**Java 8 - Part 2**

**1. Stream API**

Stream API trong Java 8 là một tính năng quan trọng trong gói java.util.stream, giúp xử lý các collections một cách tiện lợi và hiệu quả theo kiểu lập trình hặm. Thay vì sử dụng vòng lặp lồng nhau và nhiều dòng code phức tạp chúng ta có thể dùng Stream API. Chúng ta có thể biểu diễn thao tác trên tập dữ liệu qua các phương thức xử lý chuỗi gọn gàng hơn.

**1.1 Stream là gì?**

Stream đại diện cho một chuỗi các phần tử dữ liệu có thể được xử lý (chẳng hạn

List, Set, hoặc Array) và các thao tác trên dữ liệu đó. Tuy nhiên, khác với Collection, Stream không lưu trữ dữ liệu mà chỉ xử lý luồng dữ liệu.

**1.2 Tính chất của Stream API**

Không thay đổi Collection ban đầu: Các thao tác trên Stream không làm thay đổi dữ liệu nguồn.

· Lazy execution: Các thao tác chỉ được thực thi khi có kết thúc luồng (terminal

operation).

· Không lưu trữ: Stream không lưu trữ dữ liệu mà chỉ xử lý dữ liệu khi cần.

· Có thể xử lý song song: Stream API hỗ trợ xử lý song song với phương thức

parallelStream(), giúp tăng tốc độ xử lý.

**1.3 Các loại thao tác trong Stream API**

Stream API có hai loại thao tác chính:

· --Intermediate Operations (Các thao tác trung gian): Các thao tác này trả về

một Stream khác và có thể xâu chuỗi nhiều thao tác lại với nhau.

Ví dụ: fijter(), map(), sorted(), distinct(), limit(), v.v.

·-- Terminal Operations (Các thao tác kết thúc): Các thao tác này kết thúc quá

trình xử lý chuỗi và trả về kết quả (có thể là một giá trị hoặc collection). Ví dụ:

forEach(), collect(), reduce(), count(), findFirst(), v.v.

**1.4 Cách sử dụng Stream API**

Một chuỗi các thao tác với Stream API bao gồm ba bước chính:

· Bước 1: Tạo Stream: Bạn có thể tạo một Stream từ các collection hoặc mảng

với phương thức stream() hoặc parallelStream().

· Bước 2: Áp dụng Intermediate Operations: Các thao tác trung gian được áp

dụng để xử lý dữ liệu (lọc, chuyển đổi, sắp xếp, v.v.).

· Bước 3: Áp dụng Terminal Operation: Kết thúc chuỗi thao tác và nhận kết quả

cuối cùng.

// cu

List<Order> orderList = MockUp.orderList();

for (Order order : orderList) {

System.out.println(order);

// moi voi stream

MockUp.orderList().stream().filter(order -> order.getProductType().equals("Laptop")). forEach(order -> {

System out println(order);

});

// filter : lọc

// có sẵn lớp Order đc tạo với các thuộc tính trong đó có thuộc tính productType.

// getProductType là một phương thức get thuộc tính.

// forEach : duyệt trong mảng.

MockUp.orderList(): đây là một danh sách các test case có sẵn cho class order.

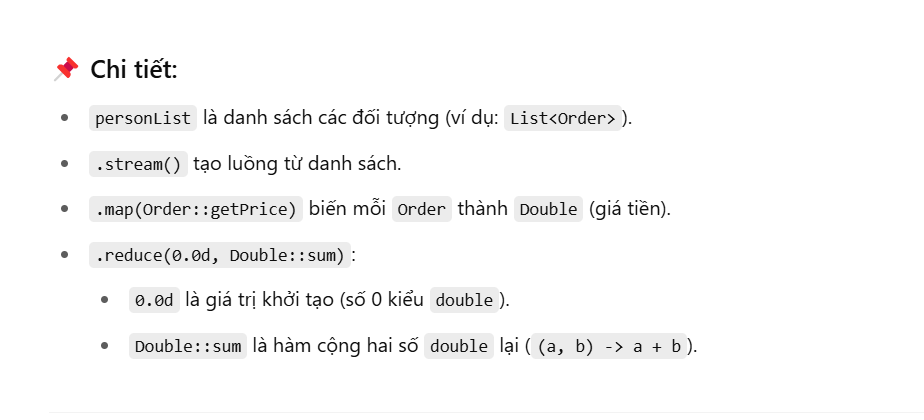
.stream() : để khởi tạo Stream.

--Kỹ năng code hm:

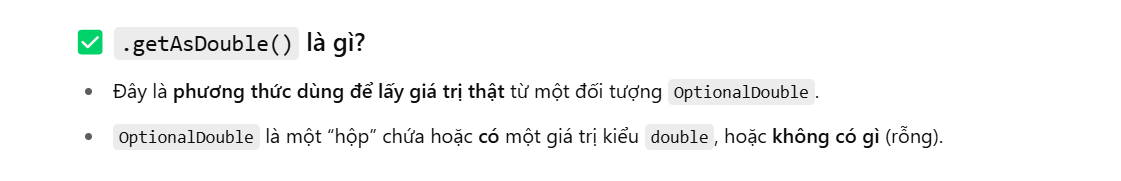
// lay ra toan bo ten nha cung cap ventor  
 List<String> ventor = Mockup.*orderList*().stream().map(Order::getVendor).toList();  
System.*out*.println(ventor);  
// lay ra ten cac nha cung cap khong lap lai  
 List<String> ventor1 = Mockup.*orderList*().stream().map(Order::getVendor).distinct().toList();  
 System.*out*.println(ventor1);

// common()  
List<Order> personList = Mockup.*orderList*();  
double totalSalary1 =  
 personList.stream().mapToDouble(Order :: getPrice).sum();  
System.*out*.println("Sum by Lambda: " + totalSalary1);  
Double totalSalary2 = personList.stream().map(Order :: getPrice)  
 .reduce( 0.0d, Double::*sum*);  
System.*out*.println("Sum by reduce, Double: :sum: " +  
 totalSalary2);  
Double totalSalary3 =  
 personList.stream().map(Order :: getPrice).reduce( 0.0d, (sum, salary) ->  
 sum + salary);  
System.*out*.println("Sum by reduce, Lambda " + totalSalary3);  
Double totalSalary4 =  
 personList.stream().collect(Collectors. *summingDouble*(Order::getPrice));  
  
System.*out*.println("Sum by Collectors. summingDouble: " +  
 totalSalary4);  
System. *out*.println("\n -- Max salary -- ");  
System.*out*.println(personList.stream().mapToDouble(Order :: getPrice).max().getAsDouble());  
System.*out*.println("\n -- Min salary -- ");  
System. *out*.println(personList.stream().mapToDouble(Order :: getPrice).min().getAsDouble());  
System.*out*.println("\n -- Average salary -- ");  
System.*out*.println(personList.stream().mapToDouble(Order :: getPrice).average().getAsDouble());

Giải thích cách 2 tính lương bằng reduce:





 Ko có .getAsDouble() nó xin ra OptionalDouble[số gì đó], có .getAsInt(), getAsFloat, getAsLong()…..

Method Stream:

---**allMatch**: trả về true nếu tất cả các dk mình xét là đúng, nếu sai 1 cái là sai.

**Tất cả** phần tử thỏa mãn điều kiện thì trả về true

Một phần từ không tm thì false.

