# 

[**Tuần 1: Kiến thức cơ bản về java core**](#_5gl8f44qvxpa)

[**Ngày 1: Kiến thức cơ bản ngôn ngữ lập trình java 1**](#_87gc91ryr4ju)

[I. Tổng quan 1](#_dabe8f5vos9d)

[II. Đặc điểm chung 1](#_n90vu4tghwyt)

[III. Đặc điểm của các tính chất trong lập trình hướng đối tượng (OOP) 2](#_q5amzjcrtzeh)

[IV. JDK, JRE, JVM 6](#_c69sq5dk1fw6)

[V. Biến kiểu dữ liệu: Chia ra làm 2 kiểu dữ liệu 6](#_u7nuauiqjd7p)

[VI. Câu lệnh điều khiển (Control Statement) 9](#_9f31dt1cut7s)

[VII. Lớp và đối tượng 16](#_mxh7haf7dlx2)

[**Ngày 2: Exception 18**](#_eqsf90suxs3x)

[I. Tổng quan 18](#_o3hxd9codddl)

[II. Phân loại: 2 loại ngoại lệ chính 19](#_8r1q3f9n9qe9)

[III. Cách để custom một exception 19](#_cyn3hnefdyp5)

[IV. Lớp java.lang.Exception 19](#_j0i2hwirjmyd)

[V. Cách thức xử lý ngoại lệ (Exception Handling) 20](#_t2mywzob9k40)

[VI. List, Set, Map 22](#_atix0gtxix12)

[VII. Iterators và các thao tác duyệt Collection 32](#_35purumgmnvi)

[**Ngày 3: Collection Framework 34**](#_wtm1je7bj7xl)

[I. Generics 34](#_yukc3r2aq28)

[II. Lambda Expressions 36](#_xz0fqmgp82qz)

[III. Streams API 37](#_60jimyojs7pl)

[**Ngày 4: Làm việc với File 39**](#_rwninab6ughv)

[I. Thread 39](#_xyns5m4c4xh)

[II. IO File 42](#_u9dw77yw2gs2)

[III. Serialization và Deserialization 44](#_ejwxa3l6e0k0)

# **Tuần 1: Kiến thức cơ bản về java core**

## **Ngày 1: Kiến thức cơ bản ngôn ngữ lập trình java**

### **Tổng quan**

* Java là ngôn ngữ đóng vai trò rất quan trọng trong khoảng cuối thế kỉ 20, đầu thế kỉ 21. Nó được coi là ngôn ngữ mang tính cách mạng và có thể chạy thống nhất trên mọi nền tảng mà chỉ cần biên dịch một lần.

### **Đặc điểm chung**

#### **Máy ảo Java (JVM: Java Virtual Machine)**

* Kiến trúc CPU của mỗi hệ thống máy tính khác nhau. Vì vậy, khi phát triển một ứng dụng nào đó trên mỗi kiến trúc CPU và hệ điều hành khác nhau, chương trình sẽ được biên dịch thành những file có đuôi khác nhau. Ví dụ: hệ điều hành window khi biên dịch có đuôi file là \*.exe, hệ điều hành linux khi biên dịch có đuôi file là \*.elf. Do đó mà khi mỗi lần chạy ứng dụng trên các hệ điều hành và kiến trúc CPU khác nhau thì đều cần phải sửa đổi và biên dịch lại.
* Từ khi có JVM: máy ảo java ra đời mà vấn đề này đã được giải quyết. Khi muốn chạy một chương trình ứng dụng viết bằng ngôn ngữ java, chương trình đó sẽ được biên dịch thành mã của máy ảo java (java bytecode), sau đó sẽ được JVM thông dịch thành mã máy (machine code). Sun Microsystems chịu trách nhiệm phát triển những máy ảo java cho những kiến trúc CPU và hệ điều hành khác nhau.

#### **Độc lập nền, khả chuyển (portable)**

* Ngôn ngữ java có thể chạy trên bất kỳ hệ điều hành và máy tính nào chỉ cần ở đó có   
  máy ảo java. Với khẩu hiệu: Viết một lần, chạy ở bất cứ nơi đâu (Write once, run anywhere).

#### **Vừa biên dịch, vừa thông dịch:**

* Những chương trình viết bằng ngôn ngữ java (file có đuôi \*.java) đầu tiên sẽ được trình biên dịch biên dịch thành file có đuôi \*.class, sau đó sẽ được thông dịch thành mã máy.

#### **Đa nhiệm, đa luồng**

* Ngôn ngữ java hỗ trợ cho việc lập trình đa nhiệm, đa luồng, cho phép nhiều tiến trình hoạt động, chạy song song cùng một thời điểm và tương tác với nhau.

#### **Hỗ trợ mạnh cho việc phát triển các ứng dụng khác nhau:**

* Ngôn ngữ java phát triển mạnh mẽ nhờ vào Sun Microsystems cung cấp nhiều thư viện phong phú và công cụ hữu ích giúp cho việc phát triển nhiều loại hình ứng dụng khác nhau:
* J2SE (Java 2 Standard Edition): hỗ trợ phát triển các ứng dụng đơn, ứng dụng client-server.
* J2EE(Java 2 Enterprise Edition): hỗ trợ việc phát triển các ứng dụng thương mại.
* J2ME (Java 2 Micro Edition): hỗ trợ việc phát triển các ứng dụng điện thoại, không dây.

#### **Garbage Collection:**

* Cơ chế thu gom rác tự động, khi có những biến không hoạt động trong chương trình.

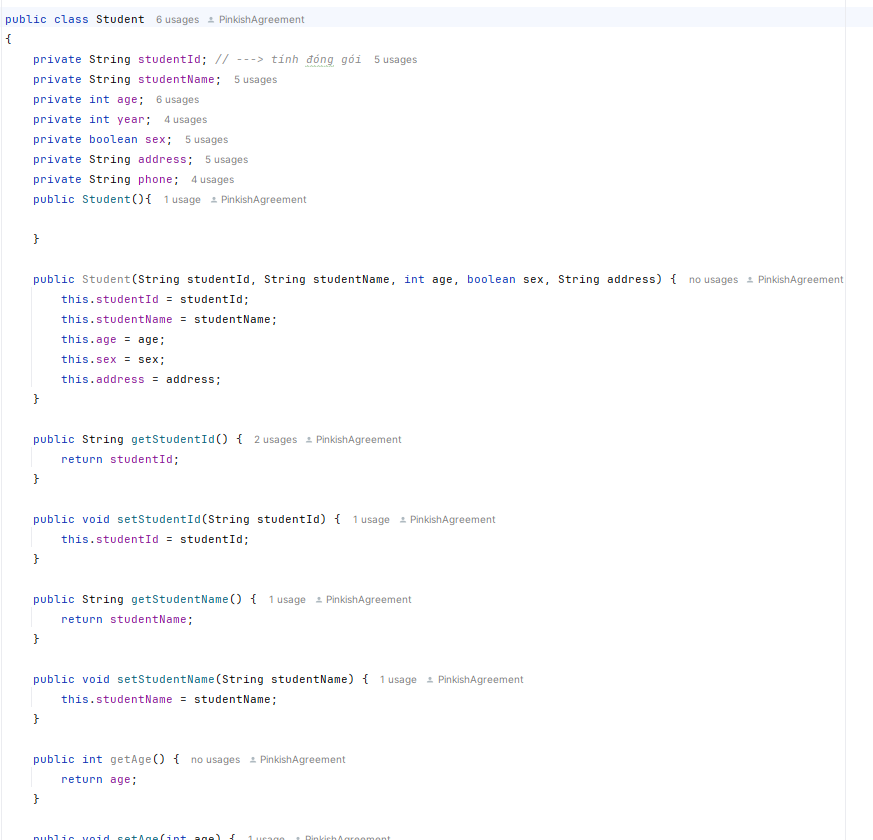
#### **Là ngôn ngữ lập trình hướng đối tượng (OOP)**

* Là ngôn ngữ sử dụng đối tượng để thao tác và biểu diễn dữ liệu. Nó cho phép giải quyết các vấn đề phức tạp bằng cách chia nhỏ các vấn đề thành các đối tượng nhỏ hơn và dễ quản lý hơn.
* Trong OOP, dữ liệu sẽ được đóng gói thành đối tượng và đối tượng sẽ có các thuộc tính (attributes) và hành vi (methods).

### **Đặc điểm của các tính chất trong lập trình hướng đối tượng (OOP)**

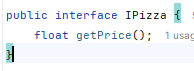
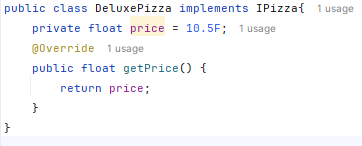
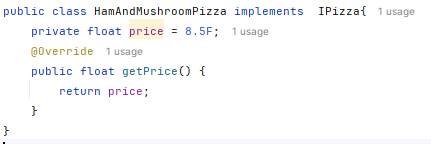
#### **Tính đóng gói (Encapsulation)**

* Là thuộc tính cho phép che giấu thông tin của đối tượng và chỉ cho phép những phương thức được định nghĩa trong lớp đó mới có thể truy cập những thuộc tính và phương thức đó. Nó giúp che giấu và bảo vệ dữ liệu bên trong khỏi sự truy cập từ bên ngoài. Điều này cũng giúp giảm thiểu sự phụ thuộc giữa các đối tượng và tăng tính bảo mật cho chương trình.
* Tính đóng gói được biểu hiện thông qua các từ khóa truy cập (access modifier), điều khiển khả năng nhìn thấy của biến và phương thức trong một lớp. Các từ khóa truy cập trong java bao gồm:
* public: các biến và phương thức công khai (public) có thể truy cập từ bất cứ đâu.
* private: các biến và phương thức riêng tư (private) có thể truy cập từ bên trong lớp mà chúng được định nghĩa.
* protected: các biến và phương thức được bảo vệ (protected) có thể truy cập từ bên trong lớp, package và các lớp con của chúng.
* VD: các đặc điểm sinh viên đóng gói trong lớp Student



