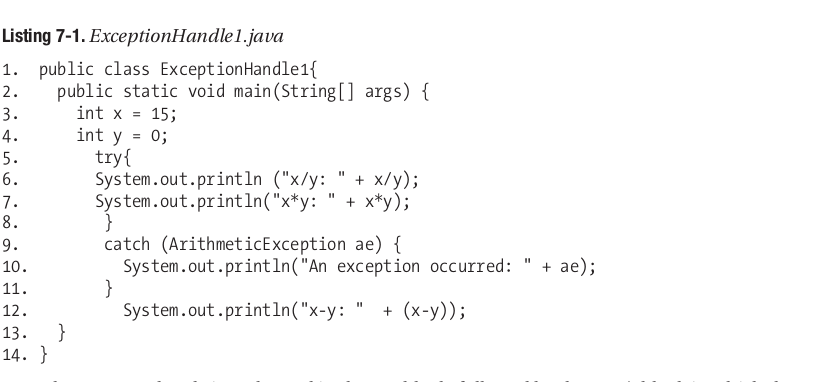
**S****tandard Exceptions**

**Basics of Exception Handling**

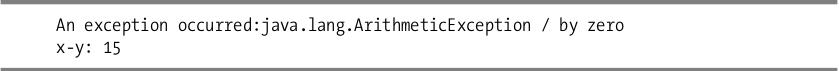
Ta bắt buộc phải xử lý nhưng checked exception, nếu không trình biên dịch sẽ báo lỗi. Cơ chế xử lý ngoại lệ của java có 5 từ khóa: try, catch, throw, throws, finally. Cơ chế xử lý ngoại lệ này có thể áp dụng cho cả runtime exception.

**Using the try and catch Blocks**

Khi chương trình thực hiện một thao tác gây ngoại lệ, một ngoại lệ sẽ dc đưa ra/ ném ra (throw). Ta có thể bắt (catch) lấy ngoại lệ đó bằng cách dùng try và catch.

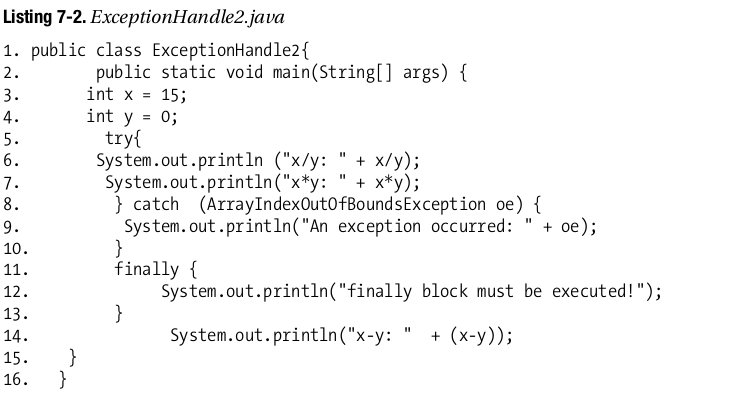
Dòng 6 có một ArithmeticException (x/y là phép chia cho 0). Nếu ngoại lệ này xảy ra, một ArithmeticException sẽ đc khói catch bắt và lưu vào biến ae, sau đó các hoạt động của khối catch sẽ được thực hiện. Do đó cương trình sẽ nhảy từ dòng 6 đến dòng 10. Cuối cùng lệnh sau khối try catch sẽ dc thực hiện.

Output:

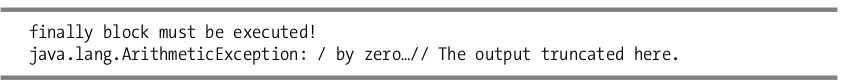
Nếu có một ngoại lệ ném ra từ khối try nhưng không có khối catch exception thích hợp, phương thức sẽ dừng tại đó - không có thêm lệnh nào dc thực hiện. Vì vậy, việc dự đoán các runtime exception và catch chúng rất quan trọng.

**Using the finally Block**

Nếu có một ngoại lể ở khối try nhưng không có khối catch thích hợp, hàm sẽ bị dừng mà không thể thực hiện thêm lệnh nào. Nếu muốn một lệnh luôn luôn được thực hiện kể cả trường hợp trên, ta có thể đưa các lệnh đó vào khối finally. Khối finally sẽ luôn được thực hiện kể cả khối catch không bắt dc exception.

Dòng 6 có exception ArithmeticException nhưng khoong có khối catch thích hợp, tuy nhiên khối finally vẫn sẽ dc thực hiện. Tuy nhiên các lệnh khác ngoài khối finally sẽ ko được thực hiện.