Vd : kiểm tra trong list Order có phải tất các các product đều có type là Mobile hay không, thì nó in ra là false;

Nếu tất cả là Mobile nó sẽ in ra true;

boolean isMobile = Mockup.*orderList*().stream().allMatch(order -> Objects.*equals*(order.getProductType(), "Mobile"));  
System.*out*.println(isMobile);  
// in ra false

--**anyMatch**: đây là kiểu booean , trả về true nếu dk mình xét chỉ cần có xuất hiện 1 lần -> in ra true, không xuất hiện lần nào in ra false.

Có **ít nhất một** phần tử thỏa mãn điều kiện thì trả về true

Ko có thì là false

boolean isMobile = Mockup.*orderList*().stream().anyMatch(order -> Objects.*equals*(order.getProductType(),"Watch"));  
System.*out*.println(isMobile);  
// in ra true;  
boolean check = Mockup.*orderList*().stream().anyMatch(order -> Objects.*equals*(order.getProductType(), "type khong co trong danh sach"));  
System.*out*.println(check);  
// in ra false

--**noneMatch()** : đây là hàm kiểu boolean, dùng để kiểm tra xem không có phần từ nào trong stream thoả mãn điều kiện được cung cấp hay không. Nếu không có phần tử nào thoả mãn thì trả về true, ngược lại thì trả về false.

**Không có bất kỳ** phần tử nào thỏa mãn điều kiện thì trả về true

Có 1 thì là false

Boolean noneMath(Predicate<> super T> predicate)

predicate: điều kiện kiểm tra ( hàm lambda hoặc method reference).

boolean check1 = Mockup.*orderList*().stream().noneMatch(order -> order.getPrice() > 1000);  
System.*out*.println(check1);  
// khong co hang nao gia tren 1 trieu nen no tra ve true

**--String.Builder** là một cách để tạo Stream bằng tay, giống như bạn xây dựng từng phần rồi mới chạy stream.

Nó thuộc về java.util.stream.Stream

Cú pháp :

Stream.Builder<T> builder = Stream.builder();

Builder.accept(….)

Stream<T> stream = builder.build();

+ Dùng Stream.Builder khi:

Khi bạn không có sẵn danh ách hoặc mảng, mà muốn thêm từng phần tử một cách linh hoạt, có thể phụ thuộc logic điều kiện.

Ví dụ đơn giản :

import java.util.stream.Stream;

public class StreamBuilderExample {

public static void main(String[] args) {

Stream.Builder<String> builder = Stream.builder();

builder.accept("Java");

builder.accept("Spring");

builder.accept("Stream");

Stream<String> stream = builder.build();

stream.forEach(System.out::println);

}

}

**Giải thích :**

**accept()** : dùng để thêm phần tử vào trong Stream.Builder

build() : dùng để kết thúc và trả về một Stream hoàn chình, khi build thì ko thể accept đc nữa. Sau khi build(), bạn sẽ có thể thao với vs stream như stream bth : .forEach(), .map, .filter()…

----**Stream.concat():**là một phương thức để nối các Stream lại với nhau thành một stream duy nhất.

Ví dụ:

//Stream.concat()  
Stream<String> stream1 = Stream.*of*("Alpha");  
Stream<String> stream2 = Stream.*of*("Beta");  
Stream<String> stream3 = Stream.*of*("Xeta");  
Stream.*concat*(Stream.*concat*(stream1,stream2) , stream3).forEach(s -> System.*out*.print(s + " "));

----**Stream.of:** là một phương thức tĩnh (static method), được dùng để tạo nhanh một đối tượng chứa các phần tử cố định.

-------- **.map()** : biến đổi từng thành phân trong Stream, ví dụ như : chuyển đổi giá trị : viết hoá , nhân số, đổi tên). DÙNG ĐỂ BIẾN ĐỔI DỮ LIỆU Và CŨNG CÓ THỂ LÀ DÙNG ĐỂ LẤY DỮ LIỆU (thuộc tính .. ) ( Như ví dụ 4)

Dùng trong các xử lý trung gian của stream: trước khi .collect(), forEach(). …

Ví dụ1:

Stream<Integer> stream1 = Stream.*of*(1,2,3);  
stream1.map(n -> n \* 2).forEach(n -> System.*out*.print(n + " "));

Biến các số 1 2 3 thành 2 4 6. Chạy chương trình trên sẽ in ra 2 4 6.

Ví dụ 2: List<String> result = Stream.of("html", "css", "java", "react", "go")

.filter(s -> s.length() > 3) // lọc từ có độ dài > 3

.map(String::toUpperCase) // viết hoa toàn bộ

.toList()// thu về dạng List

System.out.println(result);

IN ra [HTML. JAVA, REACT].

Ví dụ 3:

List<String> emails = Stream.of("a@gmail.com", "b@yahoo.com", "admin@gmail.com")

.filter(email -> email.endsWith("@gmail.com"))

.map(email -> email.toUpperCase())

.collect(Collectors.toList());

System.out.println(emails);

In ra : [A@GMAIL.COM, [ADMIN@GMAIL.COM](mailto:ADMIN@GMAIL.COM)]

Ví dụ 4 :

// lay ra toan bo ten nha cung cap ventor  
 List<String> ventor = Mockup.orderList().stream().map(Order::getVendor).toList();  
 System.out.println(ventor);

Ví dụ 6:

List<String> words = List.of("apple", "banana", "cherry");

List<List<Character>> result = words.stream()

.map(word -> word.chars() // Chuyển mỗi từ thành stream của các ký tự

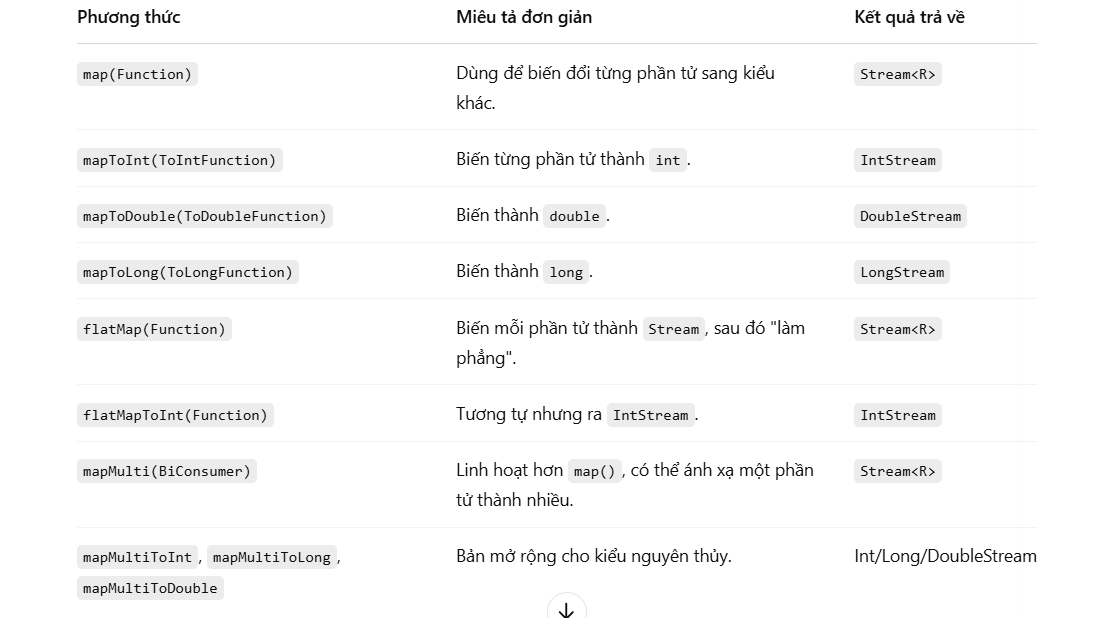
.mapToObj(c -> (char) c) // Chuyển int thành char

.collect(Collectors.toList())) // Tạo danh sách ký tự

.collect(Collectors.toList());

System.out.println(result);

In ra : [[a, p, p, l, e], [b, a, n, a, n, a], [c, h, e, r, r, y]]





--**flatMap():** dùng để biến đổi từng phần tử thành một Stream mới, sau đó gộp tất cả các Stream con thành một Stream duy nhất.