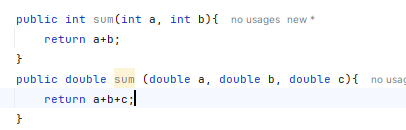
#### **Tính kế thừa (Inheritance)**

* Là thuộc tính cho phép xây dựng lớp mới dựa trên cơ sở của lớp hiện tại. Lớp mới con sẽ kế thừa các thuộc tính và phương thức của lớp hiện tại (lớp cha). Tính kế thừa giúp làm giảm thời gian và nỗ lực trong việc lập trình, tăng tính linh hoạt, tái sử dụng và giảm thiểu lỗi lặp lại.
* Tính kế thừa đạt được thông qua từ khóa “extends”. Ngoài ra, có thể đạt được tính kế thừa thông qua interface.
* VD:



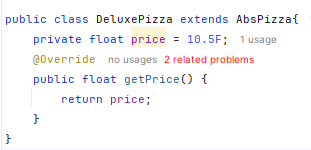
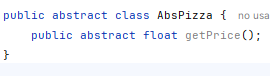
#### **Tính đa hình (Polymorphism)**

* Là thuộc tính mà các đối tượng có khả năng thực hiện các hành vi khác nhau khi được gọi cùng một phương thức. Nó cho phép các đối tượng đáp ứng theo các cách khác nhau với cùng một phương thức. Tính đa hình giúp làm giảm sự phức tạp của mã lập trình, tăng tính linh hoạt và giảm thiểu lỗi.
* Tính đa hình được biểu hiện thông qua phương thức ghi đè (override) và phương thức nạp chồng (overload):
* Phương thức ghi đè (override): cho phép lớp mới con thay đổi lại phương thức của lớp cũ cha. Khi gọi phương thức từ đối tượng lớp con, chương trình sẽ thực thi phương thức mới của lớp con thay vì phương thức cũ của lớp cha.
* Phương thức nạp chồng (overload): cùng một phương thức có thể đáp ứng theo các cách khác nhau tùy thuộc vào ngữ cảnh nó được gọi.
* Đối tượng đa hình (Object Polymorphism): Biến thuộc lớp cha có thể tham chiếu đến đối tượng của lớp con, từ đó biến thuộc lớp cha cũng có thể đáp ứng với những phương thức của lớp con.
* VD:



#### **Tính trừu tượng (Abstraction)**

* Là thuộc tính cho phép tạo ra các lớp trừu tượng mà không cần cung cấp triển khai cụ thể cho các phương thức của chúng. Tính trừu tượng giúp tăng tính linh hoạt của chương trình khi mà có thể thay đổi, thêm mới các phương thức mà không ảnh hưởng đến các lớp khác. Hơn nữa, nó còn làm giảm sự phức tạp của mã lập trình khi chỉ quan tâm đến các hành động cần thiết của đối tượng mà không cần quan tâm đến chi tiết cài đặt cụ thể.
* Có thể đạt được tính trừu tượng thông qua:
* Lớp trừu tượng (abstract class): từ 0 đến 100%.
* Interface: đến 100%.



| **Abstract Class** | **Interface** |
| --- | --- |
| Có thể có các phương thức abstract, non abstract | - Phiên bản java < 8, chỉ có các phương thức abstract.  - Phiên bản java 8, có thêm các phương thức default và static.  - Phiên bản java 9, có thêm phương thức private |
| Có thể có các biến static, non static, final, non final | Chỉ có các biến static final |
| Có thể có các phương thức static, constructor, main | Không thể có phương thức static, constructor và main |
| Không hỗ trợ đa kế thừa | Hỗ trợ đa kế thừa |
| Cung cấp các trình triển khai của interface | Interface không cung cấp bất cứ trình triển khai nào của 1 abstract class |
| Từ khóa abstract để khai báo một abstract class | Từ khóa interface để khai báo 1 interface |
| Chỉ sử dụng abstract class khi chỉ có thể hoàn thành một vài chức năng (methods) và những chức năng còn lại những class extends phải hoàn thành. Những chức năng hoàn thành có thể sử dụng như những tính năng chung. | Sử dụng interface khi muốn xây dựng một bộ khung bao gồm các chức năng mà các module cần phải có. Khi implement phải triển khai tất cả các phương thức mà đã được định nghĩa trong interface. |

### **JDK, JRE, JVM**

#### **JDK (Java Development Toolkit)**

* là bộ công cụ của java hỗ trợ cho quá trình biên dịch chương trình. Bộ công cụ này chứa những công cụ cần thiết để hỗ trợ cho quá trình biên dịch
* Compiler: Công cụ biên dịch của chương trình.
* Debugger: Công cụ hỗ trợ cho việc phát hiện và sửa lỗi.
* Other Libraries: Hỗ trợ cho việc khởi chạy các class (classloader) và kết nối đến các thư viện.

#### **JRE (Java Runtime Environment)**

* Là môi trường do máy ảo java tạo ra, là môi trường lý tưởng nhất cho java bytecode hoạt động chính xác nhất.

#### **JVM (Java Virtual Machine)**

* Là máy ảo java, giúp tạo ra môi trường cho java bytecode hoạt động hiệu quả nhất.

### **Biến kiểu dữ liệu: Chia ra làm 2 kiểu dữ liệu**

| **kiểu tham trị (primitive type)** | **kiểu tham chiếu (reference type)** |
| --- | --- |
| Character: char (1 byte - 8 bit) | Array |
| Boolean: boolean () | Object |
| Float: float (4 byte - 32 bit), double (8 byte - 64 bit) | Class |
| Integer: byte, short (2 byte - 16 bit), int (4 byte - 32 bit), long (8 byte - 64 bit) | Enum |
|  | Interface |
|  | String |
|  | Annotation |

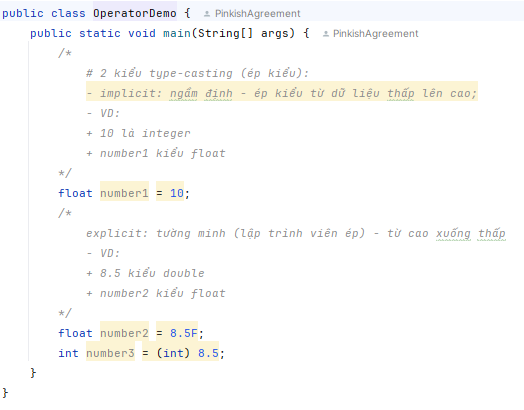
| **Code Segment** | **Data Segment** | **Stack Segment** | **Heap Segment** |
| --- | --- | --- | --- |
| - Dùng để lưu trữ mã nguồn của chương trình (machine code) | - Dùng để lưu trữ những thành phần chung của chương trình như static, constant sinh ra khi chương trình bắt đầu thực thi và giải phóng khi chương trình kết thúc | - Đây là bộ nhớ cố định về không gian và vị trí trên ổ đĩa | - Đây là bộ nhớ phân bổ cấu trúc dữ liệu có kích thước động. |
| - Dùng để lưu trữ những biến thuộc kiểu tham trị, lời gọi hàm và địa chỉ ô nhớ của những biến thuộc kiểu tham chiếu | - Dùng để lưu trữ những biến thuộc kiểu tham chiếu. |
| - Quản lý bởi hệ điều hành | - Quản lý bởi lập trình viên |
| - Kích thước bộ nhớ được biết trước | - Bộ nhớ không đủ sẽ được mở rộng ra bởi cơ chế của hệ điều hành khi không đủ vùng nhớ do được cấp phát động |
| - Biến lưu trữ trong stack sẽ giải phóng khi kết thúc một phương thức hoặc block - scope. | - Biến lưu trữ trong heap sẽ kết thúc, giải phóng trong runtime |
| - Stack không đủ sẽ dẫn đến stack overflow, trong trường hợp nhiều hàm lồng nhau hoặc đệ quy nhiều lần dẫn đến không đủ vùng nhớ. | - Biến lưu trữ trong heap sẽ được gán địa chỉ ô nhớ. |

#### **Biến**

* Cách khai báo 1 biến: Datatype variableName
* Cách khởi tạo giá trị cho biến: variableName = value
* Cách khai báo và khởi tạo giá trị cho biến: Datatype variableName = value
* Cách khai báo và khởi tạo giá trị cho biến cùng kiểu dữ liệu: Datatype variableName, variableName1, variableName(n),....
* Cách khai báo biến hằng số: final DataType VARIABLE\_CONSTANT = value
* Lưu ý:
* Tên biến sẽ sử dụng cách khai báo camelCase.
* Khi khai báo tránh sử dụng các từ khóa của java “class”, “interface”, …
* Tên biến rõ ràng và có ý nghĩa.
* Tên biến bắt đầu bằng alphabet và không chứa các ký tự “\_” và “$”.
* Tên biến không chứa các ký tự đặc biệt.
* Không thể thay đổi giá trị của biến hằng số
* Khi khai báo hằng số phải đảm bảo quy tắc định danh viết in hoa chữ cái đầu của mỗi chữ.
* Khi khai báo một hằng số phải gán giá trị cho biến.
* VD:

#### **Type Casting: 2 kiểu**

* Implicit (ngầm định): khi gán giá trị của kiểu dữ liệu có miền giá trị bé hơn cho kiểu dữ liệu có miền giá trị lớn hơn.
* VD: int -> float
* Explicit (Sáng tỏ): khi gán giá trị của kiểu dữ liệu có miền giá trị lớn hơn cho kiểu dữ liệu có miền giá trị bé hơn
* VD: double -> float
* VD:



#### **Toán tử (operator):**

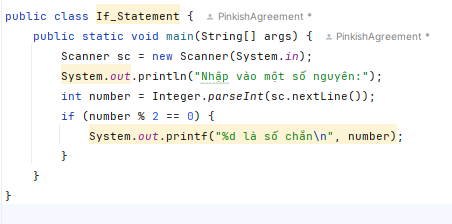
* Toán tử gán: =, +=, -=, %=, /=.
* Toán tử toán học: +, -, \*, /, %.
* Toán tử quan hệ: >, <, <=, >=, ==, !=.
* Toán tử logic: &&, ||.
* Toán tử 3 ngôi: ? :
* Toán tử tăng tiến: ++, –.

