Hướng dẫn lập trình java căn bản

I. Các khái niệm căn bản

1. Các kiểu dữ liệu nguyên thủy trong java

- Có 8 loại dữ liệu nguyên thủy (primitive data) trong JAVA:
  - Dùng cho kiểu số nguyên có 4 loại: byte, short, int, long
  - Kiểu số thực ta có: float, double
  - Kiểu ký tự: char
  - Kiểu logic: trả về giá trị true hoặc false (đúng hoặc sai)

Kiểu dữ liệu	Ghi chú	Số bit	Giá trị nhỏ nhất	Giá trị lớn nhất
byte	Số tự nhiên 8 bit	8	-128 (-2^7)	127 (2^7-1)
short	Số tự nhiên 16 bit	16	-32,768 (-2^15)	32,767 (2^15 -1)
int	Số tự nhiên 32 bit	32	- 2,147,483,648 (-2^31)	2,147,483,647 (2^31 -1)
long	Số tự nhiên 64 bit	64	-9,223,372,036,854,775,808 (-2^63)	9,223,372,036,854,775,807 (2^63 -1)
float	Số thực 32 bit	32	-3.4028235 x 10^38	3.4028235 x 10^38
double	Số thực 64 bit	64	-1.7976931348623157 x 10^308	1.7976931348623157 x 10^308
boolean	Kiểu logic		false	true
char	Kiểu ký tự	16	'\u0000' (0)	'\uffff' (65,535).

- Để chạy 1 chương trình trên Neatbean ta nhấn Ctrl F6
- Để viết tắt lệnh xuất có thể gõ sout tab
- Biến: lưu ý khi khai báo và gán giá trị cho biến kiểu float, double, long ta phải thêm vào cuối giá trị số đó chữ f, d, l để phân biệt
- Khi xuất ra câu lệnh để nối chuỗi với số ta dùng dấu '+'
- Giá trị Boolean nếu không ghi gì trong if thì default là nếu value == true, có kích thước 1 bit
- Kiểu char trong java sử dụng số nguyên k âm từ 0→2^16 – 1, đại diện cho 1 kí tự Unicode
- Trong java kiểu số nguyên mặc định là 4byte, double là kiểu mặc định của số thực
- Kiểu dl nguyên thủy được lưu trữ ntn trên bộ nhớ:
  Java không đảm bảo rằng mỗi biến sẽ tương ứng với một vị trí trên bộ nhớ. chẳng hạn Java sẽ tối ưu theo cách biến 'i' sẽ được lưu trữ trên bộ đăng ký (register), hoặc thậm trí không được lưu trữ ở đâu cả, nếu trình biên dịch

nhận thấy rằng bạn không bao giờ sử dụng giá trị của nó, hoặc nó có thể được dõi theo thông qua code và sử dụng các giá trị phù hợp một cách trực tiếp.

2. Các kiểu tham chiếu

- Trong Java một kiểu dữ liệu được tạo ra bởi sự kết hợp các kiểu nguyên thủy với nhau được gọi là kiểu tham chiếu (Reference type). Kiểu tham chiếu thường được sử dụng nhất đó là String, nó là sự kết hợp của các ký tự.
- Các kiểu dữ liệu tham chiếu được tạo ra dựa trên một lớp. Lớp (class) giống như một bản thiết kế (blueprint) để định nghĩa một kiểu tham chiếu.
- Tất cả các kiểu dl khác đều mở rộng từ Object chúng là các kiểu dl tham chiếu

- Kiểu tham chiếu được lưu trữ ntn trên bộ nhớ:
  Khi bạn sử dụng toán tử new (Ví dụ new Object()), Java sẽ tạo ra một thực thể mới trên bộ nhớ. Bạn khai báo một biến và khởi tạo giá trị của nó thông qua toán tử new, chẳng hạn Object a = new Object(); Java sẽ tạo ra một thực thể mới trong bộ nhớ, và một tham chiếu 'a' trỏ tới vị trí bộ nhớ của thực thể vừa được tạo ra.

  Khi bạn khai báo một biến b Object b = a; không có thực thể nào được tạo ra trong bộ nhớ, Java chỉ tạo ra một tham chiếu 'b', trỏ tới vị cùng vị trí mà 'a' đang trỏ tới.

3. Các toán tử

- > Lớn hơn
- < Nhỏ hơn
- >= Lớn hơn hoặc bằng
- <= Nhỏ hơn hặc bằng
- && Và
- || hoặc
- == So sánh bằng
- != So sánh khác nhau
- ! Phủ định

4. Câu điều kiện rẽ nhánh

- Cú pháp switch case:

```
// variable_to_test: Một biến để kiểm tra.
switch ( variable_to_test ) {
```

```
        case value1:
         // Làm gì đó tại đây ...
         break;
        case value2:
         // Làm gì đó tại đây ...
         break;
        default:
         // Làm gì đó tại đây ...
      }
```

- Nếu có nhiều case xử lí giống nhau bạn có thể đặt chung vs nhau vd: case 1: case 2: ….. break;

5. Vòng lặp for, while, do while

- Cú pháp for:

```
// start_value: Giá trị bắt đầu
// end_value: Giá trị kết thúc
// increment_number: Giá trị tăng thêm sau mỗi bước lặp.
for ( start_value; end_value; increment_number ) {
  // Làm gì đó tại đây ...
}
```

- Cú pháp while:

```
// Trong khi condition (điều kiện) đúng, thì làm gì đó.
while ( condition ) {
  // Làm gì đó tại đây...
}
```

- Cú pháp do while:

```
// Vòng lặp do-while làm việc ít nhất 1 bước lặp (iteration)
// và trong khi điều kiện còn đúng thì nó còn làm việc tiếp.
do {
  // Làm gì đó tại đây.
}
while (condition);
```

- Lệnh **continue** có thể xuất hiện trong một vòng lặp, khi bắt gặp lệnh **continue** chương trình sẽ bỏ qua các dòng lệnh bên trong khối lệnh và phía dưới của ***continue*** và bắt đầu một bước lặp (iteration) mới (Nếu các điều kiện vẫn đúng).

- Vòng lặp có nhãn:

- **Java** cho phép bạn dán một nhãn (Label) cho một vòng lặp, nó giống việc bạn đặt tên cho một vòng lặp, điều này có ích khi bạn sử dụng nhiều vòng lặp lồng nhau trong một chương trình.

