**1. INTRODUCTION TO JAVA**

- **Java** là ngôn ngữ OOP – lập trình hướng đối tượng.

**Java** ban đầu được **James Gosling** tại Sun Microsystems (sau đó đã được Oracle mua lại) phát triển và được phát hành vào năm **1995** như một thành phần cốt lõi của nền tảng Java của **Sun Microsystems.**

=> Slogan: “**Write one, run anywhere**” - *viết một lần chạy nhiều nơi*, nghĩa là chỉ cần viết một lần trên window chẳng hạn, sau đó vẫn chương trình đó có thể chạy trên Linux, Android, các thiết bị J2ME...

**Giải thích cơ chế:** tất cả lànhờ **JVM** - Java Virtual Machine (máy ảo Java) biên dịch **Bytecode** thành **mã máy** tương ứng.

Quy trình để java có thể "chạy ở mọi nơi" :  
  + Bước 1: Từ file " .**java**" nhờ **java compiler** biên dịch ra file " .**class**" chứa **Bytecode**.

+ Bước 2:  Từ file " .**class**" nhờ **JVM** thông dịch từ **Bytecode** sang **mã máy,** nhờ đó máy hiểu và thực thi.

(**compiler** = trình biên dịch, **Javac** chính là Compiler của java, Javac sẽ dịch **code Java** sang mã **bytecode**, bytecode chính là vị trí trung gian giữa mã nguồn (ở đây là java) và mã máy (machine code). Thông qua **JVM** sẽ chuyển bytecode sang mã máy để chạy chương trình).

Lý do phải dịch qua Bytecode là vì JVM chỉ hiểu Bytecode mà thôi. Nơi nào có JVM thì sẽ được dịch thành mã máy tương ứng (Windows, Mac OS, Linux,...).

**=> Có 4 nền tảng (Java Platform):**

+ **Java SE** (Standard Edition) – Java core (phiên bản chuẩn & cơ bản của Java).

+ **Java EE** (Enterprise Edition) – phát triển web (phiên bản doanh nghiệp).

+ **Java ME** (Micro Edition) – phát triển ứng dụng mobie & các thiết bị nhỏ.

+ **Java FX** – phát triển các ứng dụng lên quan đến mạng.

- **JDK** (Java Development Kit) – bộ công cụ để chạy được 1 ứng dụng Java hoàn chỉnh.

- **JRE** (Java Runtime Environment) – môi trường để thực thi Java.

- **JVM** (Java Virtual Machine) – chuyển từ file.**class** sang **mã máy**.

- **Biên dịch** – dịch toàn bộ file: file **.java ->** file **.class.**

- **Thông dịch** – dịch từng dòng lệnh 1: file **.class -> mã máy.**

- **Kiểu dữ liệu:**

+ **Kiểu nguyên thủy:**

- **Kiểu luận lý** (boolean): 1 = **true**, 0 = **false**.

- **Kiểu số** (số nguyên, số thực):

+ **byte**: 1 byte = 8 bit = 28 -> [-27; 27-1].

+ **short**: 2 byte = 16 bit = 216 -> [-215; 215-1].

+ **int**: 4 byte = 32 bit = 232 -> [-231; 231-1].

+ **long**: 8 byte = 64 bit = 264 -> [-263; 263-1].

+ **char**: 2 byte, bảng mã ASCII (A=65, a=97, a-A=32), mặc định là ‘\u0000’.

+ **float**: 4 byte.

+ **double**: 8 byte.

+ **Kiểu đối tượng:** **String**, **Array** (là 2 lớp có sẵn), **Student**, **Car**.

**Note**: + Kiểu số nguyên mặc định là **int**, kiểu số thực mặc định là **double**.

+ **String** trong Java không phải là một kiểu dữ liệu nguyên thủy mà là một **đối tượng** của lớp class **String**.

- **Khai báo biến:**

+ Kiểu **long** -> có hậu tố l/L.

+ Kiểu **float** -> có hậu tố f/F.

+ Kiểu **double** -> có thể có hoặc k có hậu tố.

- **Toán tử:** trong **Java** không có toán tử = = = (vì đã khai báo kiểu dữ liệu khi khai báo biến).

- **Quy tắc đặt tên:**

+ **Tên package:** **snake\_case**

+ **Tên class: PascalCase**

+ **Tên phương thức, biến:** **camelCase**

+ **Khai báo hằng:** final int **CONST\_VARIABLE\_NAME** = value;

- **Input, Output:**

+ **System.in** là 1 tiêu chuẩn để nhập dữ liệu từ bàn phím.

+ **Note**: nhập số trước -> không cho nhập chuỗi ngay sau đó (vì đang tồn tại bộ nhớ đệm).

=> Xóa bộ nhớ đệm.

- Cách 1: **scanner.nextLine();**

- Cách 2: **double** number = **Double.*parseDouble*(scanner.nextLine());**

- Cách 3: **char** string = **scanner.nextLine().charAt(0);**

- **Một số kí hiệu: %d** (số nguyên), **%f** (số thực), **%c** (kí tự), **%s** (string).

**[So sánh Java & Javascript]**

|  |  |
| --- | --- |
| **Java** | **Javascript** |
| + Giống: Cú pháp tương tự với ngôn ngữ C++.  + Khác: - Đa luồng xử lý.  - Là ngôn ngữ lập trình hướng **đối tượng**.  - Là ngôn ngữ vừa thông dịch vừa biên dịch.  - Thể hiện đầy đủ 4 tính chất.  - Chạy trên đc nhiều nền tảng (có tính độc lập).  - Nhiều ràng buộc, có tính chặt chẽ hơn. | + Khác: - Đơn luồng xử lý.  - Là ngôn ngữ lập trình hướng **sự kiện** (dựa vào đối tượng).  - Là ngôn ngữ **thông dịch** (**JS** có thể chạy, thực hiện các cậu lệnh từng dòng một trên trình duyệt mà không cần phải **compiled**).  - OOP trong **JS** chỉ thể hiện rõ 2 tính chất là tính bao đóng và tính trừu tượng.  - Chỉ chạy đc trên nền tảng có hỗ trợ **HTML**, **CSS**.  - Đây là ngôn ngữ lập trình dễ dãi hơn so với **Java**. |

**[Ưu & nhược điểm của Java]**

- **Ưu điểm:** là ngôn ngữ bậc cao, gần gũi (vì xây dựng các đối tượng tự nhiên), chặt chẽ và đa nền tảng.

- **Nhược điểm:** tốn tài nguyên và thời gian thực thi (vì vừa biên dịch, vừa thông dịch), nhiều ràng buộc, tốn hiệu suất và tính trừu tượng cao.

**2. ARRAY & LOOP STATEMENT**

- **Mảng trong Java:**

+ Các phần tử liên tục nhau trong vùng nhớ.

+ Truy xuất phần tử thông qua chỉ số Index.

+ Kiểu dữ liệu của các phần tử trong mảng phải giống nhau.

+ Không thể co giãn đc vùng nhớ.

- **Khai báo & khởi tạo mảng:**

+ **dataType[ ]** arrayName **= new dataType[**array.length**];**

+ **dataType[ ]** arrayName **= new dataType[ ] {**element 1, element 2,… element n**};**

- Khai báo & khởi tạo cùng lúc: có thể bỏ **new dataType[ ]** ở câu trúc trên.

- Khai báo trước, khởi tạo sau: bắt buộc phải có từ khóa **new**.

+ **dataType** arrayName**[ ]** **= new dataType[ ] {**element 1, element 2,… element n**};**

- Sự khác nhau của 2 cách khai báo trên:

+ i**nt[ ] arr, x;** //chương trình sẽ hiểu **arr** và **x** điều là mảng.

+ **int arr[ ], x;** //chương trình sẽ hiểu **arr** là mảng, còn **x** là biến.

- **[Note!]** ArrayIndexOutOfBoundsException – truy xuất phần tử không hợp lệ.

- **Mảng n chiều:** thực chất là mảng 1 chiều, mà mỗi phần tử của mảng 1 chiều là 1 mảng n-1 chiều khác.

- **for each:** **for (dataType** nameVariable **:** array**) {**

**System.out.println(**nameVariable**);**

**}**

**- Xuất mảng** (sử dụng phương thức có sẵng): **System.out.println(Arrays.toString**(array)**)**;

- **So sánh giá trị 2 chuỗi:** str1**.equals(**str2**)**

(đây là kiểu dữ liệu đối tượng, nên phải sử dụng equals;

còn kiểu dữ liệu nguyên thủy không phải kiểu tham chiếu, nên không hỗ trợ phương thức).

**3. METHOD**

- **Phương thức –** gom những đoạn mã thường xuyên sử dụng lại với nhau => tái sử dụng code, chỉnh sửa code dễ dàng hơn.