Output:

Một số cách/lỗi làm khối finally không được thực hiện một phần hoặc toàn bộ:

- Có một exception hoặc hàm return trong khối finally;

- hàm System.exit() được gọi trước khi chạy khối finally.

**Using Multiple catch Blocks**

Các cách trên giúp ta c ó thể chắc chắn bắt được exception trong một khối catch. Tuy nhiên có những tình huống ta cần biết rõ đó là exception nào, đến từ đâu để có thể xử lý. Hoặc xử lý cụ thể với mỗi exception. Khi đó ta có thể sử cùng nhiều khối catch, sau cùng là khối finally.

Một số lưu ý khí dùng nhiều khối catch:

- Khi một ngoài lệ được đưa ra, chỉ có 1 khối catch phù hợp bắt dc và các lệnh trong khối đó được thực hiện, các khối catch khác sẽ bị bỏ qua.

- Trường hợp có 2 khối catch với 2 exception là cha con/ ông con, thì exception con sẽ luôn phải được catch trước exception cha/ông,… Nếu ông trình biên dịch sẽ báo lỗi.

Tổng hợp lại quá trình của khối try - catch – finally:

Nếu không có khối catch nào bắt được ngoại lệ:

- Nếu không có khối finally, hàm đó sẽ bị dừng ngay lập tức

- Nếu có khối finally, chỉ các lệnh trong khối đó sẽ dc thực hiên. Sau khi thực hiện xong, hàm sẽ dừng.

Nếu có 1 khối catch bắt dc exception:

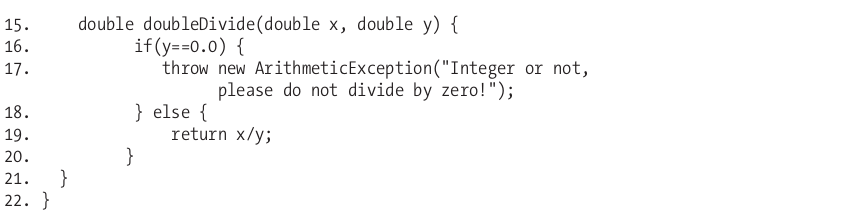
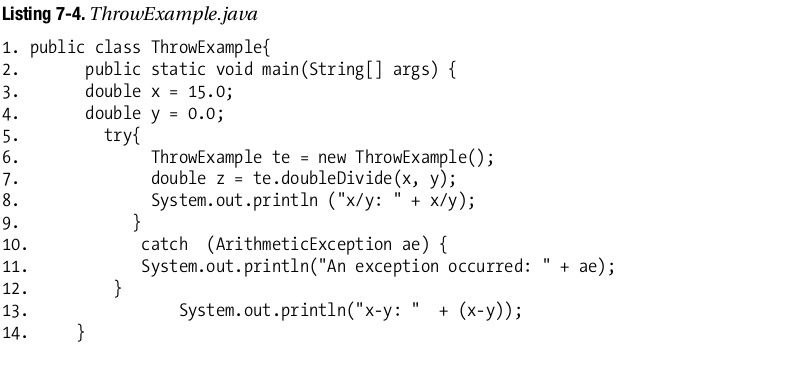
- Nếu không có khối finally, các lệnh trong khối catch đó dc thực hiện, tiếp theo là các lệnh sau khối try catch.

- Nếu có khối finally, thực hiện lệnh khối catch → finally → các lệnh sau khối try - catch - finally.

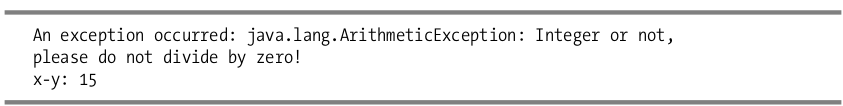
**Throwing Exceptions**

Trong phần trên, ta đã dùng try catch để bắt các exception do JVM đưa ra. Tuy nhiên, ta cũng có thể ném ra các ngọai lệ bằng từ khóa throw.