- Bạn có thể sử dụng lệnh **break labelX;** để **break** vòng lặp được dán nhãn **labelX**.
- Bạn có thể sử dụng lệnh **continue labelX;** để **continue** vòng lặp được dán nhãn **labelX.**

  Vd: break label1; trong vòng for thứ 2

- Cú pháp:

```
// Vòng lặp for với nhãn (Label)
label1: for( ... )  {

}

// Vòng lặp while với nhãn (Label)
label2: while ( ... )  {

}

// Vòng lặp do-while với nhãn (Label)
label3: do  {


} while ( ... );
```

6. Mảng

- Mảng là một danh sách các phần tử được sắp xếp liền nhau trên bộ nhớ.
- Trong **Java**, mảng (array) là một tập hợp các phần tử có cùng kiểu dữ liệu, có địa chỉ tiếp nhau trên bộ nhớ (memory). Mảng có số phần tử cố định và bạn không thể thay đổi kích thước của nó.

```
// Khai báo một mảng, chưa chỉ rõ số phần tử.
int[] array1;
```

```
// Khởi tạo mảng với 100 phần tử
// Các phần tử chưa được gán giá trị cụ thể
array1 = new int[100];

// Khai báo một mảng chỉ rõ số phần tử
// Các phần tử chưa được gán giá trị cụ thể
double[] array2 = new double[10];

// Khai báo một mảng với các phần tử được gán giá trị cụ thể.
// Mảng này có 4 phần tử
long[] array3= {10L, 23L, 30L, 11L};
//nếu là String thì phải có new String[]{1, 2, 3}; mới dk
```

- Sử dụng foreach khi muốn in hết các phần tử trong mảng ra
- Cú pháp : for(int a: arr){}

```
// arrayOrCollection: Mảng hoặc tập hợp (Collection).
for (Type variable: arrayOrCollection)  {
    // Code ...
}
```

- Khi muốn in chiều dài của mảng ta dùng arr.length
- Vòng lặp for-each được đưa vào Java từ phiên bản 5. Là một vòng lặp được sử dụng để duyệt qua (traverse) một mảng hoặc một tập hợp (collection), nó sẽ di chuyển lần lượt từ phần từ đầu tiên tới phần tử cuối cùng của mảng hoặc tập hợp.
- Java cung cấp cho bạn một số phương thức tĩnh tiện ích dành cho mảng, chẳng hạn sắp xếp mảng, gán các giá trị cho toàn bộ các phần tử của mảng, tìm kiếm, so sánh mảng... Các phương thức này được định nghĩa trong lớp Arrays.

• Arrays

```
// Truyền vào các tham số cùng kiểu, và trả về một danh sách (List).
public static <T> List<T> asList(T... a)

// Kiểu X có thể là: byte, char, double, float, long, int, short, boolean
----------------------

// Tìm kiếm chỉ số của một giá trị xuất hiện trong mảng.
// (Sử dụng thuật toán tìm kiếm nhị phân (binary search))
```

```
public static int binarySearch(X[] a, X key)

// Copy các phần tử của một mảng để tạo ra một mảng mới với độ dài chỉ định.
public static int[] copyOf(X[] original, X newLength)

// Copy một phạm vi chỉ định các phần tử của mảng để tạo một mảng mới
public static double[] copyOfRange(X[] original, int from, int to)

// So sánh hai mảng
public static boolean equals(X[] a, long[] a2)

// Gán cùng một giá trị cho tất cả các phần tử của mảng.
public static void fill(X[] a, X val)

// Chuyển một mảng thành chuỗi (string)
public static String toString(X[] a)
```

- Trong demo có vd:

```
Arrays.sort(years);
// Chuyển một mảng thành chuỗi
String yearsString = Arrays.toString(years);
```

- Mảng 2 chiều: length là một thuộc tính (property) của mảng, trong trường hợp mảng 2 chiều, thuộc tính này chính là số dòng của mảng.

```
// Khai báo một mảng có 5 dòng, 10 cột
MyType[][] myArray1 = new MyType[5][10];

// Khai báo một mảng 2 chiều có 5 dòng.
// (Mảng của mảng)
MyType[][] myArray2 = new MyType[5][];

// Khai báo một mảng 2 chiều, chỉ định giá trị các phần tử.
MyType[][] myArray3 = new MyType[][] {

   { value00, value01, value02 , value03 },
   { value10, value11, value12 }
};
```

// ** Chú ý:

// MyType có thể là các kiểu nguyên thủy (byte, char, double, float, long, int, short, boolean) hoặc là kiểu tham chiếu.

- Nếu bạn khai báo một mảng của kiểu nguyên thủy (byte, char, double, float, long, int, short, boolean), các phần tử không được chỉ định giá trị, chúng sẽ có giá trị mặc định
- Giá trị mặc định 0 ứng với các kiểu byte, double, float, long, int, short.
- Giá trị mặc định false ứng với kiểu boolean.
- Giá trị mặc định '\u0000' (Ký tự null) ứng với kiểu char.
- Ngược lại, nếu bạn khai báo một mảng của kiểu tham chiếu, nếu một phần tử của mảng không được chỉ định giá trị, nó sẽ có giá trị mặc định là null.
- Mảng 2 chiều thực sự là một mảng của mảng.
- Trong Java mảng 2 chiều thực sự là một mảng của mảng, vì vậy bạn có thể khai báo một mảng 2 chiều chỉ cần chỉ định số dòng, không cần phải chỉ rõ số cột. Chính vì mảng 2 chiều là "Mảng của mảng" nên thuộc tính length của mảng 2 chiều trả về số dòng của mảng.

7. Các phím tắt trong neatbean

- Ctrl /: bật tắt cmt
- Ctrl _: để hiện gợi ý nhanh
- Shift f6: run
- Vào property chọn run chọn main class

bo boolean
br break;
ca catch (
cl class
cn continue
db double
df default:
dowhile do { } while (condition);
eq equals

St String
ab abstract

fl float

```
fori          for (int i = 0; i < arr.length; i++) {}
forl          for (int i = 0; i < lst.size(); i++) {

ir            import
le            length

newo          Object name = new Object(args);
pe            protected
pr            private

sout          System.out.println ("|");

st            static

th            throws
tr            transient
trycatch      try {}

psvm          public static void main(String[] args){}
pu            public
re            return
```

8. Class

- Khi chúng ta nói về Cây, nó là một thứ gì đó trìu tượng, nó là một lớp (class). Nhưng khi chúng ta chỉ thẳng vào một cái cây cụ thể thì lúc đó đã rõ ràng và đó là đối tượng (object) (Cũng được gọi là một thể hiện (instance))

```
public class Person {

    // Đây là một trường (Field).
    // Lưu trữ tên người.
    public String name;

    // Đây là một Constructor (Phương thức khởi tạo)
    // Dùng nó để khởi tạo đối tượng.
    // Constructor này có một tham số.
    // Constructor luôn có tên giống tên của lớp.
    public Person(String persionName) {
        // Gán giá trị từ tham số vào cho trường name.
        this.name = persionName;
    }
```

```
    // Đây là một phương thức trả về kiểu String.
    public String getName() {
        return this.name;
    }

}
```

Trường (Field)

- Trường thông thường
- Trường tĩnh (static Field)
- Trường final (final Field)
- Trường tĩnh và final (static final Field)

- Ta in ra giá trị của static field thông qua tên lớp ClassName.Static_Field
- Các trường final là các trường mà không thể gán giá trị mới cho nó, nó giống như một hằng số.

9. Method

- Phương thức (Method):

- Phương thức thông thường.
- Phương thức tĩnh
- Phương thức **final**. (Sẽ được đề cập trong phần thừa kế của class).

- Phương thức tĩnh cũng được truy cập thông qua tên lớp, ta có thể gọi thông qua object nhưng cách này k đk khuyến khích

10. Thừa kế trong java

- Java cho phép viết class mở rộng từ một class. Class mở rộng từ một class khác được gọi là class con. Class con có khả năng thừa kế các trường và các method từ class cha.