### **Câu lệnh điều khiển (Control Statement)**

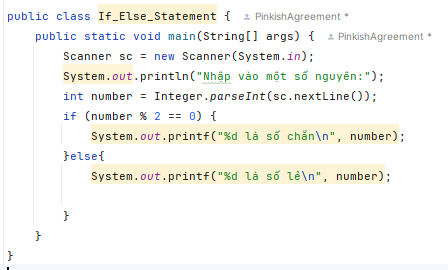
* Cho phép thay đổi luồng thực thi của chương trình.
* Dựa trên điều kiện mà một câu lệnh hoặc một nhóm các câu lệnh được thực hiện hoặc không.

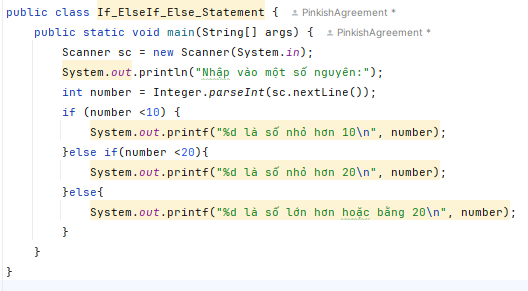
#### **Câu lệnh điều kiện (Condition Statement)**

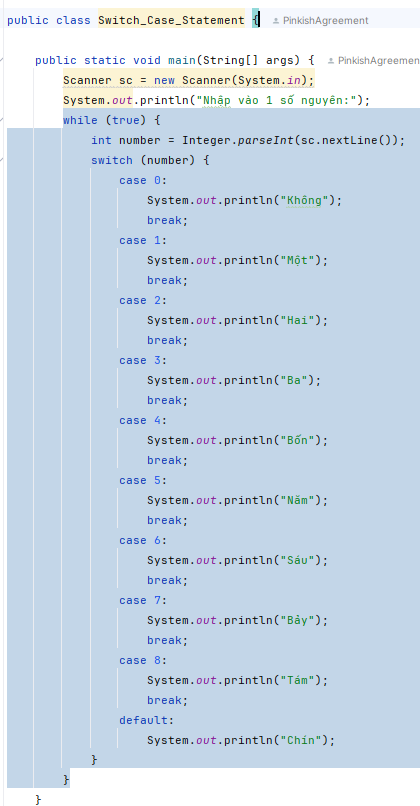
* Đánh giá một điều kiện và sẽ lựa chọn thực thi một câu lệnh nếu điều kiện đó đúng.
* Luồng thực thi sẽ được chạy ngay sau điều kiện if.
* Câu lệnh điều kiện if bao gồm các mệnh đề:
* if
* else if
* else
* Câu lệnh điều kiện switch sẽ so sánh điều kiện theo các case.
* Câu lệnh điều kiện if:
* Nếu điều kiện trong if là true, chương trình sẽ thực thi khối mã trong điều kiện if.
* VD:



* Câu lệnh điều kiện if, else:
* Nếu điều kiện trong if là true, chương trình sẽ thực thi khối mã trong điều kiện if. Nếu là false, chương trình sẽ thực thi khối mã trong else.
* VD:



* Câu lệnh điều kiện if, else if, else:
* Nếu điều kiện trong if là true, chương trình sẽ thực thi khối mã trong điều kiện if. Nếu là false, chương trình sẽ kiểm tra điều kiện mệnh đề else if.
* Nếu điều kiện trong else if là true, chương trình sẽ thực thi khối mã trong điều kiện else if. Nếu là false, chương trình sẽ thực thi khối mã trong mệnh đề else.
* VD:
* Câu lệnh điều kiện switch-case:
* Nếu biểu thức trong switch có giá trị khớp với case thì chương trình sẽ thực thi khối mã trong case đó.
* Từ khóa break để thoát ra khỏi câu lệnh điều kiện switch case.
* Nếu như một case có giá trị khớp với biểu thức trong switch và các case sau không có từ khóa break thì chương trình sẽ thực thi toàn bộ các case phía sau case đó.
* Nếu như không có case nào khớp với giá trị trong switch thì chương trình sẽ chạy default của câu lệnh switch case.
* Lưu ý:
* Câu lệnh if tương đương với câu lệnh switch case
* Câu lệnh if được sử dụng khi điều kiện xác định trên từng miền giá trị.
* Câu lệnh switch case được sử dụng khi điều kiện xác định trên từng giá trị.
* VD:



#### **Câu lệnh vòng lặp (Iterative Statement)**

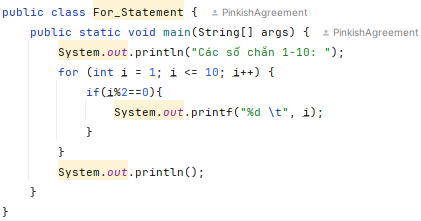
* là các câu lệnh được thực thi lặp đi lặp lại cho đến khi điều kiện xác định thỏa mãn.
* Có 4 câu lệnh vòng lặp: for, forEach, while, do while.
* Câu lệnh for:
* Gồm 3 thành phần được phân tách nhau bởi dấu “;”.
* Cú pháp:

for (initialize variableControl; condition; increment or decrement variableControl){

// statement

}

* Phần thân có thể chứa không, một, hoặc nhiều câu lệnh.
* Sử dụng khi biết trước số lần lặp.
* Thực hiện khi điều kiện xác định là true, kết thúc khi điều kiện xác định là false.
* VD:



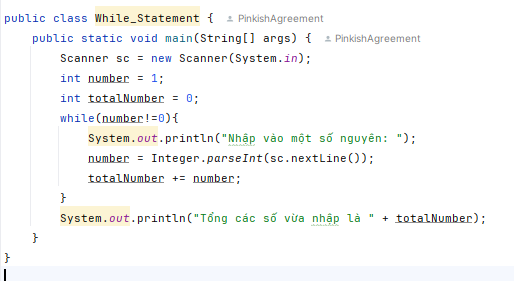
* Câu lệnh while:
* Cú pháp:

while (condition){

//statement

}

* Sử dụng khi không biết trước số lần lặp.
* Thực hiện khi điều kiện xác định là true, kết thúc khi điều kiện xác định là false.
* Lưu ý:
* Điều kiện luôn đúng sẽ khiến cho vòng lặp while vô tận, khi đó phải kiểm tra trong phần mã thân của vòng lặp while xem biến điều kiện trong while có được thay đổi trong mã thân hay không.
* VD:



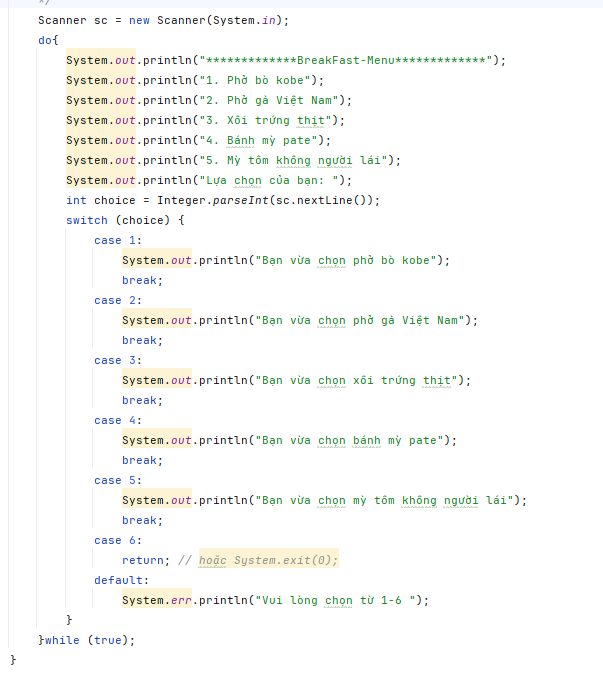
* Câu lệnh do while:
* Cú pháp:

do{

//statement

}

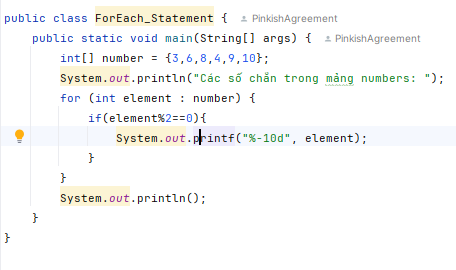
while (condition);

* Vòng lặp sẽ chạy do trước khi kiểm tra điều kiện trong while.
* Được sử dụng khi muốn thực thi vòng lặp ít nhất 1 lần.
* Thực hiện tiếp khi điều kiện xác định là true, kết thúc khi điều kiện xác định là false.
* VD: 
* Câu lệnh forEach:
* Cú pháp:

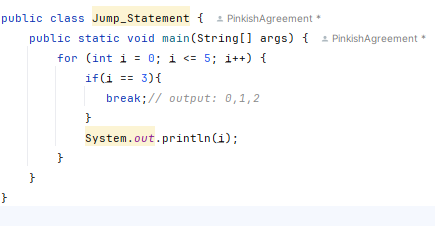
for(DataType element : array){

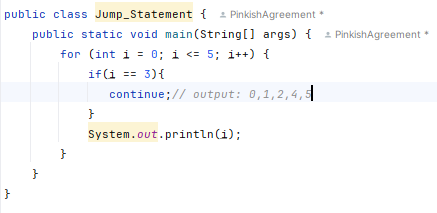
//statement

}

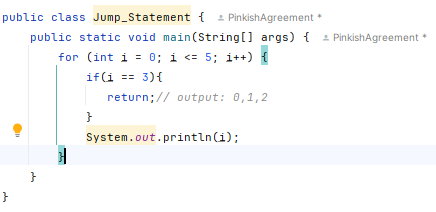
* Phần tử đầu tiên của mảng sẽ được gán cho element.
* Kiểu dữ liệu của element phải trùng với kiểu dữ liệu của các phần tử trong mảng.
* Vòng lặp sẽ kết thúc khi chạy đến phần tử cuối cùng.
  + VD:

#### **Câu lệnh ngắt (Jump Statement)**

* Bao gồm có break, continue, return.
* break:
* Được sử dụng để kết thúc vòng lặp, switch case.
* VD:
* continue:
* Được sử dụng để bỏ qua lần lặp hiện tại của vòng lặp.
* VD:



* return:
* Kết thúc phương thức và trả về giá trị cho nơi gọi hàm.
* VD:



### **VII. Lớp và đối tượng**

#### **1. Lớp**

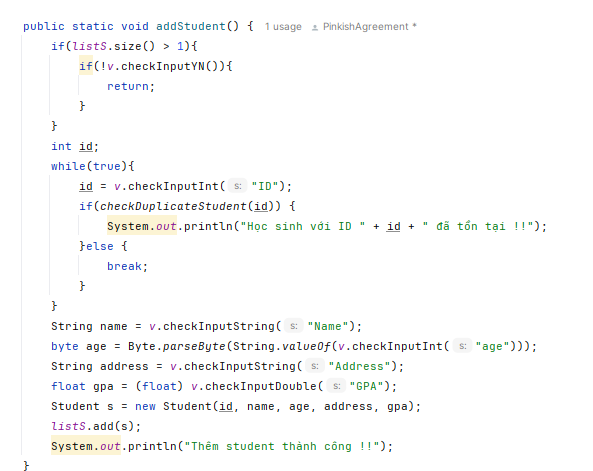
* Là một bản thiết kế hoặc mẫu để tạo ra đối tượng.
* Không cần phải phân bổ bộ nhớ khi tạo một lớp.
* Là một thực thể logic.
* Khai báo thông qua từ khóa class.
* Lớp chỉ có thể khai báo một lần.
* Một trường thành viên của lớp không có giá trị nào.
* Cú pháp khai báo 1 lớp:

AccessModifier class ClassName(){  
}

* Một lớp có thể chứa:
* Các thuộc tính.
* Constructor.
* Các phương thức.
* Lớp và interface.
* Khối block.
* VD:

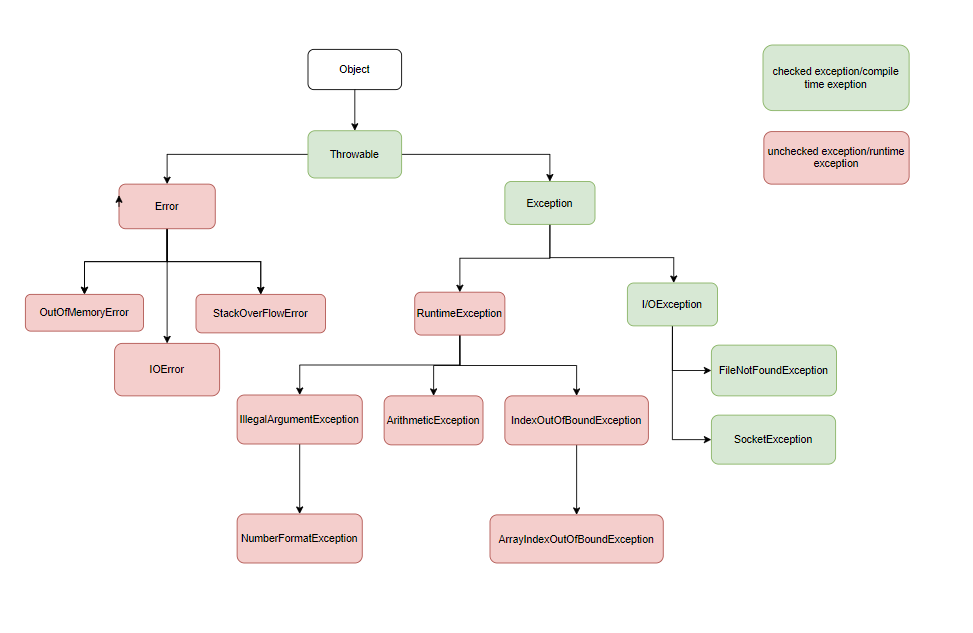


#### **2. Đối tượng**

* Là một trường hợp cụ thể của một lớp.
* Cần phải phân bổ bộ nhớ khi tạo một đối tượng.
* Là một thực thể vật lý.
* Khởi tạo thông qua từ khóa new, clone(), newInstance(), factory.
* Đối tượng có thể khai báo nhiều lần.
* Đối tượng phải bao gồm các trường thành viên và các giá trị tương ứng mỗi trường.
* VD:

## **Ngày 2: Exception**

### **Tổng quan**

* Là một tình trạng xảy ra một cách bất thường.
* Trong java, exception là một sự kiện làm gián đoạn luồng thực thi của chương trình, là một đối tượng được ném ra tại runtime.
* Exception Handling (xử lý ngoại lệ) là một cơ chế để xử lý các lỗi tại runtime như ClassNotFound, IO, SQL, remote, … 

### **Phân loại: 2 loại ngoại lệ chính**

#### **Checked**

* Đây là ngoại lệ xảy ra trong quá trình compile time (SQLException, IO Exception).

#### **Unchecked**

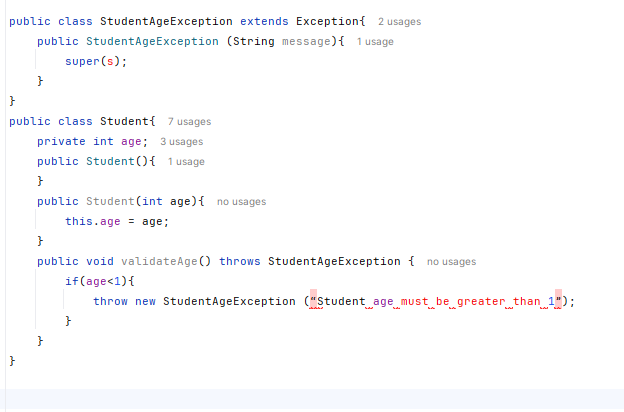
* Đây là ngoại lệ xảy ra trong quá trình runtime (ArithmeticException, NullPointerException).

#### **Error**

* Đây là những lỗi không thể khắc phục (OutOfMemoryError, VirtualMachineError).

### **Cách để custom một exception**

* Có thể hoàn toàn tự định nghĩa một exception để phù hợp bằng cách kế thừa lớp Exception hoặc các lớp con của nó.
* VD:



### **Lớp java.lang.Exception**

* Tất cả lớp của exception đều là lớp con của java.lang.Exception, lớp con của Throwable.
* Các phương thức cơ bản của lớp Exception:

| String getMessage() | Trả về một message cụ thể của một Exception đã xảy ra. Message này được khởi tạo bởi constructor của Throwable |
| --- | --- |
| Throwable getCause() | Trả về nguyên nhân xảy ra Exception biểu diễn bởi đối tượng Throwable |
| public String toString() | Trả về tên của lớp và kết quả của phương thức getMessage() |
| public void printStackTrace() | In ra kết quả của phương thức toString() cùng với stack trace của System.err |
| StackTraceElement[] getStackTrace() | Trả về mảng chứa phần tử trên stack trace. Phần tử tại index 0 biểu diễn phần trên cùng của stack trace. Phần tử tại chỉ mục cuối cùng biểu diễn phần dưới cùng của stack trace. |

### **Cách thức xử lý ngoại lệ (Exception Handling)**

* Xử lý ngoại lệ (Exception Handling) trong java là cơ chế xử lý các lỗi runtime để có thể duy trì luồng bình thường của chương trình.
* Quá trình xử lý Exception được gọi là catch Exception, nếu Runtime System không xử lý được ngoại lệ thì chương trình sẽ kết thúc.
* Java cung cấp 5 từ khóa để xử lý ngoại lệ trong java:
* try: dùng để viết đoạn mã có thể xảy ra ngoại lệ, theo sau phải là khối catch hoặc finally.
* catch: dùng để viết đoạn mã thực hiện việc xử lý ngoại lệ.
* finally: dùng để viết khối mã cần thực thi của chương trình. Khối mã này sẽ luôn được thực thi.
* throws: dùng để khai báo một ngoại lệ.
* throw: dùng để ném ra một ngoại lệ.

#### **Khối try - catch**

* Dùng để tách biệt phần mã xử lý ngoại lệ và phần mã có thể sinh ra lỗi.
* Cú pháp:

try{

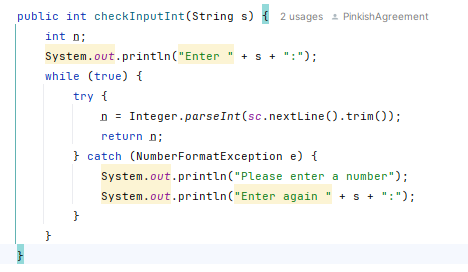
// statement // khối lệnh có thể sinh ra lỗi

}catch (ExceptionType name){ // ngoại lệ muốn bắt và xử lý

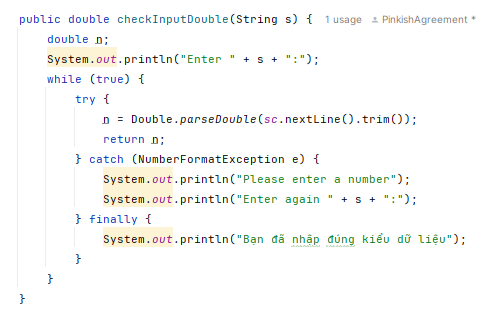
// statement // khối lệnh để xử lý lỗi

}

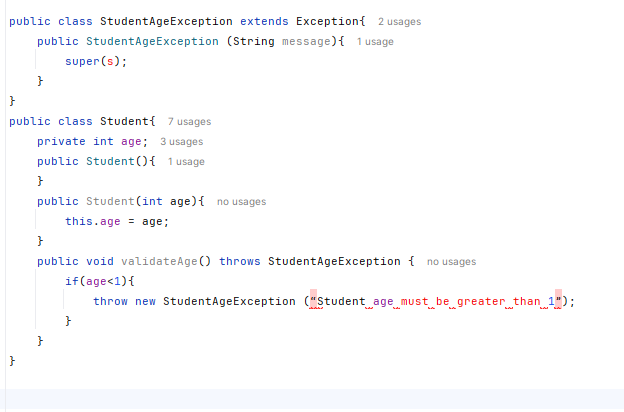
* Lưu ý:
* Khi một câu lệnh trong khối try bị lỗi thì các câu lệnh bên dưới hoàn toàn bị bỏ qua.
* Tham số của catch là ngoại lệ muốn bắt, có thể sử dụng nhiều khối catch nối tiếp nhau để có thể bắt được các ngoại lệ khác nhau.
* VD:



#### **Khối finally**

* Dùng để viết đoạn mã cần thực thi của chương trình sau khối try - catch bất kể kết quả.
* VD:

#### **Từ khóa throws, throw:**

* Throws: dùng để khai báo ngoại lệ có thể sinh ra trên 1 phương thức (Chủ yếu là checked exception).
* Throw: dùng để ném ra ngoại lệ cụ thể, thông thường sử dụng với exception người dùng tự định nghĩa.
* VD:

### **List, Set, Map**

* Đây đều là những interface extends từ interface collection.
* Collection là một interface dùng để biểu diễn một nhóm các đối tượng riêng lẻ thành một đơn vị duy nhất (single unit). Collection nằm trong gói java.util mà java cung cấp để thao tác và làm việc với dữ liệu dạng tập hợp.

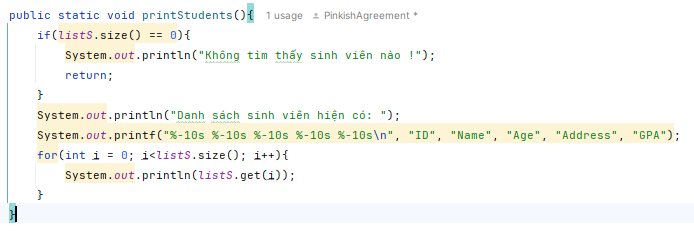
#### **List**

* Có thể lưu trữ các phần tử trùng nhau.
* Trong interface List gồm có các class triển khai (implements) bao gồm:
* ArrayList
* LinkedList
* Vector
* **ArrayList**
* Là một tập hợp các phần tử được lưu trữ và truy cập theo chỉ số index.
* Có khả năng tự thay đổi kích thước.
* Các phương thức của ArrayList:

| boolean add (Object o) | Dùng để chèn thêm phần tử chỉ định tại cuối danh sách |
| --- | --- |
| void add (int index, Object o) | Dùng để chèn thêm phần tử chỉ định theo vị trí |
| boolean addAll (Collection<? extends E> c) | Dùng để chèn tất cả các phần tử của một collection chỉ định vào cuối danh sách |
| boolean addAll (int index, Collection<? extends E> c) | Dùng để chèn tất cả các phần tử của một collection chỉ định theo chỉ số index chỉ định |
| void clear() | Dùng để xóa toàn bộ những phần tử có trong danh sách |
| Object clone() | Dùng để sao chép một danh sách |
| int size() | Dùng để trả về kích thước của danh sách |
| void sort(Comparator<? super E> c) | Dùng để sắp xếp danh sách dựa trên comparator chỉ định |
| boolean remove(Object o) | Dùng để loại bỏ một phần tử chỉ định có trong danh sách |
| boolean contains (Object o) | Dùng để kiểm tra xem phần tử chỉ định có tồn tại trong danh sách, nếu tồn tại trả về true, ngược lại là false |
| boolean isEmpty() | Dùng để kiểm tra xem danh sách có rỗng, nếu rỗng trả về true, ngược lại là false |

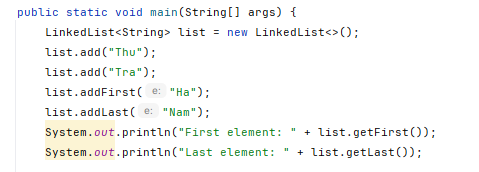
| Ưu điểm | Nhược điểm |
| --- | --- |
| Có thể dùng để chứa các đối tượng của các lớp khác nhau. | Khi kích thước của ArrayList vượt quá giới hạn có thể khiến cho chương trình bị chậm đi. |
| Dễ dàng thực hiện thao tác duyệt danh sách, tìm kiếm phần tử. | Thêm phần tử vào cuối danh sách nhanh hơn so với việc thêm vào giữa danh sách. |

* VD:



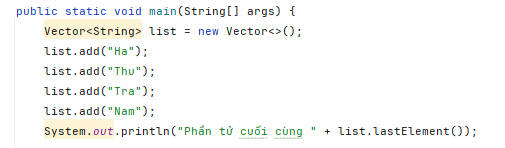
* **LinkedList**
* Là một danh sách chứa các Node liên tiếp nhau theo thứ tự tuyến tính.
* Phần tử đầu của LinkedList gọi là head của danh sách.
* Mỗi một Node là một đối tượng lưu trữ tham chiếu đến phần tử khác và một tham chiếu, gọi là next, đến một Node khác.
* Tham chiếu next bên trong một Node có thể xem như một liên kết hoặc một con trỏ đến Node khác.
* LinkedList truy cập đến các phần tử chậm hơn tương đối so với ArrayList do phải duyệt qua từng Node trong khi đó ArrayList truy cập thông qua chỉ số index của từng phần tử.
* Các phương thức của LinkedList:

| boolean add (Object o) | Dùng để nối thêm phần tử chỉ định |
| --- | --- |
| void add (int index, Object o) | Dùng để nối thêm phần tử chỉ định theo vị trí |
| void addFirst(Object o) | Dùng để chèn phần tử chỉ định vào đầu danh sách |
| void addLast(Object o) | Dùng để chèn phần tử chỉ định vào cuối danh sách |
| Object getFirst () | Dùng để trả về phần tử đầu danh sách |
| Object getLast () | Dùng để trả về phần tử cuối danh sách |
| boolean remove (Object o) | Dùng để xóa phần tử chỉ định |
| boolean contains (Object o) | Trả về true nếu tồn tại phần tử trong danh sách, ngược lại là false |
| int size () | Dùng để trả về kích thước của danh sách |
| int indexOf (Object o) | Dùng để trả về chỉ mục của phần tử chỉ định trong danh sách, nếu không tìm thấy trả về -1 |
| int lastIndexOf (Object o) | Dùng để trả về chỉ mục của phần tử chỉ định xuất hiện cuối cùng trong danh sách, nếu không tìm thấy trả về -1 |
| Object get (int index) | Trả về phần tử theo chỉ số chỉ định trong danh sách |

* VD:
* **Vector:**
* Hoạt động giống với ArrayList.
* Sử dụng trong môi trường đa luồng, tại 1 thời điểm chỉ có một luồng được thực thi, truy cập và thao tác đến danh sách.

| boolean add (Object o) | Dùng để nối thêm phần tử chỉ định |
| --- | --- |
| void add (int index, Object o) | Dùng để nối thêm phần tử chỉ định theo vị trí |
| boolean addAll (Collection c) | Thêm tất cả các phần tử của một collection chỉ định vào cuối vector |
| boolean addAll (int index, Collection c) | Thêm tất cả các phần tử của một collection chỉ định vào chỉ số index chỉ định |
| void copyInto(Object[] array) | Dùng để sao chép các phần tử vector này vào mảng chỉ định |
| Object clone() | Dùng để trả về một bản sao vector |
| int size() | Dùng để trả về kích thước của vector |
| String toString() | Dùng để trả về một chuỗi biểu diễn của từng phần tử trong vector |
| void clear() | Xóa tất cả phần tử của vector |
| boolean contains (Object o) | Trả về true nếu tồn tại phần tử trong danh sách, ngược lại là false |
| int size () | Dùng để trả về kích thước của danh sách |
| Object get (int index) | Trả về phần tử theo chỉ số chỉ định trong vector |
| Object getFirstElement | Dùng để trả về phần tử chỉ số 0 của vector |
| boolean remove(Object o) | Dùng để loại bỏ một phần tử chỉ định có trong vector |

* VD:

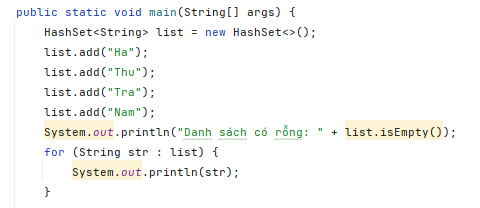


#### **Set**

* Dùng để lưu trữ các phần tử riêng biệt.
* Trong interface Set bao gồm có 3 class và 1 interface:
* HashSet
* LinkedHashSet
* SortedSet
* TreeSet
* HashSet:
* Chỉ chứa các phần tử duy nhất không được trùng lặp.
* Được phép chứa phần tử Null.
* Các phần tử đưa vào Set không đảm bảo sắp xếp theo thứ tự
* Các phương thức của HashSet:

| void add (Object o) | Dùng để nối thêm phần tử chỉ định |
| --- | --- |
| boolean contains (Object o) | Trả về true nếu tồn tại phần tử trong collection, ngược lại là false |
| boolean remove (Object o) | Dùng để loại bỏ một phần tử chỉ định |
| void clear() | Xóa tất cả phần tử |
| int size() | Dùng để trả về kích thước của collection |
| boolean isEmpty() | Dùng để kiểm tra xem danh sách có rỗng, nếu rỗng trả về true, ngược lại là false |

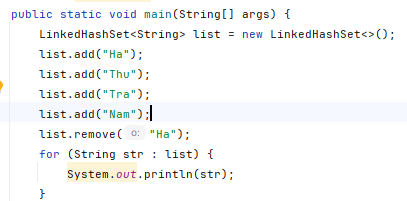
* VD:



* LinkedHashSet:
* Chỉ chứa các phần tử duy nhất không được trùng lặp.
* Được phép chứa phần tử Null.
* Các phần tử đưa vào HashSet đảm bảo thứ tự được thêm vào.
* Các phương thức của LinkedHashSet:

| void add (Object o) | Dùng để nối thêm phần tử chỉ định |
| --- | --- |
| boolean contains (Object o) | Trả về true nếu tồn tại phần tử trong collection, ngược lại là false |
| boolean remove (Object o) | Dùng để loại bỏ một phần tử chỉ định |
| void clear() | Xóa tất cả phần tử |
| int size() | Dùng để trả về kích thước của collection |
| boolean isEmpty() | Dùng để kiểm tra xem danh sách có rỗng, nếu rỗng trả về true, ngược lại là false |

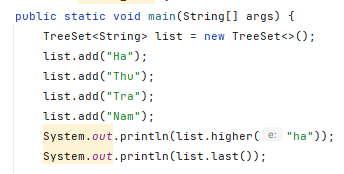
* VD:



* TreeSet:
* Chỉ chứa các phần tử duy nhất không được trùng lặp.
* Không được phép chứa phần tử Null.
* Các phần tử đưa vào TreeSet đảm bảo thứ tự được thêm vào.
* Các phương thức của TreeSet:

| void add(Object o) | Chèn các phần tử được chỉ định vào tập hợp |
| --- | --- |
| boolean addAll(Collection c) | Chèn các phần tử của tập hợp được chỉ định vào tập hợp |
| remove(Object o) | Xóa các phần tử được chỉ định vào tập hợp |
| boolean removeAll() | Xóa tất cả các phần tử khỏi tập hợp |
| Object first() | Trả về phần tử đầu tiên của tập hợp |
| Object last() | Trả về phần tử cuối cùng của tập hợp |
| Object higher(Object o) | Trả về phần tử có giá trị lớn hơn phần tử chỉ định trong tập hợp |
| Object lower(Object o) | Trả về phần tử có giá trị bé hơn phần tử chỉ định trong tập hợp |
| Object ceiling (Object o) | Trả về phần tử có giá trị bé nhất trong số các phần tử lớn hơn phần tử chỉ định |
| Object floor (Object o) | Trả về phần tử có giá trị lớn nhất trong số các phần tử bé hơn phần tử chỉ định |

* VD:

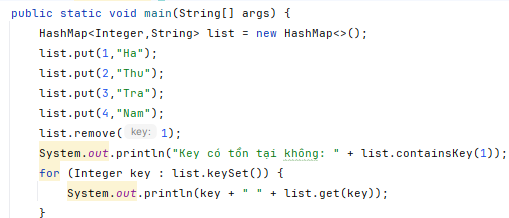


#### Map

* Lưu trữ dữ liệu dưới dạng key, value
* Không thể chứa các key bị trùng lặp.
* Mỗi cặp key và value gọi là entry.
* Có thể chứa các giá trị trùng lặp.
* Trong Map chứa 3 class và 1 interface:
* HashMap
* LinkedHashMap
* SortedMap
* TreeMap
* HashMap:
* Không đảm bảo thứ tự cho các phần tử.
* Cho phép 1 key Null và nhiều giá trị Null.
* Các phương thức của HashMap:

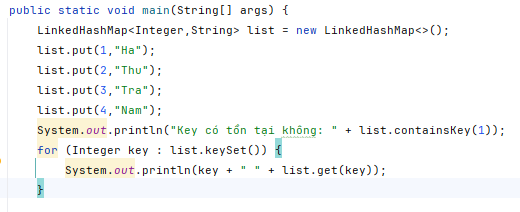
| Object put(Object K, Object V) | thêm 1 cặp khóa và giá trị vào trong map |
| --- | --- |
| Object get (Object K) | trả về giá trị của key chỉ định |
| boolean isEmpty() | kiểm tra map có rỗng |
| Object remove (Object K) | dùng để xóa 1 mục đại diện là key ra khỏi map |
| Object remove (Object K, Object V) | dùng để xóa 1 cặp key - value |
| Set entrySet() | trả về tập hợp các key-value |
| boolean containsKey (Object key) | kiểm tra xem có tồn tại key chỉ định |
| boolean containsValue (Object value) | kiểm tra xem có tồn tại value chỉ định |
| int size() | trả về kích thước của map |
| Set keySet() | trả về tập hợp các key |
| void clear() | xóa hết tất cả phần tử trong map |
| putIfAbsent(Object key, Object value) | chèn cặp key-value vào nếu key không tồn tại |
| Set values() | trả về tập hợp tất cả các value của map |

* VD:

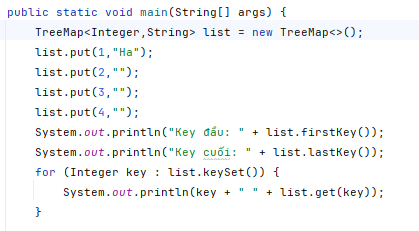


* LinkedHashMap:
* Duy trì thứ tự cho các phần tử.
* Cho phép 1 key Null và nhiều giá trị Null.
* Các phương thức của LinkedHashMap:

| Object put(Object K, Object V) | thêm 1 cặp khóa và giá trị vào trong map |
| --- | --- |
| Object get (Object K) | trả về giá trị của key chỉ định |
| boolean isEmpty() | kiểm tra map có rỗng |
| Object remove (Object K) | dùng để xóa 1 mục đại diện là key ra khỏi map |
| Object remove (Object K, Object V) | dùng để xóa 1 cặp key - value |
| Set entrySet() | trả về tập hợp các key-value |
| boolean containsKey (Object key) | kiểm tra xem có tồn tại key chỉ định |
| boolean containsValue (Object value) | kiểm tra xem có tồn tại value chỉ định |
| int size() | trả về kích thước của map |
| Set keySet() | trả về tập hợp các key |
| void clear() | xóa hết tất cả phần tử trong map |
| putIfAbsent(Object key, Object value) | chèn cặp key-value vào nếu key không tồn tại |
| Set values() | trả về tập hợp tất cả các value của map |

* VD:
* TreeMap:
* Duy trì thứ tự các phần tử.
* Không cho phép bất kỳ key nào null và nhiều giá trị được phép null.

| Object put(Object K, Object V) | thêm 1 cặp khóa và giá trị vào trong map |
| --- | --- |
| Object get (Object K) | trả về giá trị của key chỉ định |
| boolean isEmpty() | kiểm tra map có rỗng |
| Object remove (Object K) | dùng để xóa 1 mục đại diện là key ra khỏi map |
| Object remove (Object K, Object V) | dùng để xóa 1 cặp key - value |
| Set entrySet() | trả về tập hợp các key-value |
| boolean containsKey (Object key) | kiểm tra xem có tồn tại key chỉ định |
| boolean containsValue (Object value) | kiểm tra xem có tồn tại value chỉ định |
| int size() | trả về kích thước của map |
| Set keySet() | trả về tập hợp các key |
| void clear() | xóa hết tất cả phần tử trong map |
| putIfAbsent(Object key, Object value) | chèn cặp key-value vào nếu key không tồn tại |
| Collection values() | trả về tập hợp tất cả các value của map |
| Object firstKey() | trả về key đầu tiên của mảng đã được sắp xếp |
| Object lastKey() | trả về key cuối cùng của mảng đã được sắp xếp |
| SortedMap subMap((K startKey, K endKey) | Trả về một phần của Tree Map từ start key đến end key |

* VD:

### **Iterators và các thao tác duyệt Collection**

#### **Iterator:**

* Iterator là một interface dùng để truy cập và thao tác đến collection:
* Duyệt từ đầu đến cuối các phần tử của một collection.
* Cho phép xóa các phần tử khi lặp một collection.
* Các phương thức của Iterator:

| boolean hasNext() | Trả về true nếu Iterator còn phần tử kế tiếp phần tử đang duyệt |
| --- | --- |
| object next() | Trả về phần tử hiện tại và di chuyển đến phần tử tiếp theo |
| void remove() | Loại bỏ phần tử cuối được trả về bởi Iterator |

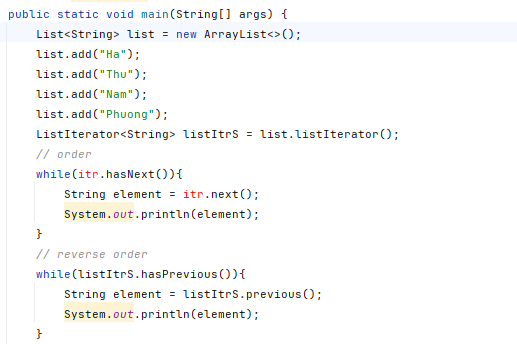
#### **ListIterator Interface:**

* Với các collection implement List, thì có thể sử dụng ListIterator.

| void add(Object obj) | Chèn obj vào phía trước phần tử sẽ được trả về bởi lệnh gọi tiếp theo tới next(). |
| --- | --- |
| boolean hasNext() | Trả về true nếu có phần tử tiếp theo. Nếu không, trả về false. |
| boolean hasPrevious() | Trả về true nếu có phần tử trước. Nếu không, trả về false. |
| Object next() | Trả về phần tử tiếp theo. Một NoSuchEuityException được ném nếu không có phần tử tiếp theo. |
| int nextIndex() | Trả về chỉ mục của phần tử tiếp theo. Nếu không có phần tử tiếp theo, trả về kích thước của danh sách. |
| Object previous() | Trả về phần tử trước. Một NoSuchEuityException được ném nếu không có phần tử trước đó. |
| int previousIndex() | Trả về chỉ mục của phần tử trước. Nếu không có phần tử trước, trả về -1. |
| void remove() | Loại bỏ các phần tử hiện tại khỏi danh sách. Một IllegalStateException được ném nếu remove() được gọi trước next() hoặc trước previous(). |
| void set(Object obj) | Chỉ định obj cho phần tử hiện tại. Đây là phần tử được trả về lần cuối bởi một lệnh gọi tới next() hoặc previous(). |
| void add(Object obj) | Chèn obj vào phía trước phần tử sẽ được trả về bởi lệnh gọi tiếp theo tới next(). |
| boolean hasNext() | Trả về true nếu có phần tử tiếp theo. Nếu không, trả về false. |

* VD:



* Có thể duyệt collection thông qua cách sau:
* Sử dụng iterator:
  + VD:

## **Ngày 3: Collection Framework**

### **Generics**

* Dùng để tham số hóa kiểu dữ liệu.
* Giúp dễ dàng bắt lỗi các kiểu dữ liệu không hợp lệ, và tăng tính tái sử dụng khi

có thể tận dụng các interface, class, method với nhiều kiểu dữ liệu khác nhau.