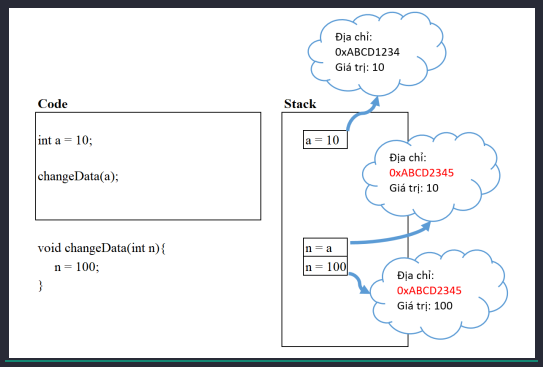
- **Khai báo:**

**access modifier** **static/non-static** **returnType** **methodName(**list parameter**) {**

//method body;

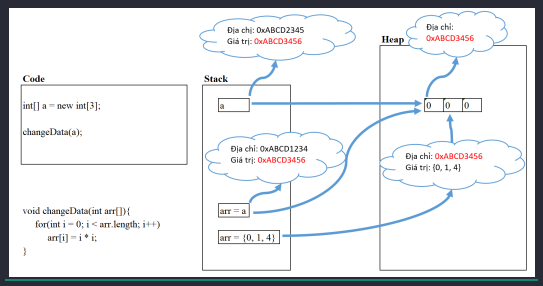
**}**

- **Tham trị:** giá trị sẽ không thay đổi khi ra khỏi phương thức.



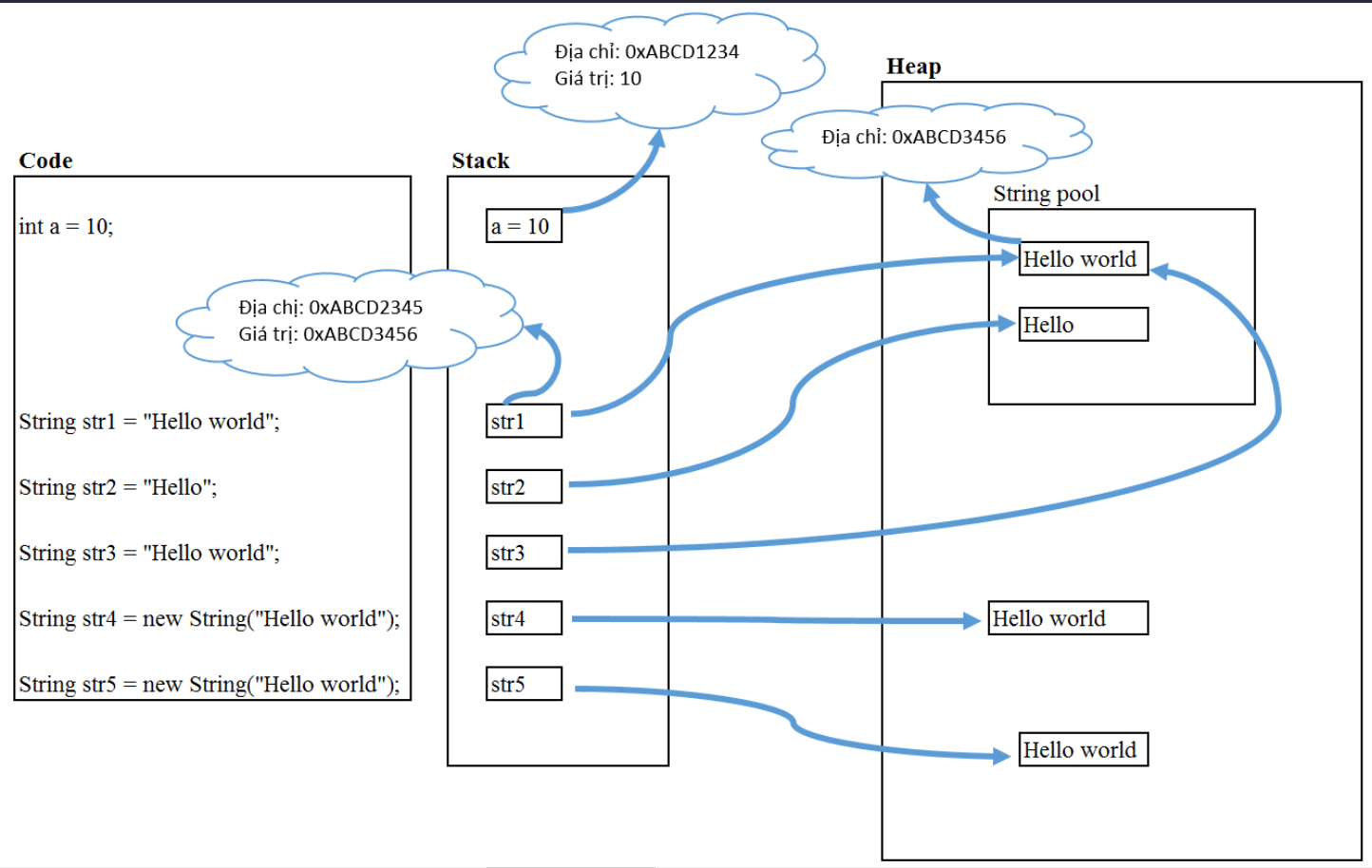
**Ví dụ:** biến **a** có 1 giá trị và địa chỉ đc lưu ở **Stack**, khi gọi đến phương thức **changeData(a)** thì sẽ **nhân bản** ra 1 biến **n = a = 10** ở 1 địa chỉ khác trong **Stack**, và gán **n = 100** ngay tại địa chỉ mới đó (tức là chỉ thao tác với **phân thân** của **a** chứ không làm thay đổi **a**).

- **Tham chiếu:** miền giá trị thay đổi khi ra khỏi phương thức.



**Ví dụ:** mảng **a** sẽ có 1 địa chỉ và giá trị ở **Stack** (giá trị này = địa chỉ **miền giá trị** của **a** ở **Heap**), khi gọi đến phương thức **changeData(a)** thì sẽ **nhân bản** ra 1 mảng **arr** **= a** ở 1 địa chỉ khác trong **Stack** (thực chất của việc nhân bản **arr = a** là giá trị của **arr** đc tham chiếu đến địa chỉ **miền giá trị** của **a** trong **Heap**) và khi khối lệnh thay đổi giá trị các phần tử của **arr** thực hiện nghĩa là ta đang thay đổi giá trị các phần tử trong **miền giá trị** mà **arr** tham chiếu tới (cũng chính là **miền giá trị** của **a**).

**Ví dụ khác:**



+ **Hello world** (đối tượng) là **miền giá trị** của **str1** (biến đối tượng).

+ **str1** (biến đối tượng) đc lưu ở bộ nhớ **Stack,** gồm **Địa chỉ** & **Giá trị**.

+ **Hello world** (đối tượng) đc lưu ở bộ nhớ **Heap**, sẽ có 1 **Địa chỉ**.

**Địa chỉ** này chính là **Giá trị** của **str1** ở bộ nhớ **Stack** (đc **tham chiếu** tới).

**Địa chỉ** này nằm trong **String pool**.

**String pool** chứa Địa chỉ của **đối tượng**. Khi khởi tạo **đối tượng** bằng kí tự (**str1**, **str2**, **str3**), nếu trong **String pool** tồn tại rồi thì **tham chiếu** tới.

+ Khi khởi tạo bằng từ khóa “**new**” (**str4**, **str5**), **đối tượng** sẽ không nằm trong **String pool**.

=> **Khai báo bằng kí tự:** tốc độ chậm, tiết kiệm vùng nhớ.

=> **Khai báo thông qua từ khóa** “**new**”: tốc độ nhanh, không tiết kiệm vùng nhớ.

**[?] Vì sao Java 100% là truyền tham trị?**

Từ các ví dụ trên ta thấy, các **biến nguyên thủy** hay **biến đối tượng** trong **Java** đều có 1 địa chỉ và giá trị đc lưu trong vùng nhớ **Stack** (nếu là **biến đối tượng** thì giá trị này chính là địa chỉ **miền giá trị** của nó trong vùng nhớ **Heap**). Địa chỉ và giá trị này thực chất không hề thay đổi trong xuyên suốt chương trình, ta chỉ thao tác với **bản sao** của các **biến** này (đối với **biến đối tượng**, vì địa chỉ của **miền giá trị** của nó trong vùng nhớ **Heap** là không đổi, nên giá trị của nó trong vùng nhớ **Stack** là không đổi).

**4. OOP, CLASS & OBJECT**

- **OOP – Object Oriented Programing** là ngôn ngữ OOP – là 1 phương pháp, 1 kỹ thuật lập trình ánh xạ đc đối tượng ở thế giới thực vào trong lập trình.

|  |  |
| --- | --- |
| **POP (Procedure Oriented Programing)** | **OOP (Object Oriented Programing)** |
| - Lấy hành động làm trung tâm.  - Tiếp cận từ trên xuống.  - **Data security** (bảo mật dữ liệu) có nguy cơ khi dữ liệu di chuyển tự do trong chương trình, **code reusability** (tái sử dụng code)không đạt được, khiến cho việc lập trình trở nên dài và khó hiểu. Các chương trình lớn dẫn đến nhiều lỗi hơn và nó làm tăng thời gian gỡ lỗi. | - Lấy dữ liệu làm trung tâm.  -Tiếp cận từ dưới lên.  - **Data security** đảm bảo hơn, giải quyết đc vấn đề về **code reusability**. |

- **Các phương pháp lập trình chính:**

+ Lập trình tuyến tính.