List<String> words = List.of("apple", "banana", "cherry");

List<Character> result = words.stream()

.flatMap(word -> word.chars() // Chuyển mỗi từ thành stream của các ký tự

.mapToObj(c -> (char) c)) // Chuyển int thành char

.collect(Collectors.toList());

System.out.println(result);

In ra : [a, p, p, l, e, b, a, n, a, n, a, c, h, e, r, r, y]

Ví dụ 2:

List<Order> orders = Mockup.*orderList*();  
List<String> productName = orders.stream().map(Order :: getProductName).collect(Collectors.*toList*());  
List<String> productType = orders.stream().map(Order :: getProductType).collect(Collectors.*toList*());  
List<List<String>> product = new ArrayList<>();  
product.add(productName);  
product.add(productType);  
System.*out*.println("Distinct all product type");  
product.stream(). flatMap(list -> list.stream().distinct()).forEach(System.*out*::println);  
System. *out*.println("\nTotal price");  
double total = Mockup.*orderList*().stream().map(Order :: getPrice).flatMapToDouble(DoubleStream::*of*). sum();  
System.*out*.println(total);  
System.*out*.println("Print first character");  
 productName.stream().flatMap(s ->  
 Stream.*of*(s.toUpperCase().charAt(0))).forEach(System.*out* :: println);

--- **findFirtst():** tìm phần tử đầu tiên nếu có.

--**findAny()** : là một terminal operation (hàm kết thúc) của Stream. Nó trả về một phần tử bất kỳ từ Stream (thường dùng sau khi .filter() ). Kết quả là Optional<T> -> có thể chứa giá trị, hoặc rỗng nếu không tìm thấy.

Ví dụ 1 :

List<String> names = List.of("Alice", "Bob", "Charlie");

Optional<String> anyName = names.stream().findAny();

anyName.ifPresent(System.out::println); // In ra một tên bất kỳ

Ví dụ 2 :

// su dung findAny  
List<String> type = Mockup.*orderList*().stream().map(Order::getProductType).distinct().toList();  
Optional<String> check = type.stream().findAny();  
if(check.isPresent()){  
 type.forEach(System.*out*::println);  
}  
else{  
 System.*out*.println("type khong tom tai");  
}

--- **Optional<T>:** là một lớp bọc giá trị (wrapper) trong Java : nhắm tránh lỗi NullPoiterException (lỗi null). Làm code rõ ràng hơn khi xử lý giá trị có thể có hoặc không có.

**Method của Optional<T>**, T : kiểu dữ liệu : String , Integer, Float, Double, Long…

1**. isPresent()** : kiểm tra xem Optional có chứa giá trị hay không.

Optional<String> name = Optional.of("Alice");

if (name.isPresent()) {

System.out.println("Có giá trị: " + name.get());

}

->Trả về true nếu có giá trị, ngược lại là false

2**. ifPresent(Consumer<T> action)** : thực hiện hành động nếu giá trị tồn tại (không phải null).

Optional<String> name = Optional.of("Alice");

name.ifPresent(n -> System.out.println("Tên: " + n));

Ví dụ :

List<String> name = new ArrayList<>(Arrays.*asList*("hoang", "an"));  
 List<String> name1 = new ArrayList<>();  
Optional<List> x = Optional.*of*(name1);  
x.ifPresent(n -> System.*out*.print(n + " "));

Không in ra gì, vì List name1 không có gì.

3**. get():** lấy giá trị trong Optional, nếu có. Nếu không sẽ ném ra NoSuchElementException.

Optional<String> name = Optional.of("Alice");

System.out.println(name.get()); // In ra "Alice"

4. **orElse (T other):** trả về giá trị nếu có, hoặc trả về giá trị default nếu Optional rỗng.

Optional<String> name = Optional.of("Alice");

String result = name.orElse("Default Name");

System.out.println(result); // In ra "Alice"

5. **orElseGet (Supplier<T> other)** : tương tự orElse(), nhưng thay vì truyền giá trị trực tiếp, bạn có thể cung cấp một hàm tạo giá trị (supplier) khi Optional rỗng.

Optional<String> name = Optional.empty();

String result = name.orElseGet(() -> "Generated Name");

System.out.println(result); // In ra "Generated Name"

6**. orElseThrow (Supplier<? Extends Throwable> exceptionSupplier)** : trả về giá trị trong Optional nếu có, nếu không, ném ra exception mà bạn cung cấp.

Optional<String> name = Optional.empty();

String result = name.orElseThrow(() -> new IllegalArgumentException("Không có tên!"));

----- .sorted() : được sử dụng để sắp xếp các phấn tử trong stream. Có hai cách để sử dụng:

1. Sắp xếp theo thứ tự tự nhiên:

1.1 Tăng dần:

List<Integer> numbers = Arrays.asList(5, 3, 8, 1);

numbers.stream()

.sorted() // Sắp xếp theo thứ tự tự nhiên (tăng dần)

.forEach(System.out::println);

In ra : 1 3 5 6

1.2 Giảm dần:

Ví dụ 1:

List<Integer> numbers = Arrays.asList(5, 3, 8, 1);

numbers.stream()

.sorted(Comparator.reverseOrder()) // Sắp xếp giảm dần

.forEach(System.out::println);

In ra 8 5 3 1

Ví dụ 2:

List<String> names = Arrays.asList("John", "Alice", "Bob", "Diana");

names.stream()

.sorted((a, b) -> b.compareTo(a)) // Sắp xếp theo thứ tự giảm dần

.forEach(System.out::println);

In ra : John Diana Bob Alice

// dung sorted()  
// sap xep tang dan theo price c1  
List<Order> list = Mockup.*orderList*();  
list.stream().sorted((o1,o2) -> Double.*compare*(o1.getPrice(), o2.getPrice())).  
 forEach(p -> System.*out*.println(p.getProductName() + " " + p.getPrice()));  
// sap xep giam dan theo price c1  
System.*out*.println("-----------------------");  
list.stream().sorted((o1,o2) -> Double.*compare*(o2.getPrice(), o1.getPrice())).  
 forEach(p -> System.*out*.println(p.getProductName() +" " + p.getPrice()));  
// sap xep tang dan theo price c2  
System.*out*.println("----------------------");  
list.stream().sorted(Comparator.*comparing*(Order::getPrice)).forEach(p -> System.*out*.println(p.getProductName() + " " + p.getPrice()));  
// sap xep giam dan theo price c2  
System.*out*.println("--------------------------------");  
list.stream().sorted(Comparator.*comparing*(Order::getPrice).reversed()).forEach(p -> System.*out*.println(p.getProductName() + " " + p.getPrice()));  
// sap xep tang dan theo price neu price bang nhau sap giam tang dan theo productName  
System.*out*.println("------------------------");  
list.stream().sorted(Comparator.*comparing*(Order::getPrice).thenComparing(Order::getProductName)).  
 forEach(p -> System.*out*.println(p.getProductName() + " " + p.getPrice()));  
System.*out*.println("-----------------------");  
// sap xep giam dan theo price neu price bang nhau thi sap xep giam dan theo productName();  
list.stream()  
 .sorted(  
 Comparator.*comparing*(Order::getPrice).reversed()  
 .thenComparing(Comparator.*comparing*(Order::getProductName).reversed())  
 )  
 .forEach(p -> System.*out*.println(p.getProductName() + " " + p.getPrice()));

---- stream.skip(n): được dùng để bỏ qua n phần tử đầu tiên trong stream.