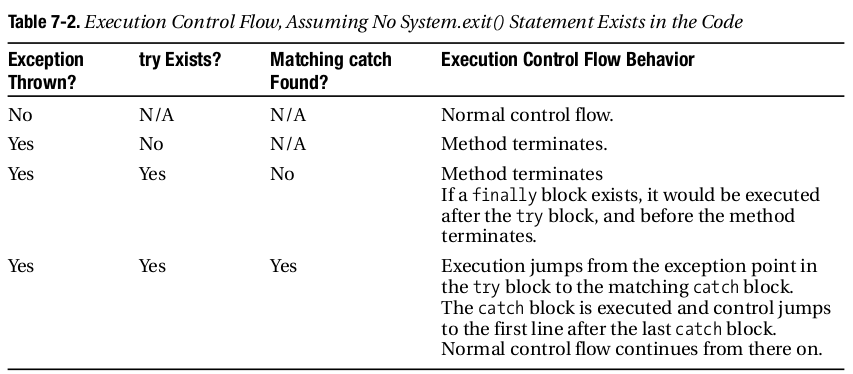
Ví dụ với ArithmeticException sẽ không bắt exceptionđối với số double, vị vậy ta cần viết hàm chia số thực và ném ra ngoại lệ này:



Output:



**Control Flow in Exception Condition**

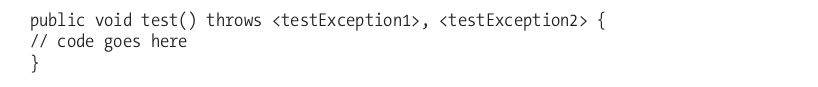


**Declaring Exceptions**

Khi chương trình có exception, làm sao ta có thể biết exception đó thuộc hàm nào - dòng nào? Câu trả lời là với các hàm có thể có 1 checked exception, ta phải khai báo nó.

**Checked Exception: Duck It or Catch It**

Tất cả các checked exception mà hàm đó có thể đưa ra (không bị bắt bởi khối try catch)đêù phải được khai báo với từ khóa throws.

Tuy nhiên, bạn có thể khai báo exception mà ko nhất thiết trong hàm phải đưa ra exception đó.

Nếu hàm gọi một hàm khác ném exception bằng throws thì hàm đó cũng phải ném exception đó bằng throws hoặc catch bằng khối try catch.

**Declaring Exceptions when Overriding**

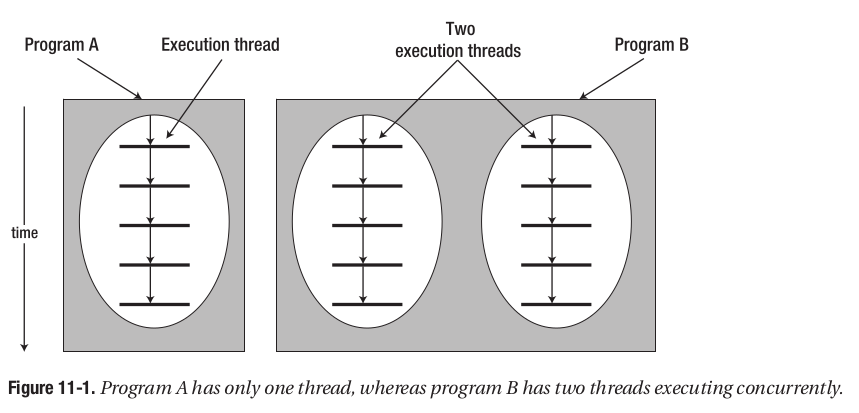
Nếu lớp con ghi đè một hàm của lớp cha, và hàm đó của lớp cha ném các ngoại lệ thì các hàm đó của lớp con không được ném bất kỳ ngoại lệ nào khác ngoại lệ của hàm tại lớp cha. Nói các khác, lớp con hoặc là ném ngoại lệ giống với ở lớp cha, hoặc ném ngoại lệ là 1 nhánh cảu ngoại lệ ở lớp cha, hoặc không ném ngoại lệ của lớp cha.

- Nếu phương thức ghi đè không ném bất kỳ ngoại lệ nào, phương thức ghi đè không thể ném bất kỳ ngoại lệ được kiểm tra nào, nhưng nó vẫn có thể ném ngoại lệ thời gian chạy.

**Multi-Thread in Java**

**Understanding Threads**

Một chương trình ứng dụng có một điều khiển thực thi thực thi một lệnh tại một thời điểm từ ứng dụng, từ trên xuống theo tuần tự. Luồng điều khiển thực thi này được gọi là luồng thực thi. Vì vậy, mỗi ứng dụng có ít nhất một luồng thực thi. Trong môi trường đa luồng, bạn có thể tạo nhiều hơn một luồng trong ứng dụng. Điều này tương đương với việc sinh ra nhiều quy trình, mỗi quy trình có luồng thực thi riêng, cho phép thực hiện đồng thời nhiều tác vụ trong cùng một chương trình ứng dụng

Việc suwr dụng multi thread sẽ giúp tận dùng các luồng của mt để chương trình xử lý nhanh hơn. Tuy nhiên cũng sẽ tạo ra nhiều vấn đề về tranh chấp tài nguyên và nhất quán dữ liệu.