* Là một tính năng tại thời điểm biên dịch (compile time) và các thông tin về kiểu sẽ được xóa bỏ (type erasure), tất cả các kiểu generic được thực hiện dưới dạng kiểu đối tượng.
* Không thể sử dụng kiểu dữ liệu nguyên thủy (primitive) dùng làm tham số kiểu trong generic. Để sử dụng kiểu dữ liệu nguyên thủy, cần phải sử dụng các wrapper class tương ứng với các kiểu dữ liệu, wrapper class: cho phép sử dụng kiểu dữ liệu nguyên thủy dưới dạng các đối tượng.

#### **Quy ước trong generics**

* Cần phải đặt tên theo tiêu chuẩn để giúp mã dễ đọc hơn.
* Các kiểu tham số:
* T (Type): kiểu dữ liệu bất kỳ (String, int, long, float, …).
* E (Element): phần tử (Collection Framework).
* K (Key): khóa.
* V (Value): giá trị.
* N (Number): kiểu số (int, float, double ,....).

#### **Các kiểu generic**

* Generic trong class:
* Là một lớp tổng quát, định nghĩa với một hoặc nhiều tham số kiểu dữ liệu (type parameter). Tham số kiểu dữ liệu này sử dụng để chỉ định kiểu dữ liệu cho các thuộc tính và phương thức của lớp
* Generic trong interface:
* Là một interface chứa một hoặc nhiều tham số kiểu dữ liệu (type parameter), đại diện cho kiểu dữ liệu khác nhau sử dụng trong các phương thức hoặc lớp generic.
* Generic trong method:
* Là một phương thức có khả năng làm việc với nhiều kiểu dữ liệu khác nhau mà không cần định nghĩa lại phương thức cho từng kiểu dữ liệu cụ thể.

#### **Giới hạn các generic**

* Cho phép đưa ra giới hạn cho loại tham số trong class hoặc interface.
* Giới hạn kiểu tham số có giới hạn trên:
  + Sử dụng từ khóa extends để có thể giới hạn các generic. Khi sử dụng từ khóa extends thì các tham số chỉ có thể là chấp nhận các đối tượng thuộc kiểu dữ liệu đó hoặc các lớp con của kiểu dữ liệu đó, hoặc các lớp implements từ interface.
  + Cú pháp: T extends ?
* Giới hạn kiểu tham số có giới hạn dưới:
* Sử dụng từ khóa super. Khi sử dụng từ khóa super, chỉ chấp nhận kiểu tham số là lớp cha của lớp chỉ định.
* Multiple bounds:
* Dùng để giới hạn kiểu dữ liệu của tham số generic và phải thỏa mãn nhiều yêu cầu.
* Cú pháp: T extends ? & ? & ?
* VD: T extends Number & Comparable<T>
* Lưu ý: Các lớp phải được đặt trước các interface, và không thể có nhiều hơn một lớp trong danh sách giới hạn.

#### **Type Erasure trong generic**

* Gọi là xóa kiểu trong quá trình biên dịch
* Để đảm bảo an toàn kiểu (safety type).
* Không ảnh hưởng đến hiệu năng của chương trình.

#### **Các ưu điểm**

* An toàn kiểu: Tránh việc ép kiểu (casting) không an toàn khi làm việc với đối tượng và cấu trúc dữ liệu, chỉ cần thêm các đối tượng thuộc kiểu đã chỉ định vào cấu trúc dữ liệu đó.
* Tái sử dụng mã: Giúp tạo ra các lớp và phương thức có thể hoạt động với nhiều kiểu dữ liệu khác nhau, tránh việc phải viết mã riêng biệt cho từng kiểu dữ liệu trong chương trình.
* Loại bỏ việc ép kiểu: Trình biên dịch sẽ tự động chèn ép các kiểu cần thiết vào mã biên dịch. Tránh việc ép kiểu thủ công và giảm thiểu lỗi do việc ép kiểu sai.
* Kiểm tra lỗi tại compile time: Giúp phát hiện và sửa lỗi sớm hơn tại thời điểm biên dịch, không cần phải đợi lúc chương trình chạy (runtime) mới phát hiện lỗi liên quan đến kiểu dữ liệu.
* VD:

### **Lambda Expressions**

* Được giới thiệu từ java 8.
* Là một hàm ẩn danh, có đặc điểm của một function (tham số và body: nội dung thực thi).
* Dùng để tạo ra thể hiện (instance) cho functional interface (là interface chỉ có duy nhất một phương thức trừu tượng). Cung cấp implements cho method định nghĩa ở functional interface.
* Sử dụng toán tử mũi tên -> để chia biểu thức thành 2 phần: tham số và nội dung thực thi.
* Cú pháp (Các argument) -> {body}
* Các argument: có thể không, một hoặc nhiều argument.
* arrow operator: dùng để liên kết các argument và phần thực thi của biểu thức.
* body: chứa các câu lệnh của biểu thức lambda.
* Chú ý:
* Không cần phải khai báo kiểu dữ liệu cho tham số vì trình biên dịch sẽ suy luận ra kiểu dữ liệu dựa vào mã thực thi
* Khi phần thực thi của biểu thức chỉ có 1 câu lệnh thì không cần phải sử dụng từ khóa return và loại bỏ dấu {} và ngược lại.
* Khi chỉ truyền vào một tham số duy nhất thì có thể bỏ qua dấu () và ngược lại.
* VD:



### **Streams API**

* Được giới thiệu từ java 8.
* Hỗ trợ thao tác với collection và mảng dễ dàng và tối ưu hơn.
* Cú pháp của stream ngắn gọn và hiệu suất tốt hơn.
* Với sự trợ giúp của stream, chương trình có thể hoạt động song song.
* Hạn chế: stream không thể tái sử dụng khi đã gọi terminal operation.

#### **Các phương thức tạo ra stream**

* stream(): trả về một stream sẽ được xử lý theo tuần tự.
* parallelStream(): trả về một stream song song, các xử lý sau đó sẽ thực hiện song song.

#### **Đặc điểm**

* Không lưu trữ các phần tử của collection hay array.
* Không phải là cấu trúc dữ liệu.
* Là immutable object, các hoạt động tổng hợp trên collection, array mà không làm ảnh hưởng đến dữ liệu của chúng.
* Không dùng index để truy xuất các phần tử trong stream.
* stream chỉ có thể dùng một lần không thể gọi lại stream.

#### **Các bước để làm việc với stream**

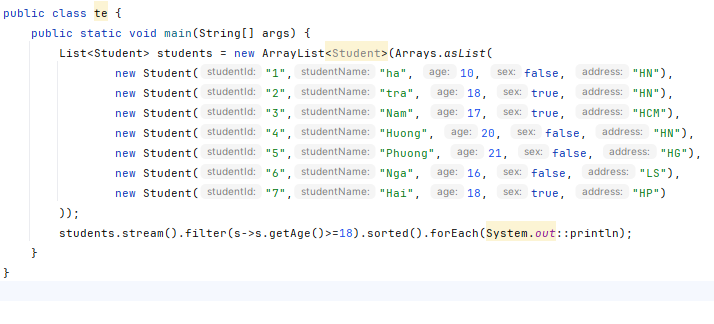
* Tạo stream (stream resource).
* Thực hiện các thao tác trung gian (intermediate operation) trên stream ban đầu để chuyển thành một stream khác và tiếp tục thực hiện các hoạt động trung gian khác.
* Thực hiện thao tác đầu cuối (terminal operation) trên stream cuối cùng để nhận kết quả và sau đó không thể tái sử dụng.
* 1 stream pipeline bao gồm việc tạo stream resource, sau đó 0, 1 hoặc nhiều các intermediate operation, và 1 terminal operation.

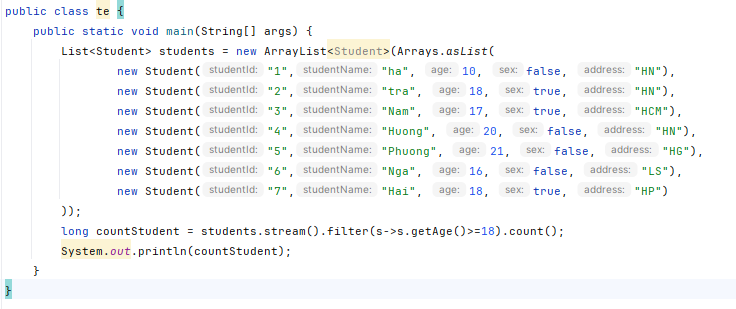
#### **Các thao tác đầu cuối (terminal operation) của stream**

| forEach() | duyệt qua các phần tử của stream() |
| --- | --- |
| collect(Collector.toList()) | giúp thu nhập kết quả Stream sang một Collection |
| anyMatch() | trả về boolean tùy thuộc vào điều kiện |
| allMatch() | trả về true nếu tất cả thỏa mãn điều kiện |
| noneMatch() | Ngược lại với allMatch |
| count() | trả về tổng số phần tử cho stream |
| min() | đối số là một Comparator để so sánh với nhau tìm giá trị tối thiểu hoặc tối đa |
| max() |
| summaryStatistic() | dùng để lấy giá trị count, min, max, sum, average với tập dữ liệu số |
| reduce() | dùng để kết hợp các phần tử stream thành một bằng cách sử dụng BinaryOperator |

#### **Parallel streams**

* Các thao tác stream tuần tự được thực hiện trên một luồng đơn (single thread).
* Các thao tác stream song song được thực hiện đồng thời trên nhiều luồng (multi-thread).
* Sử dụng parallel stream khi muốn thao tác với tập dữ liệu lớn cần hiệu suất nhanh.
* VD:





## **Ngày 4: Làm việc với File**

### **Thread**

* Luồng trong java là một đơn vị xử lý độc lập trong chương trình, mỗi luồng là một dòng thực thi độc lập trong chương trình.
* Luồng cho phép thực hiện đa luồng (multithread) giúp tối ưu hiệu suất và tận dụng tối đa tài nguyên, chạy song song với các luồng khác.