Ví dụ 1:

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Diana", "Eve");

names.stream()

.skip(2) // Bỏ qua 2 phần tử đầu: "Alice", "Bob"

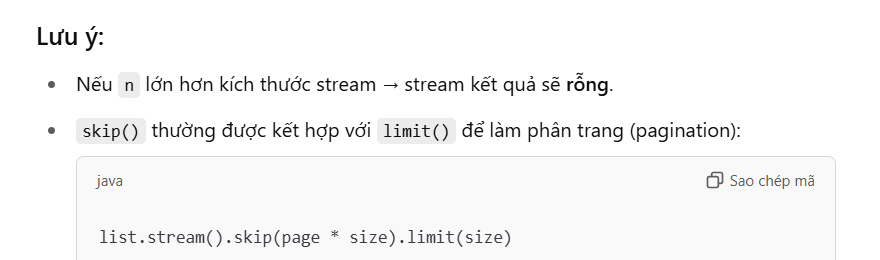
.forEach(System.out::println);

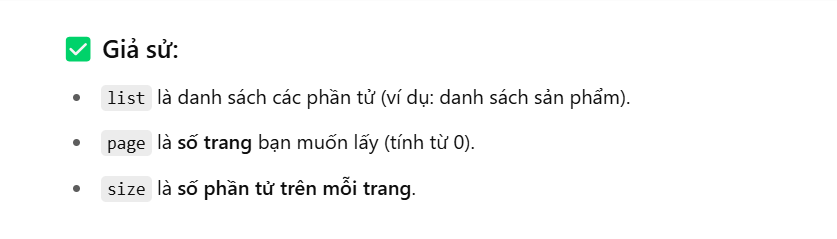
IN ra:

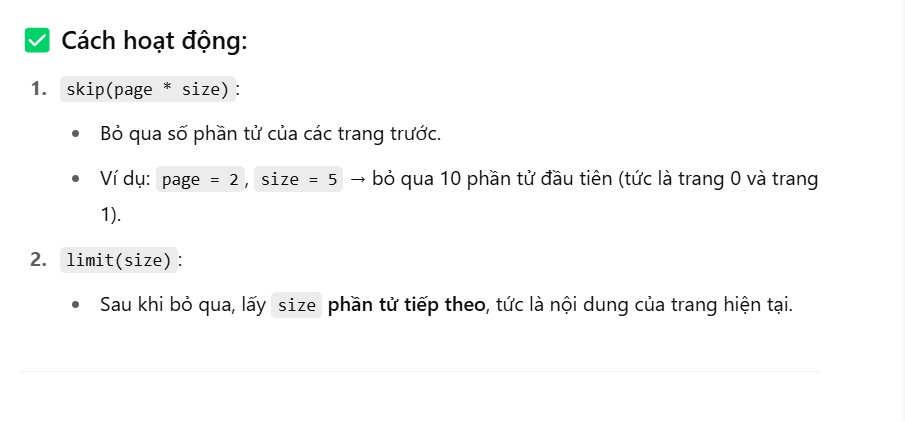
Charlie

Diana

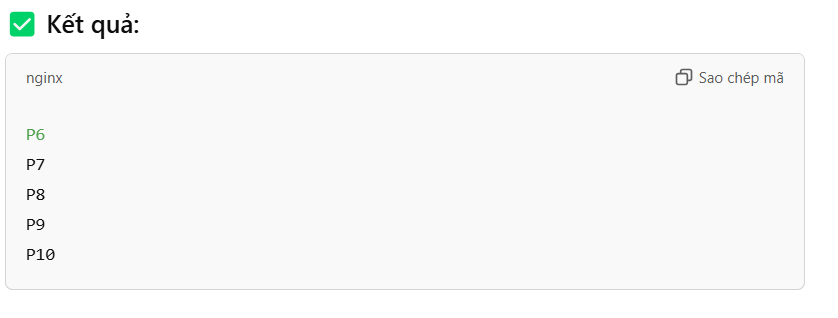
Eve

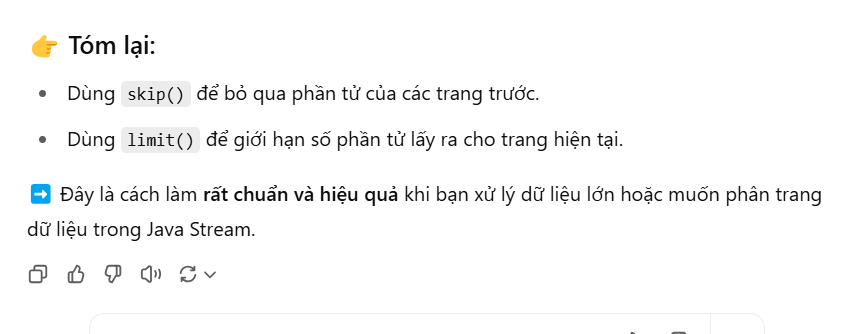




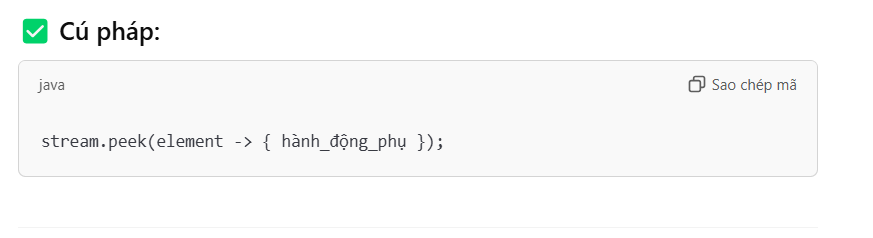








----- peek(): là một phương thức trung gian trong Java Stream, được dùng để quan sát(hoặc kiểm tra) các phần tử trong quá trình xử lý mà không làm thay đổi stream**. peek()** giúp **“nhìn trộm”** các phần tử trong stream, thường dùng để **debug (in ra)** dữ liệu giữa các bước xử lý.



Ví dụ:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()

.filter(n -> n % 2 == 0) // Lọc số chẵn

.peek(n -> System.out.println("Found: " + n)) // In ra số chẵn

.map(n -> n \* n) // Bình phương

.forEach(System.out::println); // In kết quả

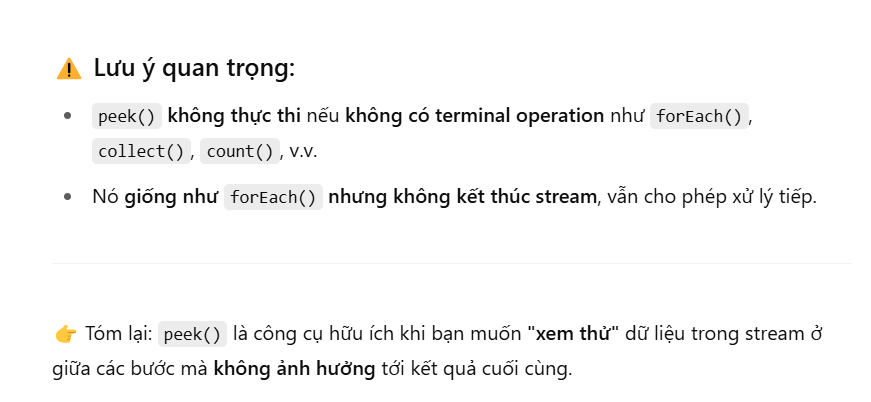
Kết quả:

Found: 2

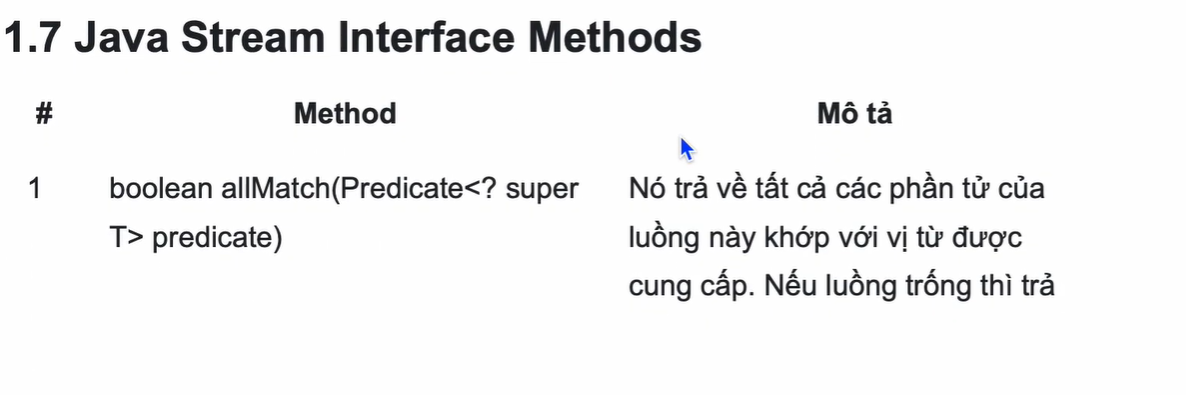
4

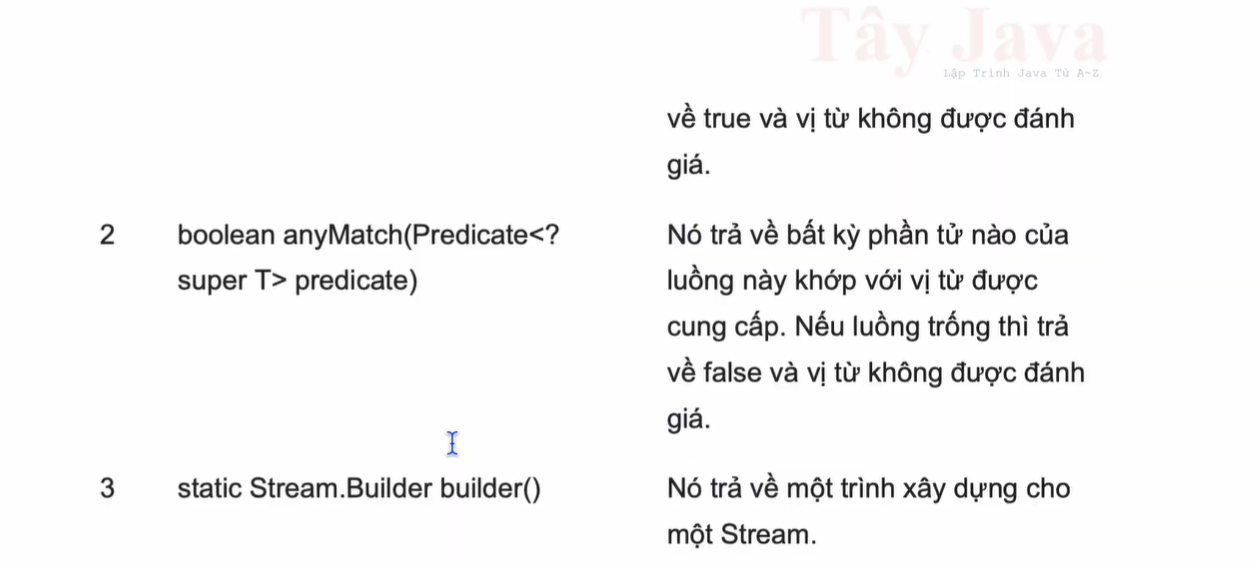
Found: 4

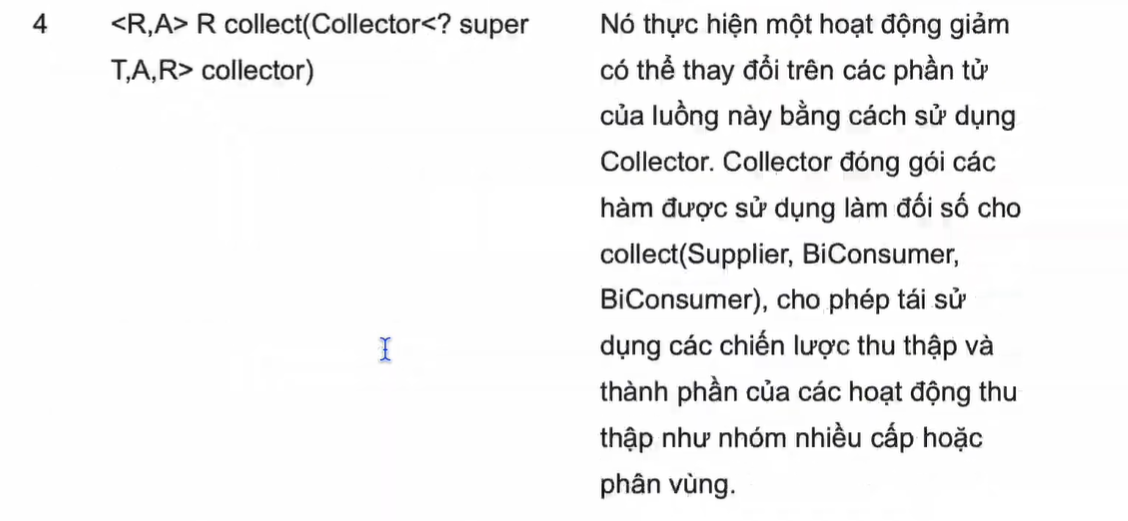
16

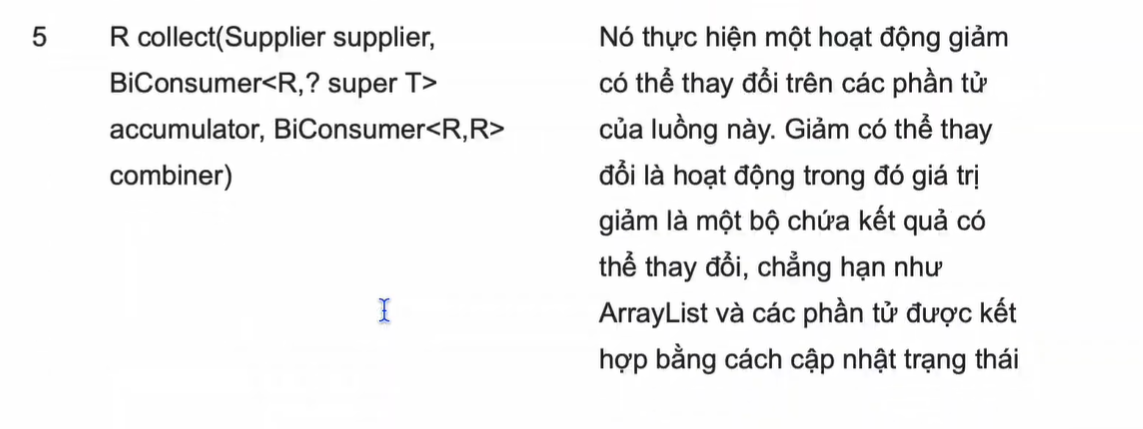


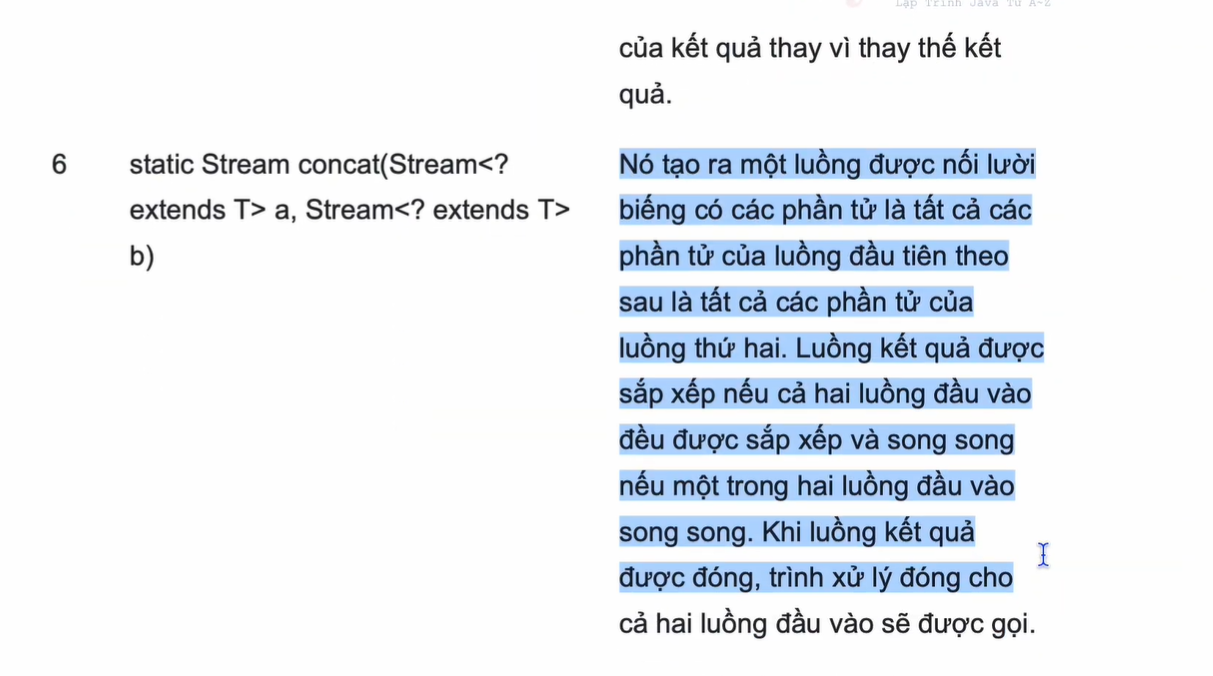