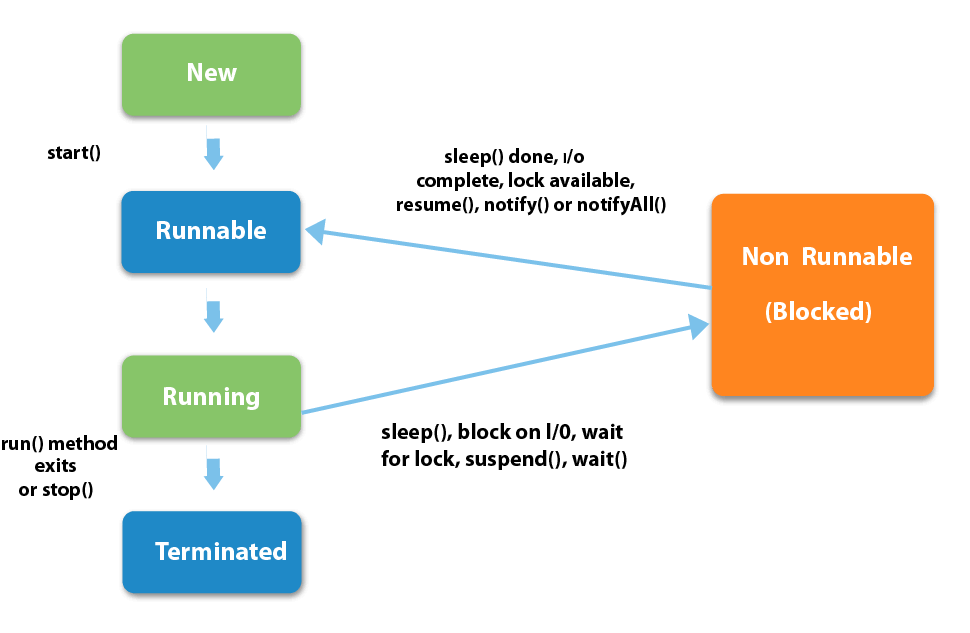
Ngay cả một chương trình không đa luồng cũng có một luồng thực thi, được gọi là luồng chính - main thread. Trong một chương trình đa luồng, bạn có thể sinh ra các luồng khác ngoài luồng chính. Bạn có thể viết một thread class theo một trong hai cách:

- Extend the java.lang.Thread class

- Implement the Runnable interface

Lớp Object chứa một số phương thức được sử dụng để quản lý vòng đời của một luồng

**Life Cycle of Thread**

****

**Creating a Thread Using the Thread Class**

Tạo thread bằng Thread class như sau:

- Viết class extends Thread Class. Sau đó ghi đè phương thức run()

- Khởi tạo luồng bằng cách khởi tạo biến class vừa tạo ở trên

- Bắt đầu luồng chạy bằng cách chạy phương thức start()

Luồng mới được tạo ra sẽ chạy theo phương thức run() mà ta đã override. Ngoài ra, luồng đó sẽ không chạy ngay khi phương thức start() được gọi. Nó chỉ đăng ký luồng với CPU và cuối cùng, nó sẽ chạy phương thức run.

Ta có thể dọi phương thức run()của class để chạy trực tiếp trên luồng gọi nó.

**Creating a Thread Using the Runnable Interface**

Java chỉ cho phép kế thừa đơn, nên nếu extends từ class Thread, ta sẽ ko thể extends từ lớp nào khác. Vì vậy, có 1 cách khác là implement Runnable interface. Có 4 bước:

1. Viết 1 class implement Runnable interface, sau đó override phương thức run() trong class.

2. khởi tạo object của class.