+ Lập trình hướng cấu trúc (**POP**).

+ Lập trình hướng đối tượng (**OOP**).

|  |  |
| --- | --- |
| **Class** | **Object** |
| - Là 1 **blueprint** (kế hoạch, bản vẽ thiết kế) hay **prototype** (nguyên mẫu) xác định **properties** và **method** (hay function) chung cho tất cả các đối tượng cùng loại.  - Sử dụng từ khóa “**class**” để khai báo.  - Chỉ đc khai báo 1 lần trong cùng 1 **package**. | - Là 1 **thể hiện** của **class**, là các đối tượng mô tả các thực thể ở thế giới thực.  - Sử dụng khóa “**new**” để khởi tạo.  - Đối tượng có thể đc khởi tạo nhiều lần. |

- **Constructor:**

+ Là **phương thức đặc biệt** giúp tạo ra đối tượng.

+ Không có kiểu dữ liệu trả về.

+ Trùng tên với tên **Class**.

+ Trong 1 class có thể có nhiều **constructor** (**overload** => nạp chồng: trùng tên method, khác kiểu dữ liệu hoặc số lượng tham số).

+ 1 **constructor** có thể gọi 1 và chỉ 1 **constructor** khác, nhưng nó phải đc gọi đầu tiên trong body .

+ Nếu như không tạo ra **constructor** nào thì **Java** hỗ trợ tạo **constructor** mặc định.

- **[Note!]** NullPointerException – xảy ra khi đối tượng bị **null** nhưng cố ý gọi đến phương thức hoặc thuộc tính.

+ **Cách khắc phục 1**: Khởi tạo giá trị khi khai báo các biến đối tượng.

+ **Cách khắc phục 2**: Dùng **if** (nếu đối tượng khác **null** thì mới gọi tới).

+ **Cách khắc phục 3**: Sử dụng đối tượng chắc chắn không bị **null** để **.equals**(đối tượng có thể **null**).

**5. STATIC, ACCESS MODIFIER**

- **Static –** sử dụng để khai báo các phương thức và thuộc tính cho **class**. Biến (hoặc phương thức) **Static** là thuộc về **class**. (nên lấy Class thay vì biến đối tượng để . tới 1 phương thức static nào đó).

=> **Static** là vùng nhớ duy nhất cho tất cả các đối tượng.

=> Khối **static** sẽ chạy trước hàm “**main**”, tạo giá trị cho thuộc tính **static**.

=> Có thể áp dụng từ khóa **static** với thuộc tính (biến của **class**), phương thức, khối **static** và **static** **nested class**.

- **Mục tiêu của static:**

+ Tiết kiệm vùng nhớ.

+ Không cần “**new**” ra đối tượng nhưng vẫn có thể sử dụng phương thức và thuộc tính **static**.

+ Giúp xây dựng những tiện ích (**util**).

- **Ràng buộc khi sử dụng static:**

+ Phương thức **static** chỉ gọi được các phương thức và thuộc tính **static**.

+ Phương thức **non-static** có thể gọi được các phương thức và thuộc tính **static**.

+ Phương thức **static** không thể gọi được các phương thức và thuộc tính **non-static**.

+ Không thể sử dụng từ khóa **this** & **super** cho phương thức **static**.

**Giải thích:**

**non-static** thuộc **đối tượng** (phải tạo ra đối tượng mới có), **static** thuộc **class**.

Khi chạy chương trình -> **static** đc đưa lên đầu chương trình (chạy trước) -> **non-static** chỉ đc gọi khi đối tượng đc tạo ra thông qua từ khóa “**new**”.

**- [Note]** Ban đầu chưa có đối tượng tạo ra nên “**main**” là **static** (nếu “**main**” không có **static** thì vẫn không báo lỗi, nhưng sẽ không chạy đc).

- **2 cách gán giá trị cho biến static:**

+ Gán ngay khi vừa khởi tạo.

+ Gán trong khối **block** { }.

Vd: **static** {

nameVariableStatic = “quần què”;

}

- **Các loại biến:**

+ **Local** (biến cục bộ) – nằm trong phương thức, **contructor**, khối **block** { }.

=> Đc lưu trong **Stack**.

=> Vòng đời: kết thúc khi ra khỏi phương thức.

+ **Instance** (biến đối tượng) – là thuộc tính **non-static**: nằm ngoài **contructor**, khối **block** { }.

=> Biến nguyên thủy đc lưu trong **Stack**, biến tham chiếu đc lưu trong **Heap**.

=> Vòng đời: kết thúc khi đối tượng đc gán bằng **null**.

+ **Static** (biến của class) – đc khai báo thông qua từ khóa **static**: nằm ngoài phương thức, **contructor**, khối **block** { }.

=> Đc lưu trong **Heap** (trong vùng nhớ riêng **Static Area**).

=> Vòng đời: kết thúc khi thoát chương trình.

- **Tính bao đóng:**

+ Là 1 quá trình **gói** mã **code** và dữ liệu lại với nhau thành 1 đơn vị duy nhất.

+ Tất cả thuộc tính phải để chế độ **private** và phải có **getter**/**setter** tương ứng.

+ Giúp **bảo mật thông tin**, chỉ **show** ra những thứ cần thiết, giúp dữ liệu đc bảo mật và chính xác.

+ Tùy chỉnh chỉ đọc (**getter**, ví dụ: usename) hoặc chỉ ghi (**setter**, ví dụ: password).

+ **Kiểm soát** dữ liệu (vd: nhập dữ liệu đầu vào).

- **Access Modifier:** có 4 phạm vi truy cập và 3 từ khóa mô tả phạm vi truy cập.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Phạm vi** | **Public** | **Protected** | **Default** | **Private** |
| Từ trong cùng một lớp (class). | YES | YES | YES | YES |
| Từ bất kì lớp (class) nào trong cùng 1 package. | YES | YES | YES | NO |
| Từ một lớp con (subclass) bên ngoài gói (package). | YES | YES, thông qua kế thừa | NO | NO |
| Từ bất kì một lớp(class) bên ngoài gói (package). | YES | NO | NO | NO |

=> Một **Class** chỉ có thể sử dụng 2 cấp độ kiểm soát truy cập (**default** hoặc **public**), các thành viên có thể sử dụng cả 4 cấp độ kiểm soát truy cập.

- **Nested class** – là một **class** được khai báo bên trong một **class** khác, còn **class** khác này được gọi là **outer class**. Có 2 loại:

+ **Inner class.**

+ **Static nested class.**



**6. INHERITANCE**

- **Tính kế thừa –** cho phép lớp con có thể sử dụng các đặc điểm và hành vi của lớp cha (bản chất là copi code).

- **Mục tiêu:** tái sử dụng mã code.

- **Đặc điểm:**

+ Lớp con không thể kế thừa **constructor** của lớp cha.

=> Gọi **constructor** của lớp cha thông qua “**super**”.

+ Trong **Java** không có **đa kế thừa** (đối với **class**).

=> Vì các lớp cha có thể trùng tên phương thức, chương trình sẽ không hiểu lớp con sẽ kế thừa phương thức nào khi gọi đến.

+ Những tài sản **private** không thể đc kế thừa.

- **super:** là đối tượng giúp hỗ trợ gọi phương thức, thuộc tính và constructor của lớp cha gần nó nhất.

- **final:** được xây dựng trong rất nhiều ngữ cảnh.

+ Nếu đặt ở **biến** => khai báo **HẰNG**.

+ Nếu đặt ở **phương thức** => không cho ghi đè (**override**).

+ Nếu đặt ở **class** => biến **class** đó thành **vô sinh** (không cho **class** khác kế thừa).

- **Object:**

+ Lớp **Object** là cha của tất cả các **class** trong **Java**.

+ **toString( )** – trả về chứa thông tin của đối tượng.

+ **equals( )** – so sánh 2 đối tượng.

- **Tính đa hình** – 1 hành động nhưng có thể thực hiện nhiều cách khác nhau thùy thuộc vào ngữ cảnh.

|  |  |
| --- | --- |
| **Overriding (cài đè)** | **Overloading (nạp chồng)** |
| - Là cơ chế cho phép lớp con định nghĩa lại nội dung các phương thức trước đó ở lớp cha.  - Phương thức phải cùng tên, cùng danh sách tham số, cùng kiểu dữ liệu trả về.  - **Access Modifier** của lớp con phải có lever **lớn hơn hoặc bằng** lớp cha.  - Xảy ra trong **class** có mối quan hệ **is-a**.  - Đa hình tại **runtime**. | - Là cơ chế cho phép 1 lớp có khả năng định nghĩa ra nhiều phương thức cùng tên nhưng khác nhau về tham số truyền vào (kiểu dự liệu, số lượng).  - Xảy ra trong cùng 1 **class**.  - Đa hình tại **compile**. |

**+ Mối quan hệ is-a:** thể hiện qua kế thừa.

Ví dụ: Lớp Dog **extend** Lớp Animal thì Dog **is-a** (là một) Animal,…

**+ Mối quan hệ has-a:** nếu một **class** có **tham chiếu** đến một **thực thể** khác, được gọi là quan hệ **has-a** (Lấy lớp này làm kiểu dữ liệu của lớp kia).