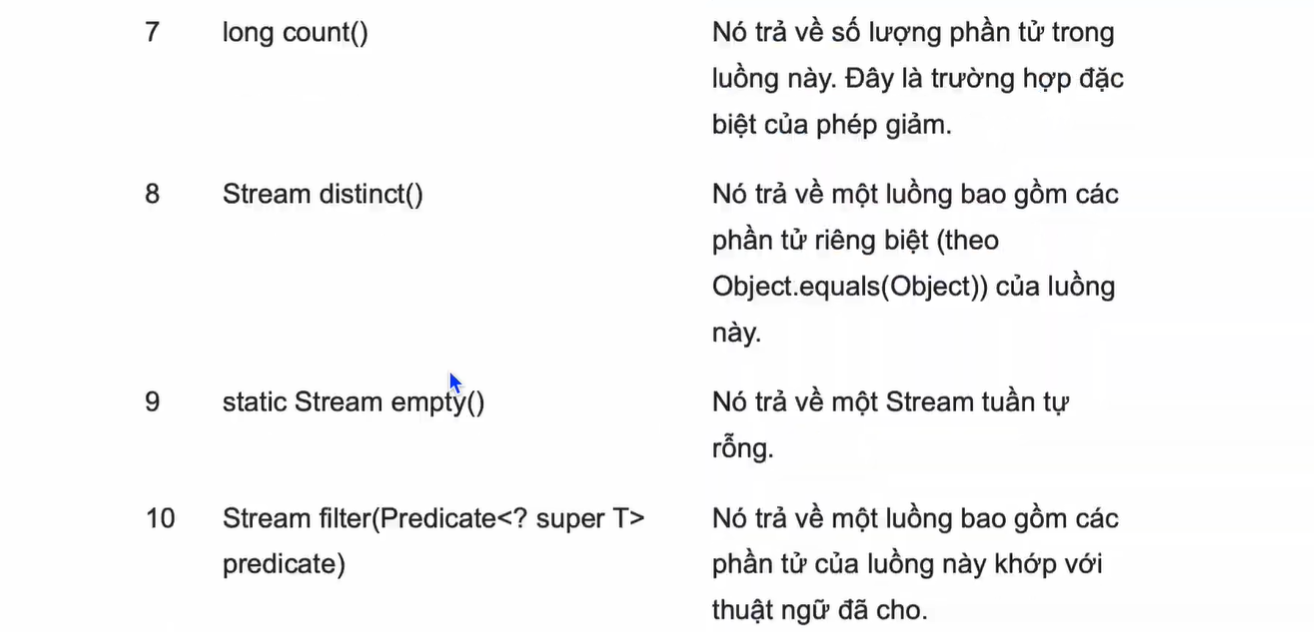


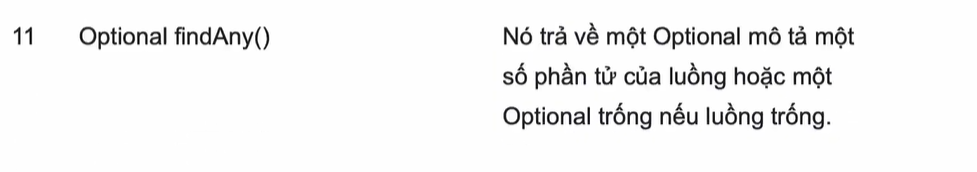


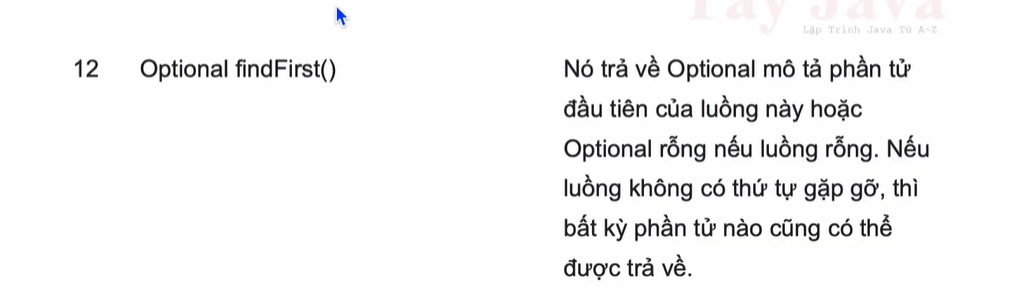


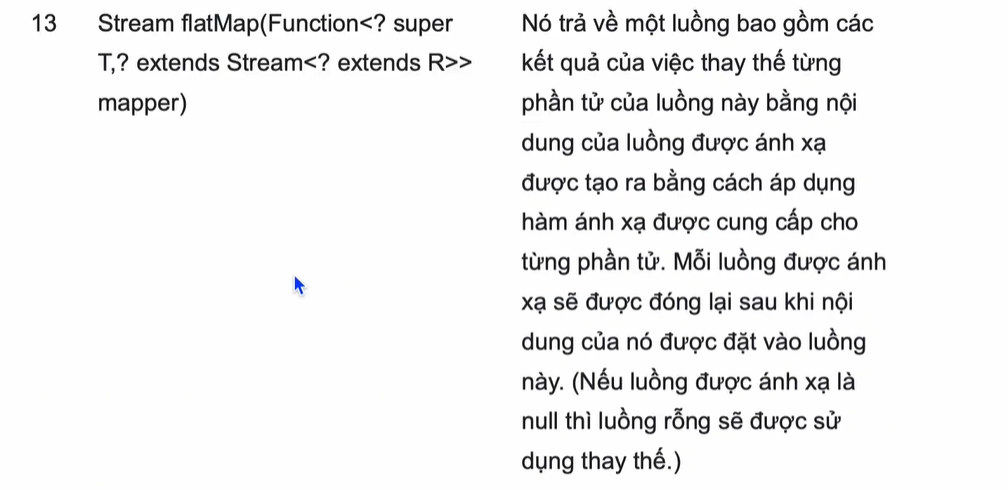


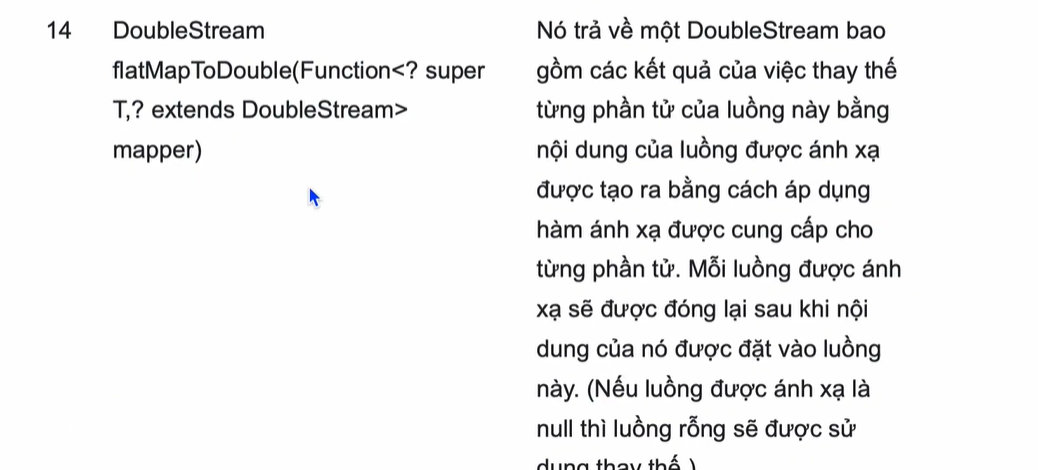