3. Khỏi tạo 1 Thread object với đầu vào constructor là object class ở bước 2.

4. Start object thread với phương thức start().

**Sleeping a thread**

Ta có thể đưa 1 thread vào trạng thái sleep, ở trạng thái này, thread sẽ tạm dừng hoạt động đến hết thời gian ngủ, sau đó sẽ tiếp tục chạy từ trạng thái trước khi sleep. Thời gian ngủ được đưa vào như là đối số của hàm:

public static void sleep(long milliseconds) throws InterruptedException { }

public static void sleep(long milliseconds, int nanoseconds) throws

InterruptedException { }

**ThreadPool**

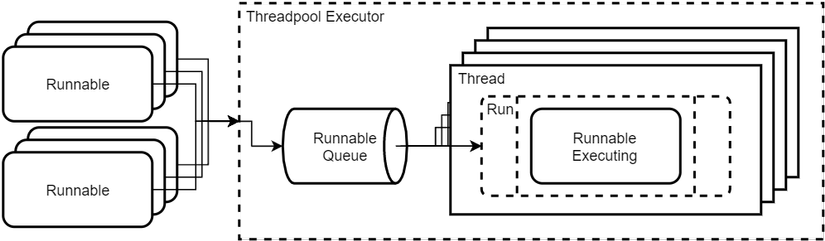
**Understanding ThreadPool**

Nhiều thread giúp chương trình thực hiện công việc nhanh hơn, tuy nhiên, nếu có qua nhiều thread tạo ra dẫn đến chương trình mất nhiều thời gian khởi tạo, cấp phát bộ nhớ → giảm hiệu năng thậm chí crash.

Ví dụ ta mỗi khi tải 1 file, ta tạo 1 thread để tải → tải 1000 hoặc hơn sẽ phải tạo hơn 1000 thread.

Để khắc phục, ta sử dụng ThreadPool để giới hạn số thread được tạo và chạy cùng lúc. Với threadpool, ta sẽ đưa các Runnable, Thread, task vào 1 queue, và threadpool sẽ lấy lần lượt các tác vụ đó ra và thực hiện. Số lượng luồng thực hiện trong threadpool sẽ được đưa vào.

**ThreadPoolExecutor**



Khởi tạo ThreadPoolExecutor

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(corePoolSize, maxPoolSize, keepAlive, unit, workQueue, handler);

Các đối số:

- Đối số 1: (corePoolSize) là số lượng Thread tối thiểu trong ThreadPool. Khi khởi tạo, số lượng Thread có thể là 0. Khi nhiệm vụ được thêm vào thì Thread mới được tạo ra và kể từ đây, nếu số lượng Thread ít hơn corePoolSize thì những Thread mới sẽ được tạo ra đến khi số Thread bằng giá trị của corePoolSize.

- Đối số 2: (maximumPoolSize) là số lượng tối đa các Thread trong ThreadPool.

- Đối số 3: (keepAliveTime): khi số Thread lớn hơn corePoolSize thì keepAliveTime là thời gian tối đa mà 1 Thread "nhàn rỗi" chờ nhiệm vụ. Khi hết thời gian chờ mà Thread đó chưa có nhiệm vụ thì nó sẽ bị hủy.

- Đối số 4: (unit) là đơn vị thời gian của keepAliveTime

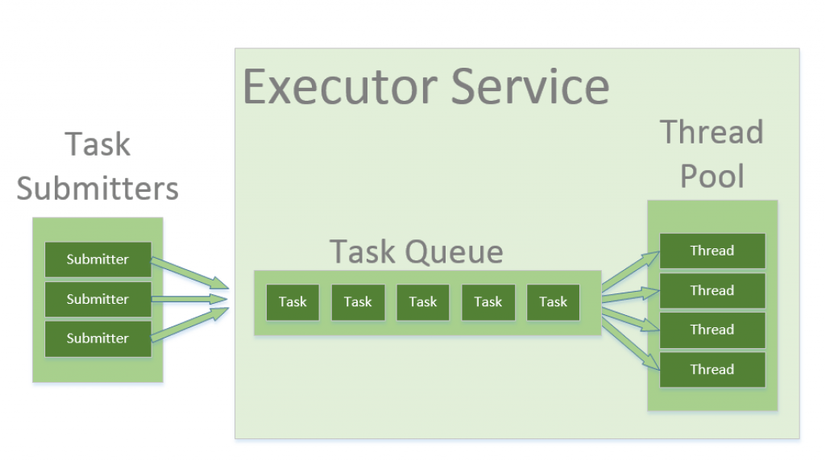
- Đối số 5: (workQueue) là hàng đợi dùng để chứa các nhiệm vụ mà các Thread sẽ lấy chúng ra và thực thi lần lượt

- Đối số 6: (handler): Hành động khi một request (task) bị từ chối (rejected)

Để thêm 1 Runaable vào queue, dùng hàm execute(Runnable run) của ThreadPoolExecutor.