Vd :

```
public class Ant extends Animal { }
```

- Sử dụng từ khóa extends để thể hiện kế thừa từ lớp nào
- Cách để kiểm tra Cat có phải là animal không

```
/ Kiểm tra xem 'cat' có phải là đối tượng của Animal hay không.
    // Kết quả rõ ràng là true.
```

boolean isAnimal = cat instanceof Animal;

11.Hướng dẫn tra cứu sử dụng Javadoc

- Link tra cứu:

http://o7planning.org/vi/10317/jdk-javadoc-dinh-dang-chm

12.Thừa kế và đa hình trong java

- Phân biệt Class, cấu tử (constructor) và đối tượng:
- Lớp **Person** mô phỏng một lớp người, nó là một thứ gì đó trừu tượng, nhưng nó có các trường để mang thông tin, trong ví dụ trên là tên, năm sinh...

* Cấu tử (Constructor)

- Người ta còn gọi là **"Phương thức khởi tạo"**

  - Cấu tử luôn có tên giống tên lớp.
  - Một class có một hoặc nhiều cấu tử.
  - Cấu tử có hoặc không có tham số, cấu tử không có tham số còn gọi là cấu tử mặc định.
  - Cấu tử được sử dụng để tạo ra một đối tượng của lớp.

  Như vậy lớp **Person** (Mô tả lớp người) là thứ trừu tượng, nhưng khi chỉ rõ vào bạn hoặc tôi thì đó là 2 đối tượng (instance) thuộc lớp **Person**. Và Constructor là phương thức đặc biệt để tạo ra đối tượng, Constructor sẽ gán các giá trị vào các trường (field) của class cho đối tượng..

- Ở đây chúng ta có class **Animal**, với một method không có nội dung.
  - public abstract String getAnimalName();
- Method này là một method trừu tượng (abstract), tại các class con cần phải khai báo và triển khai nội dung của nó. Method này có ý nghĩa là trả về tên loài.

  Class **Animal** có 1 phương thức trừu tượng nó phải được khai báo là trừu tượng (abstract). Class trừu tượng có các cấu tử (constructor) nhưng bạn không thể khởi tạo đối tượng từ nó.
- Về bản chất nghĩa là bạn muốn tạo một đối tượng động vật, bạn cần tạo từ một loại đông vật cụ thể, trong trường hợp này bạn phải khởi tạo từ cấu tử (constructor) của **Cat**, **Mouse** hoặc **Duck**.
- Cat cũng có các cấu tử của nó, và cũng có các trường của nó. Trong dòng đầu tiên của cấu tử bao giờ cũng phải gọi super(..) nghĩa là gọi lên cấu tử cha, để khởi tạo giá trị cho các trường của class cha.

- Nếu bạn không gọi, mặc định Java hiểu là đã gọi super(), nghĩa là gọi cấu tử mặc định của class cha.

```
// Thực hiện (implement) phương thức trừu tượng được khai báo tại lớp cha.
@Override
public String getAnimalName() {
   return "Cat";
}
```

- Khi gọi nó sẽ gọi constructor của lớp cha trk , khi khai báo 1 obj Animal a = new Cat(…..) ta chỉ gọi được các methods của lớp cha nên phải khai báo Cat = new Cat để SD các methods riêng của class con
- Ép kiểu cast trong java: Cat cat = (Cat) animal; //trong khi Animal = new Cat(a, b);
- Cách lấy số random

```
import java.util.Random;
// Trả về giá trị ngẫu nhiên 0 hoặc 1.
int random = new Random().nextInt(2);
```

13.Tính đa hình

- Bạn có một con mèo nguồn gốc châu Á (AsianCat), bạn có thể nói nó là một con mèo (Cat) hoặc nói nó là một con vật (Animal) đó là một khía cạnh của từ đa hình.
- Hoặc một ví dụ khác: Trên lý lịch của bạn ghi rằng bạn là một người châu Á, trong khi đó bạn thực tế là một người Việt Nam.
- Ví dụ dưới đây cho bạn thấy cách hành sử giữa khai báo và thực tế
- Class AsianCat là một class thừa kế từ Cat.

14.Abstract class và interface

```
// Đây là một lớp trừu tượng.
// Nó bắt buộc phải khai báo là abstract
// vì trong nó có một phương thức trừu tượng
public abstract class ClassA  {
```

```
    // Đây là một phương thức trừu tượng.
    // Nó không có thân (body).
    // Access modifier của phương thức này là public.
    public abstract void doSomething();

    // Access modifier của phương thức này là protected.
    protected abstract String doNothing();
    // Phương thức này không khai báo access modifier.
    // Access modifier của nó là mặc định.
    abstract void todo() ;
}

// Đây là một lớp trừu tượng.
// Nó được khai báo là abstract,
// mặc dù nó không có phương thức trừu tượng nào.
public abstract class ClassB   {

}
```

Đặc điểm của một class trừu tượng là:

1. Nó được khai báo abstract.
2. Nó có thể khai báo 0, 1 hoặc nhiều method trừu tượng bên trong.
3. Không thể khởi tạo 1 đối tượng trực tiếp từ một class trừu tượng.

- Khi không khai báo abstract ta phải thực hiện tất cả các phương thức abstract còn lại, còn nếu k thực hiện hết ta phải có từ khóa abstract
  - Interface

- Ta biết 1 class con có thể mở rộng từ class cha

```
// Lớp B là con của lớp A, hoặc nói cách khác là B mở rộng từ A
// Java chỉ cho phép một lớp mở rộng từ duy nhất một lớp khác.
public class B extends A  {
  // ....
}

// Trong trường hợp không chỉ rõ lớp B mở rộng từ một lớp cụ thể nào.
// Mặc định, hiểu rằng B mở rộng từ lớp Object.
public class B  {

}

// Cách khai báo này, và cách phía trên là tương đương nhau.
public class B extends Object  {

}
```

- Nhưng 1 class có có thể mở rộng từ nhiều interface

```
// Một lớp chỉ có thể mở rộng từ 1 lớp cha.
// Nhưng có thể thực hiện (mở rộng) từ nhiều Interface.
public class Cat extends Animal implements CanEat, CanDrink {

  // ....
}
```

- Các đặc điểm của interface

1. Interface luôn luôn có modifier là: public interface, cho dù bạn có khai báo rõ hay không.

2. Nếu có các trường (field) thì chúng đều là: public static final, cho dù bạn có khai báo rõ hay không.
3. Các method của nó đều là method trừu tượng, nghĩa là không có thân hàm, và đều có modifier là: public abstract, cho dù bạn có khai báo hay không.
4. Interface không có Constructor (cấu tử).

- Khi interface k chỉ định rõ access modifier thì access modifier của nó là mặc định nghĩa là các class trong cùng 1 packet mới thi hành interface này được
- Khi khởi tạo đối tượng CanEat e = new Cat(); lúc gọi e.eat(); thì nó sẽ biết gọi phương thức nào, tính đa hình thể hiện rõ tại đây java luôn biết đó là đối tượng kiểu gì

15. Access modifier

- Có hai loại modifier trong java: **access modifiers** và **non-access modifiers**.
- Các **access modifiers** trong java xác định độ truy cập (Phạm vi) vào dữ liệu của của các trường, phương thức, cấu tử hoặc class.