Ví dụ: một **Student** cần có các thông tin như id, name,... và thực thể **Address** chứa các thông tin city, country,... Trong class Student ta khai báo thuộc tính Address address; (bản chất là nhúng đối tượng của lớp này vào trong lớp kia).

- **Ép kiểu** – là việc gán giá trị của một biến có kiểu dữ liệu này tới biến khác có kiểu dữ liệu khác**.**

(có 2 loại: **tường minh** & **ngầm định**).

Vd: ta có cây kế thừa như sau: **OngCoNoi** -> **OngNoi** -> **Cha** -> **Con**

+ **Ngầm định:** thực hiện bởi trình biên dịch, lập trình viên không cần ép kiểu: **con** -> **cha**. (Chuyển kiểu ngược là đối tượng **lớp con** được chuyển về một thành kiểu một **lớp cha**).

**Cha** var1 = **new Con( );** //lớp **Con** đã tự động chuyển thành kiểu lớp **Cha**.

**OngCoNoi** var2 = **new Cha( );** //lớp **Cha** đã tự động chuyển thành kiểu lớp **OngCoNoi**.

+ **Tường minh:** lập trình viên phải chủ động ép kiểu (nếu không sẽ bị lỗi **compile**): **cha** -> **con**. (Chuyển kiểu xuôi, tức một kiểu **cha** chuyển thành kiểu **con**, nó không tự động mà phải viết chính xác lớp cần ép).

**OngNoi** var3 = **new OngNoi( );**

**Con** var4 **= (Con)** var3**;**

- Toán tử **instanceof** trong **java** được sử dụng để kiểm tra một đối tượng có phải là thể hiển của một kiểu dữ liệu cụ thể không (lớp, lớp con, interface).

**Cha** var5 = **new** **Cha( );**

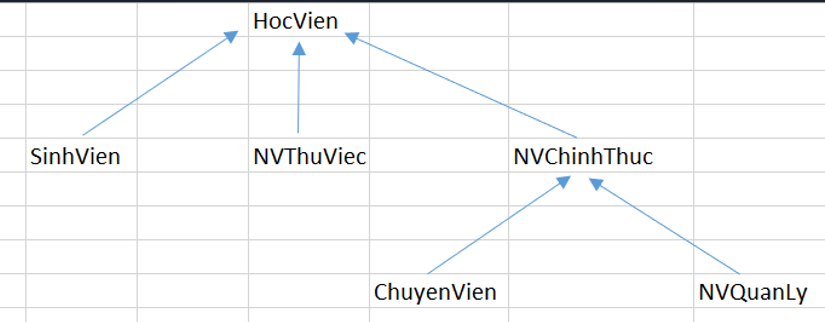
**System.out.println(**var5 **instanceof** **Cha);** // true

=> Một đối tượng có kiểu của lớp con thì cũng có kiểu của lớp cha.

**Con** var6 = **new** **Con( );**

**System.out.println(**var6 **instanceof** **Cha);** // true

**[Bài tập ép kiểu]**

****

1. **HocVien hv1 = new HocVien( );** => **Ok.**

2. **HocVien hv2 = (SinhVien) hv1;**

=> không có lỗi **compile-time**: hv1 đang là kiểu HocVien -> ép sang kiểu SinhVien -> chuyển sang kiểu HocVien (con sang cha).

=> lỗi **run-time**: hv1 là 1 HocVien, không thể ép kiểu sang SinhVien, vì lớp SinhVien sẽ chứa những thuộc tính và phương thức riêng mà lớp HocVien không có, khiến hv1 không thể sử dụng đc.

=> tạm sửa lại: **HocVien hv2 = new SinhVien( );**

3. **ChuyenVien hv3 = (ChuyenVien) hv2;**

=> không có lỗi **compile-time**: hv2 đang là kiểu HocVien -> ép sang kiểu ChuyenVien -> chuyển sang kiểu ChuyenVien.

=> lỗi **run-time**: hv2 là 1 SinhVien, không thể ép kiểu sang ChuyenVien, vì 2 lớp này không kiên quan đến nhau.

=> tạm sửa lại: **ChuyenVien hv3 = new ChuyenVien( );**

4. **NVChinhThuc hv4 = hv3;**

=> **Ok**: hv3 đang là 1 ChuyenVien -> tự động chuyển sang NVChinhThuc (ép kiểu ngầm định con sang cha).

5. **HocVien hv5 = hv3;**

=> **Ok**: như câu 4, ép kiểu ngầm định, đối tượng lớp con chuyển sang lớp cha.

6. **HocVien hv6 = (HocVien) hv2;**

=> **Ok**: hv2 đang là 1 SinhVien, nên ép sang kiểu HocVien đc, trường hợp này vẫn đúng nếu ta bỏ đi “(HocVien)”, không cần ép kiểu vì hv2 đang là kiểu HocVien: **HocVien hv6 = hv2;**

7. **HocVien hv7 = (NVQuanLy) hv6;**

=> không có lỗi **compile-time**: như câu 2.

=> lỗi **run-time**: hv6 đang là kiểu HocVien, không thể ép kiểu sang NVQuanLy, như câu 2.

=> tạm sửa lại: **HocVien hv7 = new NVQuanLy( );**

8. **NVChinhThuc hv8 = (NVChinhThuc) hv7;**

=> **Ok**: hv7 đang là 1 NVQuanLy, nên ép sang NVChinhThuc đc, trường hợp này phải ép kiểu vì hv7 đang là kiểu HocVien.

9. **SinhVien hv9 = (SinhVien) hv3;**

=> lỗi **compile-time**: hv3 đang là kiểu ChuyenVien, không liên quan tới SinhVien.

10. **HocVien hv10 = new SinhVien( );**

=> **Ok**: giống câu 4,5, ép kiểu ngầm định.

11. **ChuyenVien hv11 = (ChuyenVien) hv10;**

=> lỗi **run-time**: giống câu 3, SinhVien không liên quan đến ChuyenVien.

**[Mô hình MVC] phải đảm bảo đủ 4 tính chất:**

- **Tính bao đóng:**

+ Các thuộc tính trong các **Class** phải ở **private**.

+ Có các phương thức **getter**, **setter**, **toString**,…

- **Tính kế thừa:**

+ Các **Class** con **extend** **Class** cha.

+ Các **Interface** con **extend** các **Interface** cha.

+ Các **Class** **implement** các **Interface**.

- **Tính trừu tượng:**

+ **Class** cha phải là **abstract**, có thể có phương thức **abstract**.

+ Các phương thức trong **Interface** phải là **abstract**.

- **Tính đa hình:**

+ Cơ chế **overloading** **Constructor** của các **Class** (**Constructor** không tham số và có tham số).

+ Cơ chế **overriding** phương thức trong các **Class** khi kế thừa các **Interface** hoặc **Class** cha có phương thức **abstract**.

**7. ABSTRACT, INTERFACE**

- **Tính trừu tượng –** chỉ quan tâm đến chức năng, không cần quan tâm đến những xử lí chi tiết.

=> Trong **Java**, tính trừu tượng đc thể hiện qua **abstract class** và **interface**.

- **Abstract class:**

+ Không thể tạo đc đối tượng.

+ 1 **class** có ít nhất 1 phương thức **abstract** thì chắc chắn lớp đó phải là **abstract** (ngược lại thì không).

+ 1 **class abstract** thì có thể chứa phương thức **non-abstract**.

+ **Class abstract** không thể đi kèm với từ khóa **final**.

+ Class abstract con **extends** Class abstract cha thì không cần ghi đè các phương thức **abstract** của cha.

+ Từ khóa **private** không dùng cho phương thức **abstract**.

- **Interface:**

+ Không thể tạo đc đối tượng.

+ Các trường ở trong **interface** đều là hằng số.

+ Tất cả các phương thức đều là **abstract**.

+ Hỗ trợ đa kế thừa (1 **interface** có thể **extends** từ nhiều **interface**).

+ Không tạo đc **contructor**.

**[?]** So sánh **Abstract Class** và **Interface**:

+ Có tạo đc đối tượng không?

=> Cả 2 đều không tạo đc đối tượng.

+ Trường/ thuộc tính?

=> Ở **Interface** đều là hằng số, ở **Abstract Class** thì như Class thông thường.

+ Phương thức (abstract hay non-abstract)?

=> Ở **Interface** đều là abstract, ở **Abstract Class** có thể là abstract hoặc non-abstract.

//trong Java 8, có thể tạo đc phương thức non-abstract trong Interface thông qua default/ static.

+ Có constructor không?

=> **Interface** không có, **Abstract Class** có.

+ Đa kế thừa?

=> **Interface** hỗ trợ đa kế thừa, **Abstract Class** không hỗ trợ đa kế thừa.

**8. CLEAN CODE**

- **Clean code –** là thuật ngữ mô tả những mã nguồn “tốt”.

+ Đơn giản.

+ Dễ hiểu.

+ Dễ nâng cấp/ chỉnh sửa.

+ Định danh (tên biến, method, class, module, project,…) có ý nghĩa.