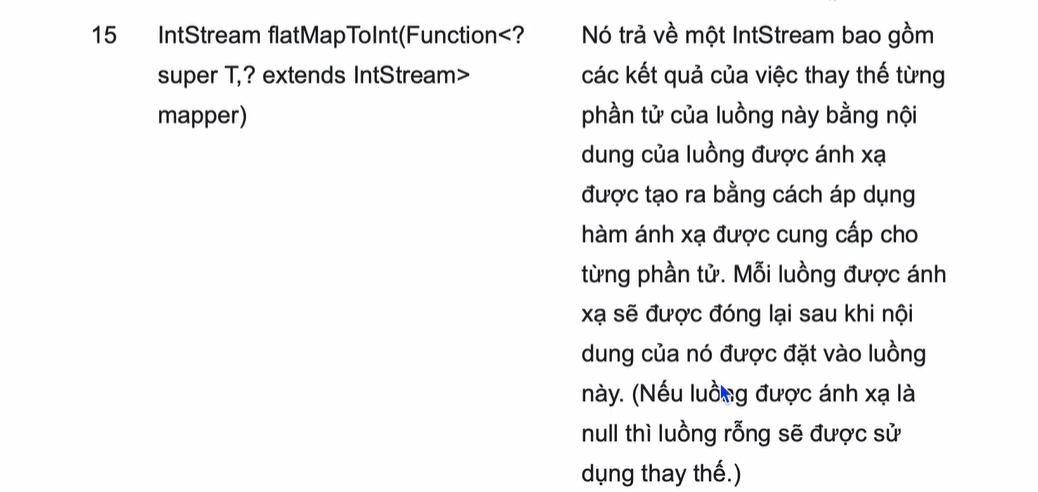


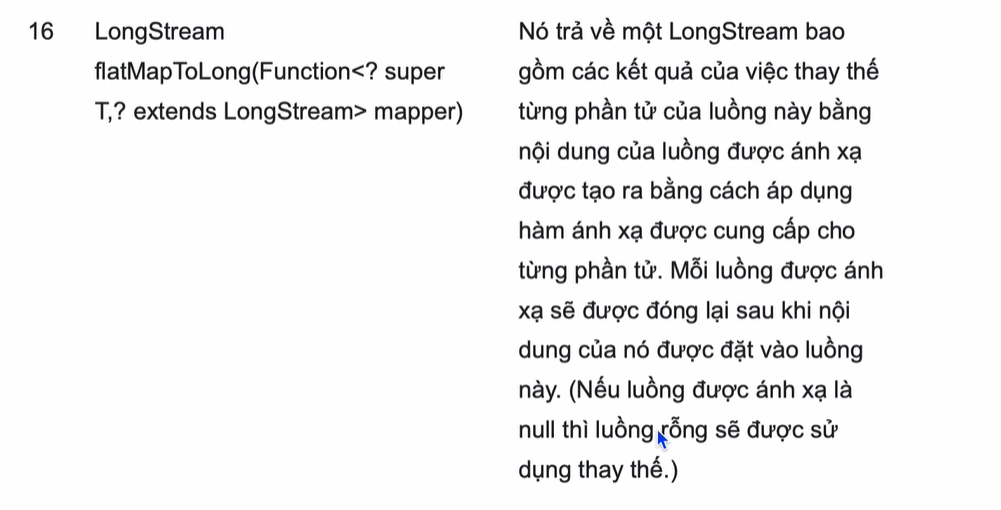


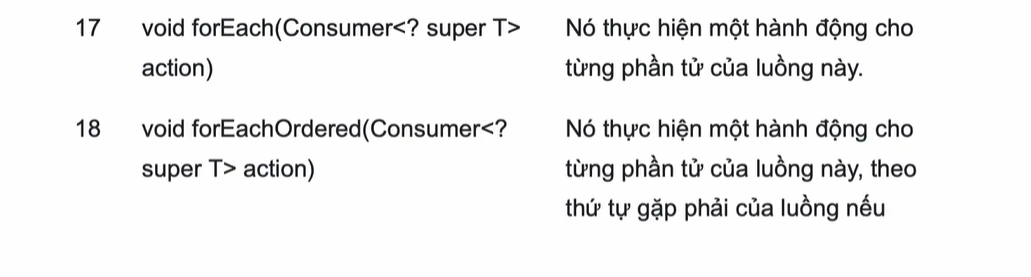


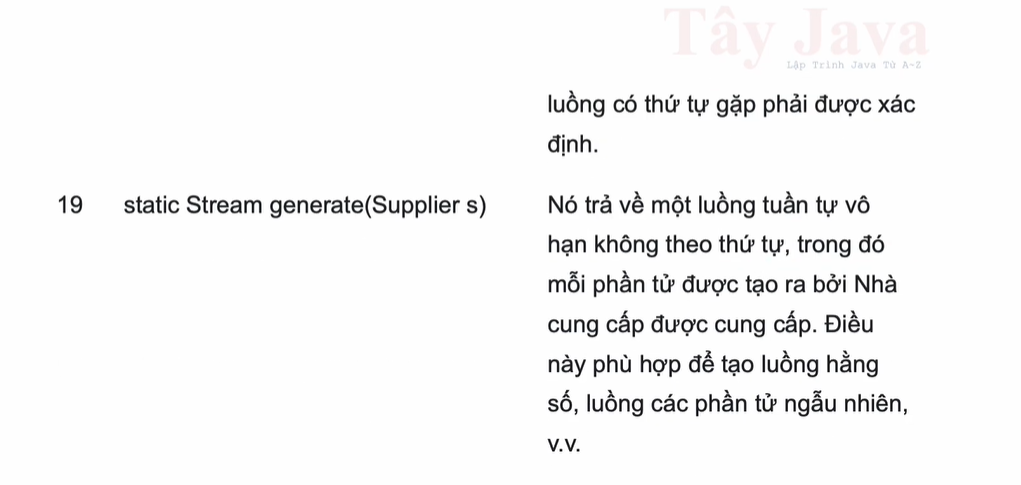


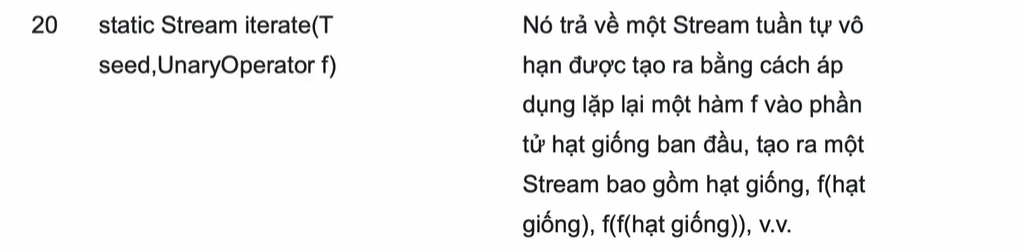