- Có 4 kiểu của java **access modifiers**:

1. **private**
2. **(Mặc định)**
3. **protected**
4. **public**

- Và có một vài **non-access modifiers** chẳng hạn **static, abstract, synchronized, native, volatile, transient**, v.v.. Trong tài liệu này chúng ta sẽ học về **access modifier**.

Bảng minh họa dưới đây cho bạn cái nhìn tổng quan về cách sử dụng các access modifier.

Access Modifier	Truy cập bên trong class?	Truy cập bên trong package?	Truy cập bên ngoài package bởi class con?	Truy cập bên ngoài class và không thuộc class con?
private	Y			
Mặc định	Y	Y		
protected	Y	Y	Y	
public	Y	Y	Y	Y

- Trong trường hợp bạn khai báo một trường, method, hoặc cấu tử (constructor), class, .. mà không ghi rõ ràng access modifier, điều đó có nghĩa là nó là access modifier mặc định. Phạm vi truy cập của access modifier mặc định là trong nội bộ package

- Với Interface khi bạn khai báo một trường (Field) hoặc một phương thức (Method) bạn luôn phải khai báo là public hoặc để mặc định, nhưng Java luôn hiểu đó là public.

```
public interface MyInterface  {


 // Đây là một trường, bạn không thể khai báo private hoặc protected.
 public static int MY_FIELD1 = 100;


 // Đây là một trường, bạn để access modifier mặc định.
 // Tuy nhiên Java coi đây là public.
 static int MY_FIELD2 = 200;
  // Đây là một method, bạn để access modifier mặc định.
 // Tuy nhiên Java coi đây là public
 void showInfo();
}
```

- Protected access modifier: có thể truy cập trong hoặc ngoài packet(nhưng phải thông qua tính kế thừa), chỉ áp dụng cho field, method và constructor. Nó không thể áp dụng cho class (class, interface, enum, annotation).

16.Ghi đè phướng thức

- Bạn có thể ghi đè một method của class cha với một method cùng tên cùng tham số tại class con, tuy nhiên bạn không được phép làm hạn chế phạm vi truy cập của method này.
- Có thể @Override chuyển từ protected thành public nhưng ta không thể chuyển từ protected thành mặc định

17.Java enum

- enum trong Java là một từ khóa, một tính năng được sử dụng để đại diện cho số cố định, Ví dụ ngày trong tuần, hành tinh trong hệ mặt trời, v..v
- Thay vì khai báo các thứ trong tuần ta có thể khai báo public static final … trung 1 cái public class khi ta truyền vào giá trị int vào hàm tiếp theo nó sẽ không an toàn
- Chẳng hạn như khi bạn gõ các giá trị cho ngày trong tuần chẳng may trùng nhau. Hoặc gọi hàm Timetable.getJob(int) mà truyền vào giá trị nằm ngoài các giá trị định nghĩa trước.
  - Không phải là kiểu an toàn: Đầu tiên thấy rằng nó là kiểu không an toàn, bạn có thể gọi method getJob(int) và truyền vào bất kỳ giá trị int nào.
  - Không có ý nghĩa trong in ấn: Nếu bạn muốn in ra các ngày trong tuần nó sẽ là các con số, thay vì một chữ có ý nghĩa như "MONDAY".

```
public enum WeekDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY, SUNDAY;
}
//trong 1 class khác
public static String getJob(WeekDay weekDay) {
    if (weekDay == WeekDay.SATURDAY || weekDay ==
WeekDay.SUNDAY) {
        return "Nothing";
    }
    return "Coding";
 }
```

- Enum là một đối tượng tham chiếu giống như class, interface nhưng nó cũng có thể sử dụng cách so sánh ==.
- Cách các obj đối tượng tham chiếu so sánh

```
// Để so sánh các đối tượng tham chiếu thông thường phải sử dụng
method equals(..)
Object obj1 = .... ;

// So sánh đối tượng với null, có thể sử dụng toán tử ==
if(obj1 == null)  {
```

```
}

Object obj2 = ....;
// So sánh khác null.
if (obj1 != null)   {
  // So sánh 2 đối tượng với nhau.
  if(obj1.equals(obj2))   {

  }
}
//còn trong enum, đây dk viết trong main
WeekDay today = WeekDay.SUNDAY;

        // Sử dụng toán tử == để so sánh 2 phần tử của Enum.
        if (today == WeekDay.SUNDAY) {
            System.out.println("Today is Sunday");
        }
```

- Enum có thể sử dụng switch case, ta có thể duyệt các phần tử của enum bằng values()

```
// Lấy ra tất cả các phần tử của Enum.
    WeekDay[] allDays = WeekDay.values();

    for (WeekDay day : allDays) {
        System.out.println("Day: " + day);
    }
```

- Enum có thể có methods và constructor , constructor để khởi tạo enum k được phép gọi từ bên ngoài chỉ dùng trong nội bộ enum
- Bạn có thể ghi đè phương thức trong enum (toString(){ if this == …. Return super.toString();})
- Phương thức trừu tượng trong enum

18. Annotnation

- Annotation (Chú thích) được sử dụng để cung cấp thông tin dữ liệu cho mã Java của bạn. Là thông tin dữ liệu, các Annotation không trực tiếp ảnh hưởng đến việc thực hiện các mã của bạn, mặc dù một số loại chú thích thực sự có thể được sử dụng cho mục đích đó. Annotation đã được thêm vào Java từ Java 5

- Annotation được sử dụng cho các mục đích:
  - Chỉ dẫn cho trình biên dịch (Compiler)
  - Chỉ dẫn trong thời điểm xây dựng (Build-time)
  - Chỉ dẫn trong thời gian chạy (Runtime)

a) Chỉ dẫn cho trình biên dịch

- Java có sẵn 3 Annotation mà bạn có thể sử dụng để cung cấp cho các hướng dẫn để trình biên dịch Java.
  - @Deprecated
  - @Override
  - @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.
- Chú thích @Deprecated được bộ biên dịch quan tâm để thông báo cho bạn nên dùng một cách nào đó thay thế. Hoặc với các IDE lập trình chẳng hạn như **Eclipse** nó cũng sẽ có các thông báo cho bạn một cách trực quan(gạch ngang).
- 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.
- Annotation @Override là không bắt buộc phải chú thích trên method đã ghi đè method của class cha. Đó là một ý tưởng tốt để sử dụng nó. Trong trường hợp một người nào đó thay đổi tên của method của class cha, method tại class của bạn sẽ không còn là method ghi đè nữa. Nếu không có chú thích @Override bạn sẽ không tìm ra. Với các chú thích @Override trình biên dịch sẽ cho bạn biết rằng các phương pháp trong các lớp con không ghi đè bất kỳ phương thức trong lớp cha.
- @Override không cần thiết ghi ra nhưng nó cần cho TH thay đổi tên của method phương thức cha nó sẽ báo lỗi cho bạn
- 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.

b) Chỉ dẫn trong thời điểm xây dựng

- Annotation có thể được được sử dụng tại thời xây dựng (Build-time), khi bạn xây dựng dự án phần mềm của bạn. Quá trình xây dựng bao gồm tạo ra các mã nguồn, biên dịch mã nguồn, tạo ra các file XML (ví dụ như mô tả

triển khai), đóng gói mã biên dịch và các tập tin vào một tập tin JAR, v..v. Xây dựng phần mềm thường được thực hiện bởi một công cụ xây dựng tự động như Apache Ant hoặc Apache Maven . Xây dựng các công cụ có thể quét mã Java của bạn và dựa vào các chú thích (Annotation) của bạn để tạo ra mã nguồn hoặc các tập tin khác dựa trên những chú thích đó.

c) Chỉ dẫn trong thời gian chạy

- Thông thường, các Annotation không có mặt trong mã Java của bạn sau khi biên dịch. Tuy nhiên có thể xác định các Annotation của bạn trong thời gian chạy. Các chú thích này sau đó có thể được truy cập thông qua Java Reflection, và được sử dụng để cung cấp cho các hướng dẫn chương trình của bạn, hoặc API của một số bên thứ ba (Third party API).