+ Ít sự phụ thuộc.

+ Phương thức phải thực hiện đúng chức năng, body không đc quá dài.

+ Thể hiện đc ý nghĩa của chương trình.

- **Smell code –** là thuật ngữ mô tả những đoạn code “xấu”.

+ Tên phương thức quá dài.

+ Phương thức thực hiện quá nhiều chức năng.

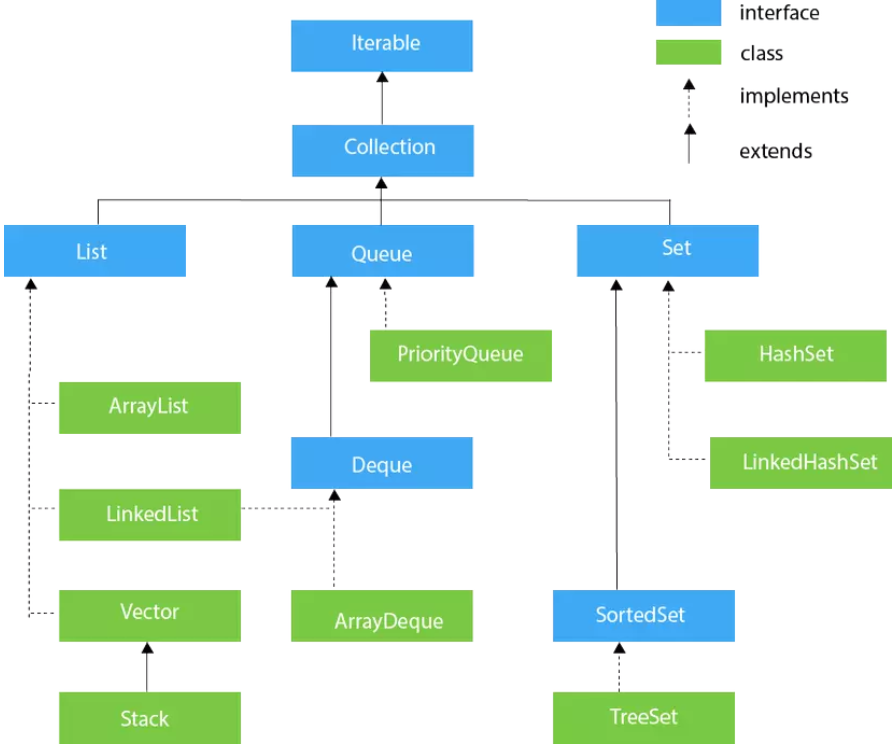
+ Phương thức có quá nhiều tham số.

+ Class có quá nhiều line.

+ Chứa những mã code trùng lặp.

+ Lạm dụng comment.

**10. DSA – LIST**



# - Collection Framework – là 1 khuôn khổ cung cấp 1 kiến trúc (abstract class, interface) để lưu trữ và thao tác với nhóm các đối tượng. Tất cả các hoạt động mà bạn thực hiện trên 1 dữ liệu (tìm kiếm, phân loại, chèn, xóa,..) có thể đc thực hiện bởi Java Collection.

- **Collection** (**bộ sưu tập**) – là tập hợp các đối tượng gần tương tự nhau.

- **Collection Interface** – định nghĩa những phương thức cơ bản khi làm việc với tập hợp, đây là gốc cũng là nền móng để từ đó **xây dựng** lên cả bộ thư viện **Java Collection Framework**.

+ **List** (**danh sách**) – là cấu trúc dữ liệu tuyến tính, trong đó các phần tử đc sắp xếp theo 1 thứ tự xác định, cho phép các phần tử đc trùng lặp nhau.

+ **Set** (**tập hợp**) – là cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử chỉ xuất hiện duy nhất 1 lần (tương tự như tập hợp trong toán học).

+ **Queue** (**hàng đợi**) – là cấu trúc dữ liệu nổi tiếng với kiểu vào ra FIFO (first-in-first-out/ vào trước ra trước).

- **Set và các lớp triển khai:**

+ **HashSet** – các phần tử đc lưu trữ dưới dạng bảng băm, không duy trì thứ tự chèn, không lưu phần tử trùng lặp.

+ **Linked HashSet** – các phần tử đc lưu trữ dưới dạng bảng băm với cấu trúc dữ liệu dạng danh sách liên kết, duy trì thứ tự chèn.

+ **Tree Set** – sử dụng 1 tree cho lưu giữ, mặc định các phần tử đc sắp xếp tăng dần.

- **Array List** – là 1 **class** dạng **list**, **implement** từ mảng có thể thay đổi đc kích thước.

=> **CRUD**: add, get, set, remove, indexOf, valueOf, size, contains,…

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| - Kích thước cố định.  - Chỉ có thuộc tính **length**.  - Có thể lưu kiểu nguyên thủy lẫn đối tượng.  - Tốc độ thao tác nhanh hơn. | - Kích thước có thể thay đổi.  - Có nhiều phương thức hỗ trợ.  - Chỉ cho phép lưu vào kiểu đối tượng. Muốn sử dụng kiểu nguyên thủy thì phải sử dụng **Wrapper class** (lớp **Wrapper** trong java cung cấp cơ chế để chuyển đổi kiểu dữ liệu nguyên thủy thành kiểu đối tượng và ngược lại).  - Tốc độ lưu trữ và thao tác chậm hơn (do chứa nhiều phương thức). |

- **Linked List** – là 1 **class** dạng **list**, lưu trữ data theo cấu trúc danh sách liên kết (đơn, đơn vòng, đối, đôi vòng). Lớp **LinkedList** trong **Java** sử dụng cấu trúc danh sách **liên kết đôi** (Doubly) để lưu trữ các phần tử.

|  |  |
| --- | --- |
| **Array List** | **Linked List** |
| - Sử dụng mảng để lưu trữ.  - Vùng nhớ liên tục nhau.  - Truy xuất nhanh hơn.  - Thêm/xóa chậm hơn.  - Chỉnh sửa phần tử nhanh hơn hoặc bằng (bằng khi cùng sử dụng vòng lặp for). | - Sử dụng danh sách liên kết để lưu trữ.  - Vùng nhớ không cần liên tục nhau.  - Truy xuất chậm hơn.  - Thêm/ xóa nhanh hơn. |

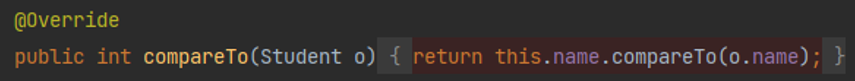
**[Comparable & Comparetor]**

- **Comparable** chỉ sắp xếp theo 1 tiêu chí.

=> **Class** phải **implements** **comparable<E>.**

=> **Override** phương thức **compareTo( ).**

Vd: chỉ sắp xếp học sinh theo tên.



(có thể thay thế **this.name** hoặc **name** bằng phương thức **getName( )**).

- **Sắp xếp tăng dần:**



hoặc 

- **Sắp xếp giảm dần:**



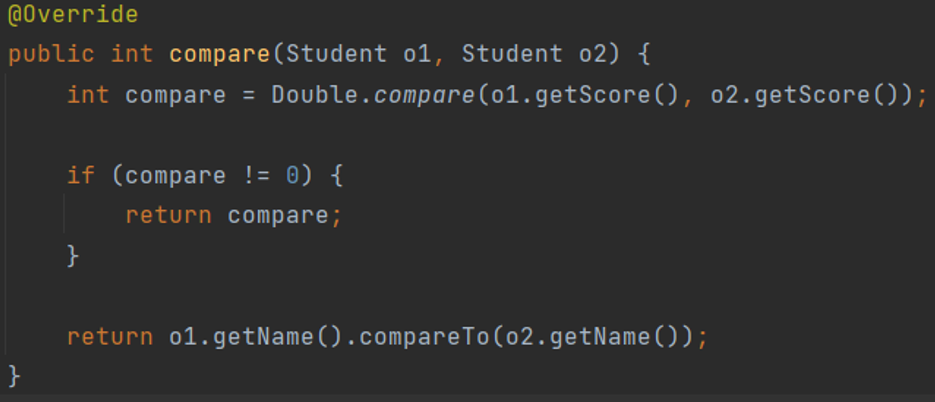
hoặc 

- **Comparator** sắp xếp theo 1 hoặc nhiều tiêu chí (vd: sắp xếp theo tên, điểm, id,...).

=> **Class** phải **implements** **comparator<E>.**

=> **Override** phương thức **compare( ).**

Vd: sắp xếp học sinh theo điểm, nếu bằng điểm thì sắp xếp theo tên.



- **Sắp xếp tăng dần:**



hoặc 

- **Sắp xếp giảm dần:** đổi vị trí **o1** và **o2** trong phương thức **compare**.

**[hashCode( ) & equals( )]**

Trong java, **Object class** mặc định là cha của tất cả các **class**, bên trong nó định nghĩa sẵn 2 method **equals()** và **hashCode()** – đều này có nghĩa rằng tất cả các **class** đều mặc định sẽ có 2 method này.

- **equal( )** – dùng để kiểm tra 2 đối tượng bất kỳ có bằng nhau hay không?