#### **Multithreading**

* Cho phép thực hiện đồng thời nhiều tác vụ trong cùng một chương trình. Một chương trình java gồm hai hay nhiều luồng chạy đồng thời, chúng chia sẻ tài nguyên hệ thống như bộ nhớ và CPU. Các thread làm việc độc lập hoặc phối hợp với nhau hoàn thành một công việc lớn hơn.
* Các luồng chia sẻ không gian địa chỉ ô nhớ giống nhau.

#### **Vòng đời**

* New: Đây là phương thức khởi tạo luồng của lớp Thread nhưng chưa được start().
* Runnable: Sau khi gọi phương thức start(), các luồng sẽ được cấp phát tài nguyên và lịch điều phối CPU cho luồng.
* Waiting: Dùng để làm gián đoạn luồng không giới hạn cho đến khi một luồng khác làm cho nó chạy tiếp.
* Timed\_Waiting: Dùng để làm gián đoạn luồng trong thời gian giới hạn.
* Blocked: Đây là trạng thái not runnable, luồng vẫn còn sống nhưng hiện tại không được chọn để chạy.
* Terminated: Một luồng ở trong trạng thái terminated hoặc dead khi phương thức run() kết thúc.

| Ưu điểm | Nhược điểm |
| --- | --- |
| Luồng độc lập, khi có ngoại lệ xảy ra tại một luồng không ảnh hưởng đến luồng khác | Do nhiều luồng hoạt động đồng thời nên có thể xảy ra việc tranh chấp bộ nhớ |
| Có thể có nhiều hoạt động với nhau giúp tiết kiệm thời gian | Càng nhiều luồng xử lý càng phức tạp |
|  | Có thể xảy ra hiện tượng deadlock, khi có luồng không được sử dụng trong chương trình |

#### **Tạo luồng**

* Extends từ lớp Thread:
* Khai báo 1 lớp mới kế thừa từ class Thread.
* Override lại phương thức run ở lớp này, sau khi luồng chạy xong tất cả các câu lệnh trong phương thức run thì luồng cũng tự hủy.
* Tạo 1 thể hiện hoặc 1 đối tượng lớp vừa khai báo.
* Sau đó gọi phương thức start() để bắt đầu thực thi luồng.
* Chú ý:
* Sau khi đã start và luồng đã kết thúc thì không thể sử dụng luồng lại được nữa, nếu sử dụng tiếp sẽ xảy ra ngoại lệ IllegalThreadStateException.
* Implement từ interface Runnable:
* Khai báo 1 lớp mới kế thừa từ interface Runnable.
* Override lại phương thức run ở lớp này.
* Tạo một thể hiện hoặc đối tượng của lớp vừa khai báo.
* Tạo một thể hiện của lớp Thread.
* Gọi phương thức start().

#### **Phương thức Thread**

| suspend() | tạm ngưng hoạt động của một luồng bằng cách ngừng cung cấp CPU cho luồng này |
| --- | --- |
| start() | Khởi tạo một thread |
| run() | Chạy phương thức run trong thread |
| resume() | làm cho phương thức chạy lại khi luồng bị dừng do phương thức suspend() bên trên, đưa luồng vào lịch điều phối CPU để luồng được cấp CPU chạy bình thường |
| destroy() | dừng hẳn luồng |
| isAlive() | kiểm tra xem luồng còn hoạt động, trả về true nếu luồng chưa được start hoặc chưa dead, trả về false nếu luồng mới được khởi tạo hoặc đang ở trạng thái dead |
| yield() | khi gọi phương thức này luồng sẽ bị ngừng cấp CPU và nhường cho các luồng khác trong hàng chờ, khác với suspend() là tạm dừng luồng mà chỉ ngừng cấp trong lần nhận CPU đó |
| sleep() | tạm dừng luồng trong khoảng thời gian (millisecond) |
| join() | thông báo chờ thread hoàn thành, sau đó thread cha sẽ được chạy |
| getName() | trả về tên thread |
| setName() | thay đổi tên thread |
| getState() | trả về trạng thái của thread |
| getId() | trả về Id của thread |
| getPriority() | trả về mức độ ưu tiên của thread |
| setPriority() | thay đổi mức độ ưu tiên của thread |
| interrupt() | làm gián đoạn một luồng |
| isInterrupted() | kiểm tra nếu thread đã bị ngắt |

* VD:



### **IO File**

* Chú ý:
* Byte stream và character stream việc đọc và xuất file được thực hiện trực tiếp dưới quyền của hệ điều hành, gây lãng phí thời gian và tài nguyên.

#### **Byte stream**

* Các chương trình sử dụng byte stream để đọc ghi file với từng byte (8 bit). Tất cả các class byte stream có nguồn gốc từ InputStream và OutputStream. Có nhiều lớp liên quan đến byte stream, nhưng các lớp thường dùng là FileInputStream và FileOutputStream
* Việc đóng một Stream là một việc rất quan trọng để tránh bị rò rỉ tài nguyên. Khi thực hiện xong việc đọc ghi file có thể sử dụng khối finally để đảm bảo đóng các Stream ngay cả khi có lỗi xảy ra.

#### **Character stream**

* Dùng để thực hiện input và output cho Unicode 16 bit. Tất cả các class của character stream có nguồn gốc từ reader và writer. Các lớp thường dùng là FileReader và FileWriter.

#### **Buffered stream**

* Dùng để giảm thiểu thời gian, tăng tốc độ đọc ghi, bằng cách tạo một khoảng nhớ đệm với kích thước cụ thể, không cần truy cập vào ổ cứng.
* Có thể chuyển đổi byte stream và character stream bằng cách sử dụng các wrapper class
* Byte stream: BufferedInputStream, BufferedOutputStream.
* Character stream: BufferedReader, BufferedWriter.
* VD:



### **Serialization và Deserialization**

#### **Serialization**

* Là quá trình chuyển đổi đối tượng java thành các dạng dữ liệu có thể truyền hoặc lưu trữ. Đối tượng sau khi serialization sẽ trở thành một chuỗi byte, được gửi đi qua mạng hoặc lưu trữ vào file.
* Nếu superclass là serializable thì các lớp con của nó sẽ tự động được serializable.
* Khi serialize bất kỳ object nào tham chiếu đến object khác thì java serialization sẽ serialize luôn object đó (nếu object được tham chiếu không triển khai java.io.Serializable interface thì sẽ xảy ra ngoại lệ java.io.NotSerializableException.
* Lưu ý: Không thể serialize các biến static.
* Ưu điểm:
* Truyền dữ liệu qua mạng: Serialization cho phép đối tượng java được chuyển đổi thành dữ liệu có thể chuyển qua mạng một cách dễ dàng. Giúp cho việc tương tác linh hoạt hơn giữa máy chủ và máy khách.
* Lưu trữ trạng thái của đối tượng: dễ dàng được ghi vào file, sau đó được đọc chuyển thành đối tượng java khi cần.
* Chia sẻ dữ liệu giữa các ứng dụng: là công cụ mạnh mẽ để chia sẻ giữa các thành phần khác nhau của ứng dụng hoặc ứng dụng độc lập. Tạo ra kiến trúc linh hoạt và dễ bảo trì.
* Remote Method Invocation: để gọi các phương thức của một đối tượng (object ở xa), từ một máy tính có thể gọi phương thức của một đối tượng đang chạy trên một máy tính khác.
* Duy trì trạng thái ứng dụng: Cho phép tạm dừng và thực thi mà không làm mất dữ liệu.

#### **Deserialization**

* Là quá trình chuyển đổi dữ liệu serialization trở lại thành đối tượng java. Các dữ liệu được đọc từ nguồn như mạng hoặc file, sau đó sẽ chuyển thành đối tượng java.

#### **SerialVersionUID**

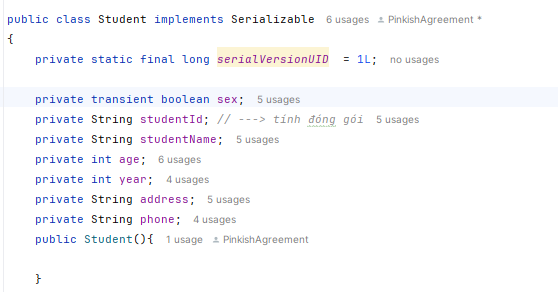
* Là một trường trong một lớp để định danh khi đối tượng trong lớp đó được tuần tự hóa (serialized).
* Để kiểm tra xem phiên bản của lớp hiện tại có khớp với phiên bản đã tuần tự hóa hay không, nếu không khớp sẽ có ngoại lệ được ném ra (InvalidClassException), dẫn đến việc tuần tự hóa sẽ thất bại.

#### **Transient (biến tạm thời)**

* Khi một biến sử dụng transient giá trị của nó sẽ không được bao gồm trong dữ liệu tuần tự hóa. Khi Deserialization, giá trị của biến sẽ được thiết lập lại theo giá trị mặc định theo kiểu dữ liệu của nó.

#### **Externalizable interface**

* Dùng để tùy chỉnh quá trình tuần tự hóa và giải tuần tự hóa theo cách riêng.
* Đòi hỏi phải triển khai phương thức writeExternal và readExternal.
* Lưu ý khi implement:
* Phải có constructor mặc định để giải tuần tự hóa.
* Không có serialVersionUID mặc định, nếu không tự định nghĩa, java sẽ sử dụng cách tính toán dựa trên cấu trúc lớp và có thể dẫn đến việc tương thích ngược.
* VD:

****