- Viết annotation của bạn:

- Sử dụng @interface là từ khóa khai báo một Annotation, annotation khá giống một interface. Annotation có hoặc không có các phần tử (element) trong nó.
- Đặc điểm của các phần tử (element) của annotation:
    - Không có thân hàm
    - Không có tham số hàm
    - Khai báo trả về phải là một kiểu cụ thể:
        - Các kiểu nguyên thủy (boolean, int, float, ...)
        - Enum
        - Annotation
        - Class (Ví dụ String.class)
    - Có thể có giá trị mặc định

```
public @interface MyFirstAnnotation {

    // Phần tử 'name'.
    public String name();

    // Phần tử 'description', có giá trị mặc định "".
    public String description() default "";

}
```

- Annotation có thể được gắn trên:

1. TYPE - Gắn trên khai báo Class, interface, enum, annotation.

2. FIELD - Gắn trên khai báo trường (field), bao gồm cả các hằng số enum.
3. METHOD - Gắn trên khai báo method.
4. PARAMETER - Gắn trên khai báo parameter
5. CONSTRUCTOR - Gắn trên khai báo cấu tử
6. LOCAL_VARIABLE - Gắn trên biến địa phương.
7. ANNOTATION_TYPE - Gắn trên khai báo Annotation
8. PACKAGE - Gắn trên khai báo package.

Vd: *@MyFirstAnnotation*(name = "Some name", description = "Some description")

- Các annotation có phần tử tên value có 1 số đặc biệt

```
// Phần tử có tên 'value' là đặc biệt.
// Thay vì viết @AnnWithValue(value = 100)
// Bạn chỉ cần viết @AnnWithValue(100)
@AnnWithValue(100)
```

- @Retention & @Target

@Retention & **@Target** là 2 annotation sẵn có của **Java**. (học sau)

@Retention

@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

@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.

19. Cách so sánh trong java

- Trong Java có 2 kiểu so sánh:
  - Sử dụng toán tử ==
  - Sử dụng phương thức (method) equals(..)
- Toán tử == dùng so sánh các kiểu nguyên thủy và các kiểu tham chiếu.
- Toán tử equals(..) là một phương thức chỉ dùng cho các kiểu tham chiếu.
  - Đối với kiểu tham chiếu
- Khi bạn so sánh 2 đối tượng tham chiếu theo toán tử ==, có nghĩa là so sánh vị trí trên bộ nhớ mà 2 đối tượng tham chiếu này trỏ tới. Về bản chất là kiểm tra xem 2 tham chiếu đó có cùng trỏ tới một thực thể trên bộ nhớ hay không
- Còn equals() là so sánh nội dung obj trên bộ nhớ
  - Bạn có thể ghi đè phương thức boolean equals(Object obj); mọi class con đều kế thừa class này
- Sắp xếp 1 mảng String: String là 1 class mà các đối tượng của nó có thể so sánh với nhau, theo thứ tự bảng chữ cái, sử dụng phương thức tĩnh của Arrays.sort(str); để sắp xếp
  - Các đối tượng có thể so sánh với nhau(comparable)

    ```
    // Để so sánh được với nhau, lớp Actor cần thi hành interface Comparable.
    public class Actor implements Comparable<Actor>
    ```

    - B.compareTo(a) nếu so sánh bằng thì trả về 0
  - Sắp xếp 1 List:

    ```
    import java.util.ArrayList;
    import java.util.Collections;
    import java.util.List;

    List<Actor> actors = new ArrayList<Actor>();
    ```

```
        actors.add(actor1);
    // Sử dụng phương thức Collections.sort(List)
        // để sắp xếp một danh sách (List)
        Collections.sort(actors);
```

- Sắp xếp sử dụng bộ so sánh comparator: khai báo như comparable, ở dưới @Override compare(o1, o2)

```
    // Sắp xếp mảng, sử dụng: <T> Arrays.sort(T[],Comparator<? supers T>).
        // Và cung cấp một Comparator (Bộ so sánh).
        Arrays.sort(array, new PersonComparator());
          Collections.sort(list, new PersonComparator());
```

- Có thể xem thêm vd ở trên mạng

20. String builder, String, String buffer

- Khi làm việc với các dữ liệu văn bản, Java cung cấp cho bạn 3 class String, StringBuffer và StringBuilder. Nếu làm việc với các dữ liệu lớn bạn nên sử dụng StringBuffer hoặc StringBuilder để đạt hiệu năng nhanh nhất. Về cơ bản 3 class này có nhiều điểm giống nhau(final).
  - String là không thể thay đổi (*immutable*), khái niệm này sẽ được nói chi tiết ở trong tài liệu, và không cho phép có class con.
  - StringBuffer, StringBuilder có thể thay đổi (*mutable*)

- StringBuilder và StringBuffer là giống nhau, nó chỉ khác biệt tình huống sử dụng có liên quan tới đa luồng (Multi Thread).

- Nếu xử lý văn bản sử dụng nhiều luồng (Thread) bạn nên sử dụng StringBuffer để tránh tranh chấp giữa các luồng.
- Nếu xử lý văn bản sử dụng 1 luồng (Thread) nên sử dụng StringBuilder.

⇨ Nếu so sánh về tốc độ xử lý StringBuilder là tốt nhất, sau đó **StringBuffer** và cuối cùng mới là String.

- Mutable có nghĩa là bạn có thể thay đổi giá trị cho nó thông qua method SET còn immutable thì ta không thể set giá trị cho nó từ bên ngoài, nếu muốn thay đổi bạn chỉ có thể tạo đối tượng khác
- String là 1 class k thể thay đổi, có các thuộc tính như length là k thể thay đổi, tất cả các đối tượng String đều tạo ra 1 đối tượng String khác

a) String là 1 lớp rất đặc biệt

- Trong java, **String** là một class đặc biệt, nguyên nhân là nó được sử dụng một cách thường xuyên trong một chương trình, vì vậy đòi hỏi nó phải có hiệu suất và sự mềm dẻo. Đó là lý do tại sao **String** có tính đối tượng và vừa có tính nguyên thủy (primitive).