+ Toán tử ‘= =’ được dùng để so sánh địa chỉ 2 đối tượng và giá trị của các biến nguyên thủy. Còn **equals()** được dùng để định nghĩa thế nào là 2 đối tượng trùng nhau, **equals()** chỉ áp dụng cho kiểu đối tượng, không áp dụng cho kiểu nguyên thủy.

+ Giả sử rằng 2 **đối tượng** có các thuộc tính đều bằng nhau, thế nhưng kết quả so sánh từ **equals()** lại trả về **FALSE**?

=> Vì mặc định, **equals()** được triển khai trong **Object class** sẽ so sánh địa chỉ **object** thay vì so sánh giá trị các thuộc tính trong **object**. Hai **đối tượng** này là duy nhất và được cấp phát vùng nhớ riêng trong **heap**.

=> Muốn **equals()** so sánh giá trị các thuộc tính trong **object** thay vì so sánh địa chỉ thì phải **override** lại phương thức **equals().**

- **hashCode( )** – **Mã băm**, là giá trị định danh cho 1 đối tượng, **hashCode** trả về của đối tượng được tạo ra sẽ là **mã băm** của địa chỉ để **gom nhóm** các đối tượng, giúp tăng tốc độ tìm kiếm của chương trình lên từ **30-50** lần.

+ Nếu hai đối tượng bằng nhau thì chúng phải có cùng **hashCode()**.

+ Nếu hai đối tượng có cùng **hashCode()** thì chúng có thể bằng nhau hoặc là không.

- **hashTable( )** – **Bảng băm** là 1 cấu trúc dữ liệu thường được sử dụng như một từ điển: mỗi phần tử trong **bảng băm** là một cặp **key-value**. Nếu so sánh với mảng, **key** được xem như chỉ số (chỉ mục) của mảng, còn **value** giống như giá trị mà ta lưu tại chỉ số tương ứng.

Ở đây, ta hiểu rằng các đối tượng gần giống nhau sẽ được gom nhóm lại với nhau, có **hashCode** bằng nhau (**hashCode** này do thuật toán riêng của **Java** tạo ra), và được lưu trữ ở 1 **bucket** (xô chứa) trong **hashTable** (**hashTable** gồm nhiều xô chứa, địa chỉ của xô chứa chính là **hashCode**), hay xô chứa chính là 1 **list** (danh sách) chứa các đối tượng (phần tử) gần giống nhau đó.

Khi tìm kiếm 1 đối tượng, đối tượng đó sẽ đc băm thành 1 **hashCode**, và chỉ cần duyệt các phần tử trong xô chứa (**list**) có **hashCode = = hashCode** của đối tượng đó là xong, không cần phải duyệt tất cả các phần tử.

**11. DSA – STACK, QUEUE**

- **Generic** – là cơ chế cho phép truyền dữ liệu vào như là tham số (**tham số hóa** kiểu dữ liệu), cho phép sử dụng **method**, **class**, **interface** với nhiều kiểu dữ liệu khác nhau.

+ Không cần phải ép kiểu.

+ Xây dựng đc bài toán tổng quát, tái sử dụng đc mã nguồn.

+ Bắt đc lỗi lúc **compile** (biên dịch).

+ Một số quy tắc đặt tên:

- **E** – element.

- **T** – type.

- **N** – number.

- **K** – key.

- **V** – value.

- **Stack (ngăn xếp)** – là cấu trúc dữ liệu dạng danh sách, thêm và xóa phần tử theo cơ chế FILO.

=> **Ví dụ:**

+ Khi rửa bát đĩa, ta xếp lên chồng, khi sử dụng thì ta lấy những cái ở trên cùng trước.

+ Khi lấy bi ra khỏi lỗ (trong bi-a lỗ), viên nào vào sau thì lấy ra trước.

=> **CRUD:** push, peek, pop, isEmpty, isFull, size,…

- **Queue (hàng đợi)** – là cấu trúc dữ liệu dạng danh sách, thêm và xóa phần tử theo cơ chế FIFO.

=> **Ví dụ:** xếp hàng mua vé; bốc số và chờ gọi đến số thứ tự ở bệnh viện, ngân hàng,…; xe qua trạm thu phí;…

=> **CRUD:** add/offer, element/peek, remove/poll, isEmpty, size,…

=> **Các triển khai của Queue:**

+ **LinkedList** (đã học).

+ **PriorityQueue** – là 1 dạng **Queue** mà trong đó, các phần tử sẽ đc sắp xếp.

+ **ArrayDeque** – là 1 dạng **Deque** (**Queue** 2 chiều), implement **Deque** và **Queue**.

**[Nhược điểm của Generic]**

- Không đc sử dụng cho kiểu nguyên thủy.

- Không sử dụng đc từ khóa **static** (vì chưa xác định đc kiểu dữ liệu nên chưa tạo đc bộ nhớ).

- Không thể ép kiểu để kiểm tra (ví dụ: không ép đc Person -> Student).

- Không khởi tạo đc **instance** (ví dụ: <E> e = new <E>();).

- Không thể **overloading** (Ví dụ: phương thức List<String> và List<Integer> thì nó chỉ hiểu là List mà thôi).

- Không thể khởi tạo mảng <E>.

**12. MAP & TREE**

- **Map** – là một phần của **Java collection framework**.

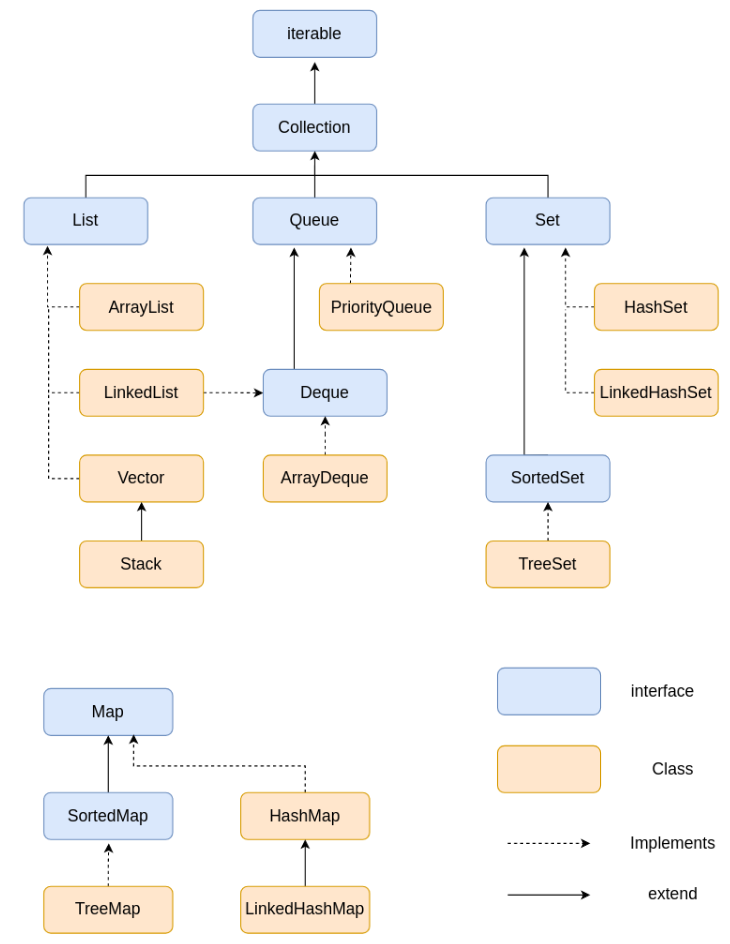
**Map Interface** không kế thừa từ **Collection Interface** mà đây là 1 **interface** độc lập với các phương thức của riêng mình.

Mặc dù các **map** không là **collections** về khái niệm, nhưng chúng hoàn toàn tương thích với **collection**.

+ Lưu trữ và truy xuất dữ liệu thông qua cặp **key** (khóa) – **value** (giá trị).

+ Cặp **key** – **value** đc gọi là **entry**.

+ **Key** không đc phép trùng, **value** có thể trùng.



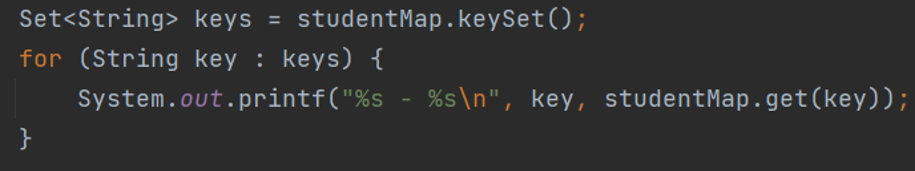
- **HashMap:**

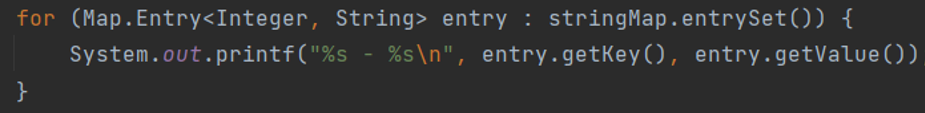
+ Lưu trữ dữ liệu theo cặp **key** – **value**.