**ExecutorService**

Kể từ Java 5 trở đi, ThreadPool đã được xây dựng sẵn trong gói java.util.concurrent, vì vậy chúng ta không cần phải tạo một ThreadPool mà thay vào đó chúng ta sẽ sử dụng các lớp có sẵn của gói này. Java cung cấp cho chúng ta lớp Executor, interface của lớp Executor là ExecutorService. Interface ExecutorService đại diện cho cơ chế thực thi bất đồng bộ có khả năng thực thi các nhiệm vụ trong background. ExecutorService là một đối tượng chịu trách nhiệm quản lý các luồng và thực hiện các tác vụ Runnable được yêu cầu xử lý. Nó tách riêng các chi tiết của việc tạo Thread, lập kế hoạch (scheduling), … để chúng ta có thể tập trung phát triển logic của tác vụ mà không quan tâm đến các chi tiết quản lý Thread.

Tạo ExecutorService:

ExecutorService pool = Executors.newFixedThreadPool(5);

Ta có thể tạo thread pool thông qua ExecutorService, các task sẽ được gửi vào pool và và dc xử lý tự động bằng 1 trong những phương thức sau:

- Single Thread Executor : Trong ThreadPool chỉ có 1 Thread và các task sẽ được sử lý một cách tuần tự. Tên method “newSingleThread-Executor()”

- Cached ThreadPool : Trong ThreadPool sẽ có rất nhiều Thread, và các task sẽ được sử lý một cách song song. Các Thread cũ sau khi sử lý xong sẽ được sử dụng lại cho tác vụ mới. Mặc định nếu một Thread không được sử dụng trong vào 60 giây thì Thread đó sẽ được hủy (shut down). Tên method “newCachedThreadPool()”

- Fixed Thread Pool : Trong ThreadPool sẽ được cố định (fixed) số lượng các Thread. Nế u một task mới được đưa vào mà các thread đều đang “bận rộn” thì task đó sẽ được gửi vào Blocking Queue và ngay sau khi một Thread đã thực thi xong nhiệm vụ của nó thì nhiệm vụ đang ở trong Queue đó sẽ được push ra khỏi Queue và được Thread đó xử lý tiếp. Method “newFixedThreadPool()”

- Scheduled Thread Pool : Tương tự như “Cached Thread Pool” nhưng sẽ có khoảng delay giữa các Thread. Method “newScheduledThreadPool()”

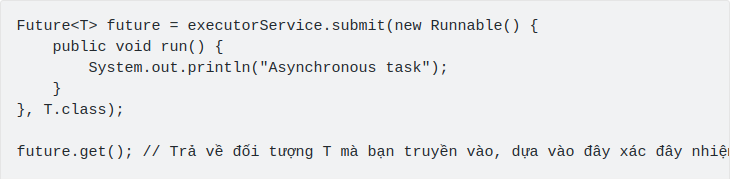
- Single Thread Scheduled Pool : Tương tự như “Single Thread Executor” nhưng sẽ có khoảng delay giữa các Thread. Method “newSingleThreadScheduledExecutor()”

**Cách dùng ExecutorService**

Các phương thức khác nhau để đưa task vào ExecutorService

- execute(Runnable): Đưa vào Runnable, chạy Runaable bất đồng bộ và ko lấy kết quả trả về

- submit(Runnable): Đưa vào Runnable, chạy Runaable bất đồng bộ và lấy kết quả trả về 1 đối tương, đựa vào đây để xác nhận phương thức hoàn tất.

****

- submit(Callable): Đưa vào Callable, trả về kết quả của Callable. Hàm call() của Callable tương tự hàm run() nhưng có kết quả trả về. Kết quả của Callable dc trả về trông qua Future<T> - T là class mà Callabletrả về. Future<T> dùng phương thức get() để lấy kết quả. Nó sẽ block thread gọi nó cho đến khi Callable xử lý xong và trả về kết quả.

- invokeAny(Collection<? extends Callable<T>> tasks): Trả về kết quả của Callable hoàn thành đầu tiên trong Collection.

- invokeAll(): Trả về list các Future kết quả quả tất cả các Callable trong Collection.

- shutdown(): ngừng nhận thên Runnable, tắt Service khi các Runaable chạy xong.

- shutdownNow(): Dừng ngay lập tức Server, tuy ta ko chắc chắn tất cả luồng sẽ dừng nhưng đây là điều tốt nhất có thể làm.

**Understanding and Using CompletableFuture**

CompletableFuture được sử dụng để lập trình bất đồng bộ trong Java.