Tính nguyên thủy:

Bạn có thể tạo một string literal (chuỗi chữ), string literal được lưu trữ trong ngăn sếp (stack), đòi hỏi không gian lưu trữ ít, và rẻ hơn khi thao tác.

- String literal = "Hello World";

Bạn có thể sử dụng toán tử + để nối 2 string, toán tử này vốn quen thuộc và sử dụng cho các kiểu dữ liệu nguyên thủy int, float, double.
Các string literal được chứa trong một bể chứa (common pool). Như vậy hai string literal có nội dung giống nhau sử dụng chung một vùng bộ nhớ trên stack, điều này giúp tiết kiệm bộ nhớ.

Tính đối tượng

Vì String là một class, vì vậy nó có thể được tạo ra thông qua toán tử new.

- String object = new String("Hello World");

Các đối tượng String được lưu trữ trên Heap, yêu cầu quản lý bộ nhớ phức tạp và tốn không gian lưu trữ. Hai đối tượng String có nội dung giống nhau lưu trữ trên 2 vùng bộ nhớ khác nhau của Heap.

```
// Tạo ngầm một String, thông qua "string literal".
// Đây là một "string literal".
// Cách này thể hiện tính nguyên thủy của String.

String str1 = "Java is Hot";
String str3 = str1;//cung tham chiếu cùng trỏ tới 1 vt

// Tạo một cách rõ ràng thông qua toán tử new.
// Đây là một "String object".
// Cách này thể hiện tính đối tượng của String,
// giống như các đối tượng khác trong Java.

String str2 = new String("I'm cool");
```

b) String literal và String object

- Có hai cách để xây dựng một chuỗi (String): ngầm xây dựng bằng cách chỉ định một chuỗi chữ (String literal) hay một cách rõ ràng tạo ra một đối tượng String thông qua toán tử new và cấu tử của String.
- Phương thức ***equals()*** sử dụng để so sánh 2 đối tượng, với **String** nó có ý nghĩa là so sánh nội dung của 2 string. Đối với các kiểu tham chiếu (reference) toán tử == có ý nghĩa là so sánh địa chỉ vùng bộ nhớ lưu trữ của đối tượng. Trong thực tế bạn hay sd String literal hơn vì nó tang tốc chương trình của bạn

c) Methods

SN	**Methods**	**Description**
1	char charAt(int index)	Trả về một ký tự tại vị trí có chỉ số được chỉ định.
2	int compareTo(Object o)	So sánh một String với một Object khác.
3	int compareTo(String anotherString)	So sánh hai chuỗi theo từ điển. (Phân biệt chữ hoa chữ thường)
4	int compareToIgnoreCase(String str)	So sánh hai chuỗi theo từ điển. (Không phân biệt chữ hoa chữ thường)
5	String concat(String str)	Nối chuỗi được chỉ định đến cuối của chuỗi này.
6	boolean contentEquals(StringBuffer sb)	Trả về true nếu và chỉ nếu chuỗi này đại diện cho cùng một chuỗi ký tự như là StringBuffer quy định.
7	static String copyValueOf(char[] data)	Trả về một chuỗi đại diện cho chuỗi ký tự trong mảng quy định.
8	static String copyValueOf(char[] data, int offset, int count)	Trả về một chuỗi đại diện cho chuỗi ký tự trong mảng quy định.
9	boolean endsWith(String suffix)	Kiểm tra nếu chuỗi này kết thúc với hậu tố quy định.

10	boolean equals(Object anObject)	So sánh với một đối tượng
11	boolean equalsIgnoreCase(String anotherString)	So sánh với một String khác, không phân biệt chữ hoa chữ thường.
12	byte[] getBytes()	Mã hóa chuỗi này thành một chuỗi các byte bằng cách sử dụng bảng mã mặc định của flatform (nền tảng), lưu trữ kết quả vào một mảng byte mới.
13	byte[] getBytes(String charsetName)	Mã hóa chuỗi này thành một chuỗi các byte bằng cách sử dụng bảng mã cho trước, lưu trữ kết quả vào một mảng byte mới.
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copy các ký tự từ chuỗi này vào mảng ký tự đích.
15	int hashCode()	Trả về một mã "hash code" cho chuỗi này.
16	int indexOf(int ch)	Trả về chỉ số trong chuỗi này xuất hiện đầu tiên của ký tự cụ thể.
17	int indexOf(int ch, int fromIndex)	Trả về chỉ số trong chuỗi này xuất hiện đầu tiên của ký tự được chỉ định, bắt đầu tìm kiếm từ chỉ số cụ thể đến cuối.
18	int indexOf(String str)	Trả về chỉ số trong chuỗi này xuất hiện đầu tiên của chuỗi quy định.
19	int indexOf(String str, int fromIndex)	Trả về chỉ số trong chuỗi này xuất hiện đầu tiên của chuỗi quy định, bắt đầu từ chỉ số xác định.
20	String intern()	Returns a canonical representation for the string object.
21	int lastIndexOf(int ch)	Trả về chỉ số trong chuỗi này về sự xuất hiện cuối cùng của ký tự cụ thể.

22	int lastIndexOf(int ch, int fromIndex)	Trả về chỉ số trong chuỗi này về sự xuất hiện cuối cùng của ký tự được chỉ định, tìm kiếm lùi lại bắt đầu từ chỉ số xác định.
23	int lastIndexOf(String str)	Trả về chỉ số trong chuỗi này xảy ra cuối cùng bên phải của chuỗi quy định.
24	int lastIndexOf(String str, int fromIndex)	Trả về chỉ số trong chuỗi này về sự xuất hiện cuối cùng của chuỗi xác định, tìm kiếm lùi lại bắt đầu từ chỉ số xác định.
25	int length()	Trả về độ dài chuỗi.
26	boolean matches(String regex)	Kiểm tra chuỗi này khớp với biểu thức chính quy chỉ định hay không.
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	Kiểm tra chuỗi có một phần giống nhau.
28	boolean regionMatches(int toffset, String other, int ooffset, int len)	Kiểm tra chuỗi có một phần giống nhau.
29	String replace(char oldChar, char newChar)	Trả về một chuỗi mới từ thay thế tất cả các lần xuất hiện của ký tự oldChar trong chuỗi này với ký tự newChar.
30	String replaceAll(String regex, String replacement)	Thay thế tất cả các chuỗi con của chuỗi này khớp với biểu thức chính quy bởi String mới replacement (is|te)- thay 2 chuỗi
31	String replaceFirst(String regex, String replacement)	Thay thế chuỗi con đầu tiên của chuỗi này khớp với biểu thức chính quy bởi một String mới replacement
32	String[] split(String regex)	Tách chuỗi này thành các chuỗi con, tại các chỗ khớp với