+ Có thể chứa tối đa 1 **key** null hoặc nhiều **value** null.

+ Không duy trì thứ tự sắp xếp.

+ Các cách duyệt:





- **LinkedHashMap** (giống **HashMap** nhưng khác 1 số đặc điểm):

+ Duy trì thứ tự thêm vào.

- **TreeMap** (giống **LinkedHashMap** nhưng khác 1 số đặc điểm):

+ Không chứa **key** null.

+ Phần tử đưa vào sẽ đc sắp xếp (mặc định sắp xếp tăng dần theo **key**).

+ Đối với kiểu dữ liệu do người dùng tự định nghĩa thì phải **implement** **Comparable**.

- **Binary Tree:**

+ Hoạt động dựa vào các **node**.

+ Mỗi **node** có tối đa 2 cây con (**left/right sub-tree**).

+ Node trên cùng là **node gốc** (root node), node không có con đc gọi là **node lá** (leaf node), các node có cùng cha là **node anh em** (sibling).

+ **Node** con bên trái có giá trị < **node** cha < **node** con bên phải.

- **Duyệt cây:**

+ **Inorder** (trung thứ tự) – cây con trái -> node cha -> cây con phải.

+ **Postorder** (hậu thứ tự) – cây con trái -> cây con phải -> node cha.

+ **Preorder** (tiền thứ tự) – node cha -> cây con trái -> cây con phải.

+ **Breath-first** – duyệt lần lượt theo từng **level**.

**13. SEARCH ALGORITHM**

- **Linear Search:**

+ Duyệt qua tất cả các phần tử ở trong mảng, tìm thấy thì trả về **index**, không tìm thấy thì trả về **-1**.

+ Sử dụng trong trường hợp: mảng nhỏ, không sắp xếp.

+ Độ phức tạp của thuật toán: tốt nhất là **O(1)**, xấu nhất là **O(n)**.

- **Binary Search:**

+ Mảng đầu vào phải đc sắp xếp tăng giảm.

+ Bước 1: kiểm tra phần cần tìm với phần tử giữa mảng.

Bước 2: nếu phần tử cần tìm = = phần tử giữa mảng => trả về index.

Bước 3: nếu phần tử cần tìm < phần tử giữa mảng => xét mảng con bên trái.

Bước 4: nếu phần tử cần tìm > phần tử giữa mảng => xét mảng con bên phải.

Bước 5: lặp lại Bước 1 -> Bước 4.

+ Độ phức tạp của thuật toán: tốt nhất là **O(1)**, xấu nhất là **O(log2(n))**.

- **Độ phức tạp của thuật toán:** được hiểu là số phép toán thực hiện, phụ thuộc vào tập kết quả đầu vào **n**.

+ **n** đại diện cho số phần tử của mảng trong thuật toán sắp xếp, hoặc tìm kiếm.

+ **n** đại diện cho độ lớn trong trường hợp kiểm tra xem **n** có phải là số nguyên tố hay không.

**14. SORT ALGORITHM**

- **Bubble Sort:**

+ So sánh 2 phần tử liền kề, hoán vị 2 phần tự đó nếu không thỏa mãn điều kiện đang xét.

+ Độ phức tạp: tốt nhất là **O(n)**, xấu nhất là **O(n2)**.

+ Phát hiện đầu vào đã đc sắp xếp.

- **Selection Sort:**

+ Tìm phần tử bé nhất trong mảng, sau đó hoán vị phần tử bé nhất cho số đầu tiên của mảng, xét tiếp mảng từ vị trí 1 -> length-1, tiếp tục cho đến khi mảng đã ddc sắp xếp.

+ Độ phức tạp: tốt nhất/ xấu nhất đều là **O(n2)**.

- **Insertion Sort:**

+ Chèn phần tử vào mảng con đã đc sắp xếp, đảm bảo thứ tự sắp xếp sau khi chèn.

+ Độ phức tạp: tốt nhất là **O(n)**, xấu nhất là **O(n2)**.

**[Thuật toán nào là nhanh nhất?]**

- Trong trường hợp tốt nhất: **Bubble Sort** và **Insertion Sort** là nhanh nhất.

+ **Bubble Sort** phát hiện đc đầu vào đã được sắp xếp (nhờ cờ - biến boolean).

+ **Insertion Sort** sẽ hạn chế đc số vòng lặp nếu mảng con đã đc sắp xếp trước đó.

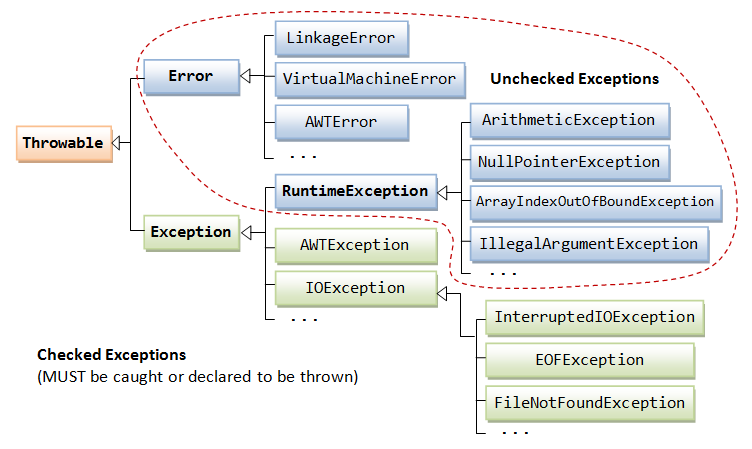
- Trong trường hợp xấu nhất: **Selection Sort** là nhanh nhất. Vì **swap** nằm ngoài vòng lặp **for** trong cùng nên độ phức tạp chỉ là **O(n)** ở cả 2 trường hợp.

**15. EXCEPTION & DEBUG**

- **Exception** – là 1 **sự kiện bất thường**, nó làm phá vỡ **flow** (luồng thực thi) của chương trình và có thể làm chết chương trình.

+ **Class** ở mức cao nhất là **Throwable.**

+ Hai **class** con trực tiếp là **Error** và **Exception**.



- **Checked Exception** – xảy ra tại thời điểm **compile** (có thể gọi là **compile time exceptions**), bắt buột phải xử lí (**handle**).

+ **FileNotFoundException** (một file được chỉ định với đường dẫn không tồn tại, hoặc không có quyền truy cập vào file đó).

+ **IOException** (lỗi không đọc được file).

+ **ClassNotFoundException** (nếu không thể chuyển kiểu object này sang kiểu object khác hoặc không tìm thấy class muốn tham chiếu tới).

+ **DatabaseException**.

+ **NoSuchFieldException**.

- **Unchecked Exception** – xảy ra lúc **runtime**, không chắc chắn xảy ra, không bắt buộc phải **handle** tại thời điểm **compile**.

+ **NullPointerException** (được ném ra khi chương trình cố gắng tham chiếu tới một đối tượng nhưng nó không có vị trí nào trên bộ nhớ, tức là có giá trị null).

+ **ArrayIndexOutOfBoundsException** (truy nhập mảng với chỉ số không hợp lệ).

+ **ArithmeticException** (xảy ra khi chia cho số 0).

+ **NumberFormatException** (xảy ra khi cố gắng để chuyển đổi một chuỗi thành một số, xuất hiện khi khởi tạo biến bằng Integer.parseInt(scanner.nextLine()) hoặc Double.parseDouble(scanner.nextLine())).

+ **InputMismatchException** (xảy ra khi cố gắng để chuyển đổi một chuỗi thành một số, xuất hiện khi khởi tạo biến bằng scanner. .nextInt() hoặc scanner. .nextDouble ()).

- **Error** – lỗi do môi trường thực thi (**JVM**), không thể **handle** + chương trình sẽ chết. Lớp **Error** định nghĩa các ngoại lệ mà không thể bắt (**catch**) từ chương trình.

+ **StackOverflowError** (tràn vùng nhớ stack, stack đạt đến giới hạn tối đa).

+ **VirtualMachineError** (được ném ra khi máy ảo Java gặp lỗi nội bộ hoặc giới hạn tài nguyên, điều này khiến máy không thể hoạt động. Đây là một cơ chế tự bảo vệ được JVM sử dụng để ngăn toàn bộ ứng dụng bị treo).

+ **OutOfMemoryError** (khi hết bộ nhớ trong heap, chủ yếu khi bạn cố gắng tạo một object và không có đủ không gian trên heap để phân bổ đối tượng đó).

- **Cách xử lý:** 2 cách:

+ **throw/throws** – ném ngoại lệ cho phương thức khác xử lí.

+ **try – catch**: xử lí ngay tại nơi xảy ra **exception**.

- Tại một thời điểm chỉ xảy ra một ngoại lệ và tại một thời điểm chỉ có một khối **catch** được thực thi. Khi **exception** đã bị bắt ở một catch thì các catch tiếp theo sẽ không được bắt.

- Tất cả các khối **catch** phải được sắp xếp từ cụ thể nhất đến chung nhất (**từ exception con đến exception cha**), tức là phải khai báo khối lệnh catch để xử lý lỗi **NullPointerException**, **ArithmeticException**, … trước khi khai báo catch để xử lý lỗi **Exception**.

- Khối lệnh **finally** luôn được thực thi dù chương trình có xảy ra ngoại lệ hay không (ngay cả sử dụng lệnh **return**).

- Đối với mỗi khối **try**, có thể không có hoặc nhiều khối **catch**, nhưng chỉ có một khối **finally**.

- Khối **finally** sẽ không được thực thi nếu chương trình bị thoát bằng cách gọi **System.exit()** hoặc xảy ra một lỗi (**Error**) không thể tránh khiến chương trình bị chết.

**16. IO: TEXT FILE**

- **Stream** – là hoạt động nhập xuất dữ liệu. Có 2 loại:

+ **Character Stream** – hỗ trợ nhập xuất dữ liệu cho **Unicode**.

+ **Byte Stream** – hỗ trợ nhập xuất dữ liệu cho **Byte** (nhập xuất theo nhị phân).

**17. IO: BINARY FILE & SERIALIZATION**

- **Serialization** – là cơ chế cho phép chuyển đổi từ đối tượng (**object**) sang **Byte Stream**, ngược lại **Deserialization** là cơ chế chuyển từ **Byte Stream** sang **Object**.

+ Trong **Java**, để 1 **Object** có thể **Serialization** thì class đó phải **implement** **Interface** **Serializable**.

+ Khi lớp cha đã **implement** **Serializable** thì lớp con không cần **implement** lại **Serializable**.

+ Từ khóa **transient** giúp cho thuộc tính không bị **Serializable**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Abstract Class** | **Class** hay dùng | **Class** đọc/ghi **object** | **Class** tạo bộ nhớ đệm |
| InputStream  OutputStream | FileInputStream FileOutputStream | ObjectInputStream ObjectOutputStream | BufferedInputStream BufferedOutputStream |

**19. STRING & REGEX**

- **String** – là 1 lớp đc định nghĩa sẵn trong Java, sử dụng để lưu trữ/ làm việc với chuỗi trong Java.

+ Có 2 cách để gán biến **String**:

- Gán trực tiếp cho biến (giống kiểu dữ liệu nguyên thủy).

- Thông qua từ khóa “**new**”.

=> **String** là kiểu dữ liệu đặc biệt, có tình chất vừa nguyên thủy, vừa đối tượng.

+ String là không thể thay đổi, bất biến (**immutable**).

+ Một số phương thức hay dùng: concat, length, subString, split, trim, charAt,..

- **StringBuilder**/**StringBuffer** – mô tả các dữ liệu dạng chuỗi có thể sửa đổi linh động (**mutable**).

+ Khác về đồng bộ:

- **StringBuffer**: hỗ trợ đồng bộ (**synchronization**).

- **StringBuilder**: không hỗ trợ đồng bộ.

+ Một số phương thức hay dùng: **append** (đây là phương thức quan trọng nhất), insert, delete, reverse, toString,…

- **Regular Expression** (**Regex**) – là 1 chuỗi được sử dụng để truy định dạng thức của 1 chuỗi khác, đc sử dụng cho **validata** (xác thực) dữ liệu và tìm kiếm.

|  |  |  |
| --- | --- | --- |
| **STT** | **Biểu thức chính quy** | **Mô tả** |
| 1 | **.** | Khớp (match) với bất kỳ ký tự nào |
| 2 | **^regex** | Biểu thức chính quy phải  khớp tại điểm bắt đầu |
| 3 | **regex$** | Biểu thức chính quy phải khớp ở cuối dòng. |
| 4 | **[abc]** | Thiết lập định nghĩa, có thể khớp với a hoặc b hoặc c. |
| 5 | **[abc][vz]** | Thiết lập định nghĩa, có thể khớp với a hoặc b hoặc c theo sau là v hay z. |
| 6 | **[^abc]** | Khi dấu ^ xuất hiện như là nhân vật đầu tiên trong dấu ngoặc vuông, nó phủ nhận mô hình. Điều này có thể khớp với bất kỳ ký tự nào ngoại trừ a hoặc b hoặc c. |
| 7 | **[a-d1-7]** | Phạm vi: phù hợp với một chuỗi giữa a và điểm d và con số từ 1 đến 7. |
| 8 | **X|Z** | Tìm X hoặc Z. |
| 9 | **XZ** | Tìm X và theo sau là Z. |
| 10 | **$** | Kiểm tra kết thúc dòng. |
| 11 | **\d** | Số bất kỳ, viết ngắn gọn cho [0-9] |
| 12 | **\D** | Ký tự không phải là số, viết ngắn gon cho [^0-9] |
| 13 | **\s** | Ký tự khoảng trắng, viết ngắn gọn cho [ \t\n\x0b\r\f] |
| 14 | **\S** | Ký tự không phải khoản trắng, viết ngắn gọn cho [^\s] |
| 15 | **\w** | Ký tự chữ, viết ngắn gọn cho [a-zA-Z\_0-9] |
| 16 | **\W** | Ký tự không phải chữ, viết ngắn gọn cho [^\w] |
| 17 | **\S+** | Một số ký tự không phải khoảng trắng (Một hoặc nhiều) |
| 18 | **\b** | Ký tự thuộc a-z hoặc A-Z hoặc 0-9 hoặc \_, viết ngắn gọn cho [a-zA-Z0-9\_]. |
| 19 | **\*** | Xuất hiện 0 hoặc nhiều lần, viết ngắn gọn cho {0,} |
| 20 | **+** | Xuất hiện 1 hoặc nhiều lần, viết ngắn gọn cho {1,} |
| 21 | **?** | Xuất hiện 0 hoặc 1 lần, ? viết ngắn gọn cho {0,1}. |
| 22 | **{X}** | Xuất hiện X lần, {} |
| 23 | **{X,Y}** | Xuất hiện trong khoảng X tới Y lần. |
| 24 | **\*?** | \* có nghĩa là xuất hiện 0 hoặc nhiều lần, thêm ? phía sau nghĩa là tìm kiếm khớp nhỏ nhất. |

**20. S.O.L.I.D**

- **SOLID** – là 1 trong những nguyên tắc (chỉ dẫn) để giúp chúng ta xây dựng đc các ứng dụng **OOP** hiệu quả, giúp lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain (bảo trì).

- **S** – **Single responsibility principle** – Nguyên lý Trách nhiệm Duy nhất.

+ Mỗi lớp chỉ nên đảm nhiệm 1 nhiệm vụ duy nhất.

=> Dễ quản lí mã nguồn.

=> Các lớp tập trung vào nhiệm vụ của mình.

=> Giảm tính phụ thuộc giữa các thành phần.

=> Có thể phát triển đồng thời các lớp độc lập với nhau.

=> Dễ dàng mở rộng.

=> Dễ dàng bảo trì.

- **O** – **Open closed principle** – Nguyên lý đóng mở.

+ Có thể thoải mái mở rộng 1 **class**, nhưng không đc sửa lỗi bên trong **class** đó.

(Các đối tượng (hoặc thực thể) nên mở đối với việc mở rộng, nhưng đóng đối với việc thay đổi).

=> Dễ mở rộng.

=> Dễ thay đổi.

- **L** – **Liskov substitution principle** – Nguyên lý thay thế Liskov.

+ Trong 1 chương trình, các **object** của **class con** có thể thay thế **class cha** mà không làm thay đổi tính đúng đắn của chương trình.

=> Tránh sai sót khi mở rộng thiết kế.

- **I** – **Interface segregation principle** – Nguyên lý phân tách Interface.

+ Thay vì dùng 1 **Interface** lớn, nên tách thành nhiều **Interface** nhỏ với nhiều mục đích cụ thể.

(nghĩa là không nên bắt buộc phải triển khai một **Interface** nếu không cần đến nó, cũng không nên bắt buộc phải phụ thuộc vào các phương thức mà không cần đến chúng).

=> Tránh dư thừa mã nguồn không cần thiết.

=> Dễ quản lí hơn.

- **D** – **Dependency inversion principle** – Nguyên lý đảo ngược phụ thuộc.

+ Các **module** cấp cao không nên phụ thuộc vào các **module** cấp thấp, cả 2 nên phụ thuộc vào **abstraction**.

+ **Interface** (**abstraction**) không nên phụ thuộc vào chi tiết, mà chi tiết nên phụ thuộc vào trừu tượng.

**21. DESIGN PATTERN**

- Là 1 giải pháp tổng thể cho các vấn đề chung trong thiết kế phần mềm. DP không phải là Ngôn ngữ lập trình, được sử dụng phổ biến trong OOP. Gồm 3 nhóm:

- **Creational Design Pattern (khởi tạo)** – có 5DP, giải quyết các vần đề liên quan đến khởi tạo đối tượng.

+ Singleton.

+ Factory.

- **Structoral Design Pattern (cấu trúc)** – có 7DP, thiết kế các thành phần của lớp và đối tượng.

+ Proxy.

+ Façade.

- **Behavioral Design Pattern (hành vi/ tương tác)** – có 11DP, giải quyết các vấn đề trong giao tiếp giữa các đối tượng.

+ Observer.

+ Strategy.

**The Last Shinobi…**