		biểu thức chính quy cho trước.
33	String[] split(String regex, int limit)	Tách chuỗi này thành các chuỗi con, tại các chỗ khớp với biểu thức chính quy cho trước. Tối đa limit chuỗi con.
34	boolean startsWith(String prefix)	Kiểm tra nếu chuỗi này bắt đầu với tiền tố quy định.
35	boolean startsWith(String prefix, int toffset)	Kiểm tra nếu chuỗi này bắt đầu với tiền tố quy định bắt đầu một chỉ số xác định.
36	CharSequence subSequence(int beginIndex, int endIndex)	Trả về một chuỗi ký tự mới là một dãy con của dãy này.
37	String substring(int beginIndex)	Trả về một chuỗi ký tự mới là một dãy con của dãy này. Từ chỉ số cho trước tới cuối
38	String substring(int beginIndex, int endIndex)	Trả về một chuỗi ký tự mới là một dãy con của dãy này. Từ chỉ số bắt đầu cho tới chỉ số cuối.
39	char[] toCharArray()	Chuyển chuỗi này thành mảng ký tự.
40	String toLowerCase()	Chuyển tất cả các ký tự của chuỗi này sang chữ thường, sử dụng miền địa phương mặc định (default locale)
41	String toLowerCase(Locale locale)	Chuyển tất cả các ký tự của chuỗi này sang chữ thường, sử dụng miền địa phương (locale) cho trước.
42	String toString()	Trả về String này.
43	String toUpperCase()	Chuyển tất cả các ký tự của chuỗi này sang chữ hoa,

		sử dụng miền địa phương mặc định (default locale)
44	String toUpperCase(Locale locale)	Chuyển tất cả các ký tự của chuỗi này sang chữ hoa, sử dụng miền địa phương (locale) cho trước.
45	String trim()	Trả về một String mới, sau khi loại bỏ các ký tự trắng (whitespace) bên trái và bên phải.
46	static String valueOf(primitive data type x)	Returns the string representation of the passed data type argument.

d) String builder and String buffer

- StringBuilder và StringBuffer là rất giống nhau, điều khác biệt là tất cả các phương thức của StringBuffer đã được đồng bộ, nó thích hợp khi bạn làm việc *với ứng dụng đa luồng*, nhiều luồng có thể truy cập vào một đối tượng StringBuffer cùng lúc. Trong khi đó StringBuilder có các phương thức tương tự nhưng không được đồng bộ, nhưng vì vậy mà hiệu suất của nó cao hơn, bạn nên sử dụng StringBuilder trong ứng dụng đơn luồng, hoặc sử dụng như một biến địa phương trong một phương thức.
- Các method của StringBuffer (StringBuilder cũng tương tự)
    - // Cấu tử.
    - StringBuffer() // an initially-empty StringBuffer
    - StringBuffer(int size) // with the specified initial size
    - StringBuffer(String s) // with the specified initial content
    -
    - // Độ dài
    - int length()
    -
    - // Các method xây dựng nội dung
    - // type ở đây có thể là kiểu nguyên thủy (primitive), char[], String, StringBuffer, .v.v..
    - StringBuffer append(type arg) // ==> chú ý (ở trên)
    - StringBuffer insert(int offset, type arg) // ==> chú ý (ở trên)
    -

- // Các method thao tác trên nội dung.
- StringBuffer delete(int start, int end)
- StringBuffer deleteCharAt(int index)
- void setLength(int newSize)
- void setCharAt(int index, char newChar)
- StringBuffer replace(int start, int end, String s)
- StringBuffer reverse()
-
- // Các method trích ra toàn bộ hoặc một phần dữ liệu.
- char charAt(int index)
- String substring(int start)
- String substring(int start, int end)
- String toString()
-
- // Các method tìm kiếm vị trí.
- int indexOf(String searchKey)
- int indexOf(String searchKey, int fromIndex)
- int lastIndexOf(String searchKey)
- int lastIndexOf(String searchKey, int fromIndex)

21. Hướng dẫn xử lí ngoại lệ

- Đây là mô hình sơ đồ phân cấp của Exception trong java.
  - Class ở mức cao nhất là Throwable
  - Hai class con trực tiếp là Error và Exception.
- Trong nhánh Exception có một nhánh con RuntimeException là các ngoại lệ sẽ không được java kiểm tra trong thời điểm biên dịch. Ý nghĩa của được kiểm tra và không được kiểm tra tại thời điểm biên dịch sẽ được minh họa trong các ví dụ phần sau.
- *Các class tùy biến của bạn nên viết thừa kế từ 2 nhánh **Error** hoặc **Exception**, không viết thừa kế trực tiếp từ**Throwable**.*
  - Error
- Khi liên kết động thất bại, hoặc trong máy ảo xảy ra một vấn đề nghiêm trọng, nó sẽ ném ra một Error. Các chương trình Java điển hình không nên bắt lỗi (Error)
  - Exceptions
  - Hầu hết các chương trình ném và bắt các đối tượng là con của class Exception. Trường hợp Exception cho thấy một vấn đề xảy ra nhưng vấn đề

không phải là một vấn đề mang tính hệ thống nghiêm trọng. Hầu hết các chương trình bạn viết sẽ ném và bắt Exception.

- Một class đặc biệt có ý nghĩa trong java là RuntimeException: đại diện cho TH ngoại lệ xảy ra trong thời gian chạy CT
- Checked Exception & Unchecked Exception:

- AgeException là con của Exception, TooOldException và TooYoungException là 2 class con trực tiếp của AgeException, nên chúng là các "Checked Exception"
- Trong method AgeUtils.checkAge(int) có ném ra ngoài các ngoại lệ này vì vậy trên khai báo của method bạn cần phải liệt kê chúng thông qua từ khóa "throws". Hoặc bạn có thể khai báo ném ra ở mức tổng quát hơn
    - psv check(int i) throws Exception.

- Tại các nơi sử dụng AgeUtils.checkAge(int) cũng phải có sử lý để bắt các ngoại lệ đó, hoặc tiếp tục ném ra vòng ngoài. (throw new tooYoung(…));

```
public class AgeException extends Exception {

 public AgeException(String message) {
    super(message);
 }
}
```

- Sau đó ta tiếp tục ném tiếp ra vòng ngoài hoặc SD try catch:
    1. psvm throws Aexception, Bexception{ … staticClass.method(tham so)}
    2. try … catch(Aexception | Bexception e){sout(e.getMessage());} catch{}
- try catch finally:

```
try {

  // Làm gì đó tại đây.
} catch (Exception1 e) {

  // Làm gì đó tại đây.
} catch (Exception2 e) {

  // Làm gì đó tại đây.
} finally {

  // Khối finally luôn luôn được thực thi.
```

```
        // Làm gì đó tại đây.
    }
```

22. generics

- Bạn tạo ra một đối tượng ArrayList với mục đích chỉ chứa các phần tử có kiểu String, tuy nhiên tại nơi nào đó trong chương trình bạn thêm vào danh sách này một phần tử không phải String (Việc này hoàn toàn có thể), khi bạn lấy ra các phần tử đó và ép kiểu về String, một ngoại lệ sẽ bị ném ra.
- Java 5 đưa vào khái niệm Generics. Với sự trợ giúp của Generics, bạn có thể tạo ra một đối tượng ArrayList chỉ cho phép chứa các phần tử có kiểu String, và không cho phép chứa các phần tử có kiểu khác.

```
        // Tạo một ArrayList (Một danh sách)
    // Danh sách này chỉ cho phép chứa các phần tử kiểu String.
    ArrayList<String> userNames = new ArrayList<String>();

    // Thêm các String vào danh sách.
    userNames.add("tom");
    userNames.add("jerry");

    // Bạn không thể thêm các phần tử không phải String vào danh sách.
    // (Lỗi khi biên dịch).
    userNames.add(new Integer(100)); // Compile Error!

    // Bạn không cần phải ép kiểu (cast) của phần tử.
    String userName1 = userNames.get(0);
```

- Kiểu generic cho class và interface
- K, V trong class KeyValue<K,V> được gọi là tham số generics nó là một kiểu tham chiếu nào đó. Khi sử dụng class này bạn phải xác định kiểu tham số cụ thể.
- Thừa kế lớp generics : Một class mở rộng từ một class generics, nó có thể chỉ định rõ kiểu cho tham số generics, giữ nguyên các tham số generics hoặc thêm các tham số generics. Không hỗ trợ exception
- Một phương thức trong class hoặc Interface có thể được generic hóa (generify).

```
        // Lớp này mở rộng (extends) từ lớp KeyValue<K,V>.
        // Xác định rõ kiểu tham số <K> là String.
        // Vẫn giữ kiểu tham số Generic <V>.
```

```
public class StringAndValueEntry<V> extends
KeyValue<String, V> {

   public StringAndValueEntry(String key, V value) {
      super(key, value);
   }
 }

public interface GenericInterface<G> {

  public G doSomething();//ở class con @override lại

  }

// <K,V> : Nói rằng phương thức này có 2 kiểu tham số K,V

   // Phương thức trả về một đối tượng kiểu K.

   public static <K, V> K getKey(KeyValue<K, V> entry) {

      K key = entry.getKey();

      return key;

   }
```

- Phải import java.util.ArrayList;

```
public static <E> E getFirstElement(ArrayList<E> list)
```

- Khởi tạo đối tượng generic

```
// Khởi tạo đối tượng Generic.
T t = new T(); // Error
```

- Việc khởi tạo một đối tượng generic như trên là không được phép, vì <T> không hề tồn tại ở thời điểm chạy của **Java**. Nó chỉ có ý nghĩa với trình biên dịch kiểm soát code của người lập trình. Mọi kiểu <T> đều như nhau nó được hiểu là **Object** tại thời điểm chạy của **Java**.

  Muốn khởi tạo đối tượng generic **<T>** bạn cần cung cấp cho Java đối tượng **Class<T>**, Java sẽ tạo đối tượng **<T>** tại thời điểm runtime bằng **Java Reflection**.

Vd: MyGeneric<Bar> mg = new MyGeneric<Bar>(Bar.class);

Bar bar = mg.getTObject();

bar.currentDate();

- Mảng generic

```
// Bạn có thể khai báo một mảng generic.
T[] myarray;

// Nhưng không thể khởi tạo mảng generic.
// (Điều này không được phép).
T[] myarray = new T[5];  // Error!
```

Lý do là kiểu generic không hề tồn tại tại thời điểm chạy, ***List<String>*** hoặc ***List<Integer>*** đều là ***List***. Generic chỉ có tác dụng với trình biên dịch để kiểm soát code của người lập trình. Điều đó có nghĩa là trình biên dịch của Java cần biết rõ **<T>** là cái gì mới có thể biên dịch (compile) **new T[10];**. Nếu không biết rõ nó mặc định coi T là **Object**. Nếu muốn khởi tạo mảng Generic bạn cần phải truyền cho Java đối tượng Class<T>, giúp Java có thể khởi tạo mảng generic tại thời điểm runtime bằng cách sử dụng Java Reflection. Hãy xem ví dụ:

- Kí tự đại diện

Trong mã Generic, dấu chấm hỏi (?), được gọi là một đại diện (wildcard), nó đại diện cho một loại không rõ ràng. Một kiểu tham số đại diện (wildcard parameterized type) là một trường hợp của kiểu Generic, nơi mà ít nhất một kiểu tham số là wildcard.

Ví dụ của *tham số đại diện (wildcard parameterized)* là ***:***

- *Collection<?>*
- *List<? extends Number>*
- *Comparator<? super String>*
- Pair<String,?>.

Các ký tự đại diện có thể được sử dụng trong một loạt các tình huống: như kiểu của một tham số, trường (field), hoặc biến địa phương; đôi khi như một kiểu trả về (Sẽ được nói rõ hơn trong các ví dụ thực hành). Các đại diện là không bao giờ được sử

dụng như là một đối số cho lời gọi một phương thức Generic, khởi tạo đối tượng class generic, hoặc kiểu cha (supertype).

Các ký hiệu đại diện nằm ở các vị trí khác nhau có ý nghĩa khác nhau:

- Collection<?> mô tả một tập hợp chấp nhận tất cả các loại đối số (chứa mọi kiểu đối tượng).
- List<? extends Number> mô tả một danh sách, nơi mà các phần tử là kiểu Number hoặc kiểu con của Number.
- Comparator<? super String> Mô tả một bộ so sánh (Comparator) mà thông số phải là String hoặc cha của String.

- Một kiểu tham số ký tự đại diện không phải là một loại cụ thể để có thể xuất hiện trong một toán tử new. Nó chỉ là gợi ý các quy tắc thực thi bởi generics java rằng những loại có giá trị trong bất kỳ tình huống cụ thể mà các kí hiệu đại diện đã được sử dụng.
- Một kiểu tham số ký tự đại diện (wildcard parameterized type) không phải là một loại cụ thể, và nó không thể xuất hiện trong một toán tử **new**.

```
// Tham số Wildcard không thể tham gian trong toán tử new.

List<? extends Object> list= new ArrayList<? extends Object>();
```

II. Collection framework

- Khi dùng mảng chi phí thêm, xóa lớn, có số pt cố định, các pt in a row in memory
- Vd với Linklisted

```
import java.util.LinkedList;
        // Tạo một đối tượng LinkedList.
    LinkedList<String> list = new LinkedList<String>();

        // Thêm một số phần tử vào danh sách.
    list.add("F");
        // Thêm phần tử vào cuối danh sách.
    list.addLast("Z");

            // Thêm phần tử vào vị trí đầu tiên của danh sách.
    list.addFirst("A");
```

```
        // Trèn một phần tử vào ví trí có chỉ số 1.
        list.add(1, "A2");

        // Ghi ra tất cả các phần tử của danh sách:
        System.out.println("Original contents of list: " + list);

        // Loại bỏ một phần tử khỏi danh sách
        list.remove("F");

        // Loại bỏ phần tử tại vị trí có chỉ số 2.
        list.remove(2);
        // Loại bỏ phần tử đầu tiên và cuối cùng trong danh sách.
        list.removeFirst();
        list.removeLast();

        // In ra danh sách sau khi đã xóa.
        System.out.println("List after deleting first and last: " + list);

        // Lấy ra phần tử tại chỉ số 2.
        Object val = list.get(2);

        // Sét đặt lại phần tử tại vị trí có chỉ số 2.
        list.set(2, (String) val + " Changed");
        System.out.println("List after change: " + list);
```

- Đặc điểm của linked list:
  - Các phần tử có thể không nằm liên tiếp nhau trong bộ nhớ
  - Nó là 1 lk có tính 2 chiều ở mỗi pt
- Linked list là một đối tượng chứa dl cần quản lí và 2 tham chiếu tới pt link phía trk và sau nó
  -