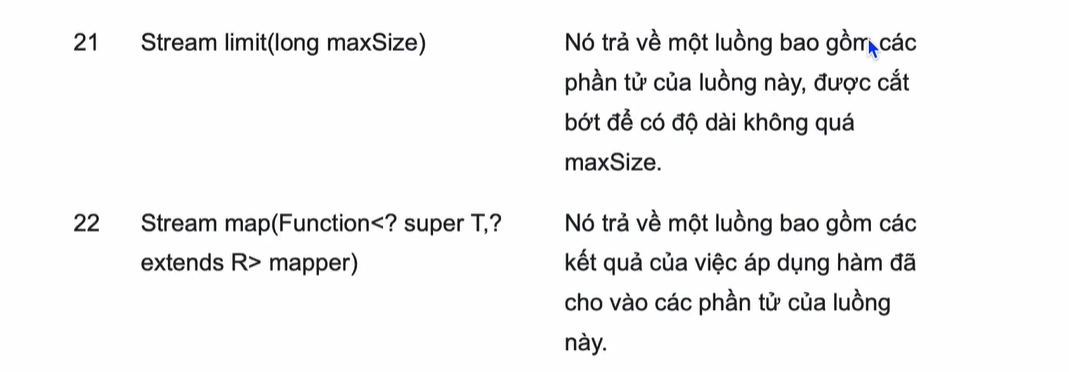


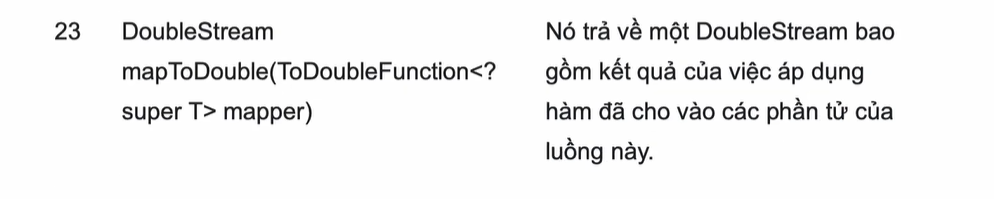


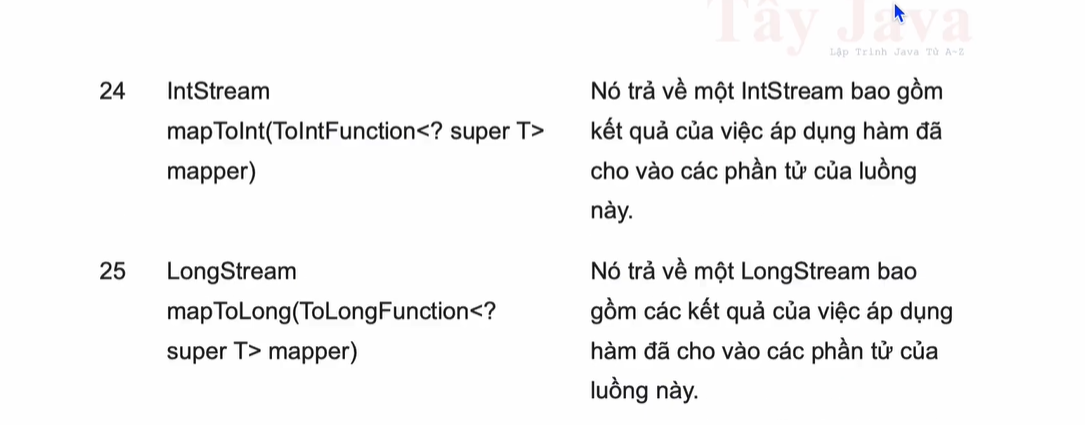


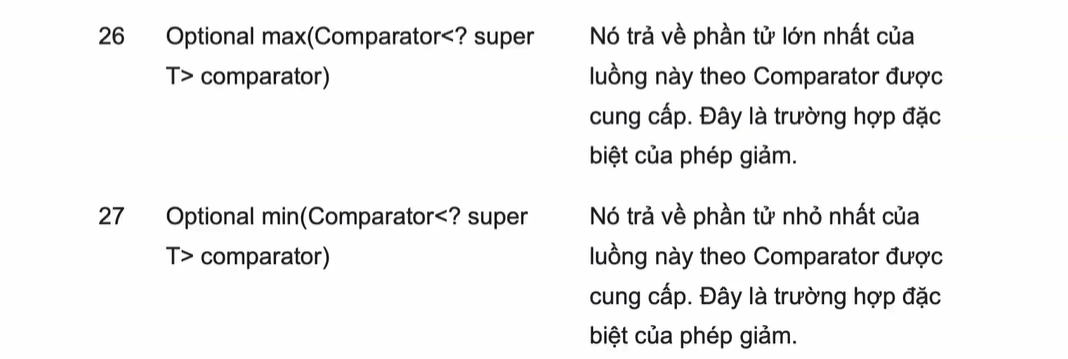


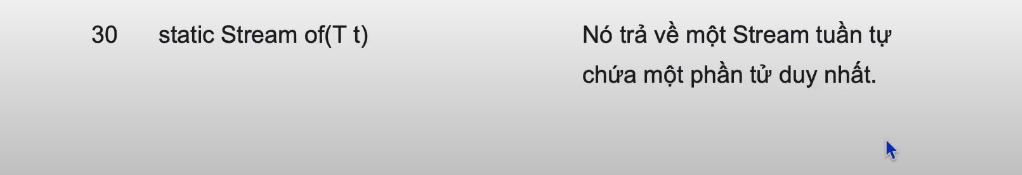




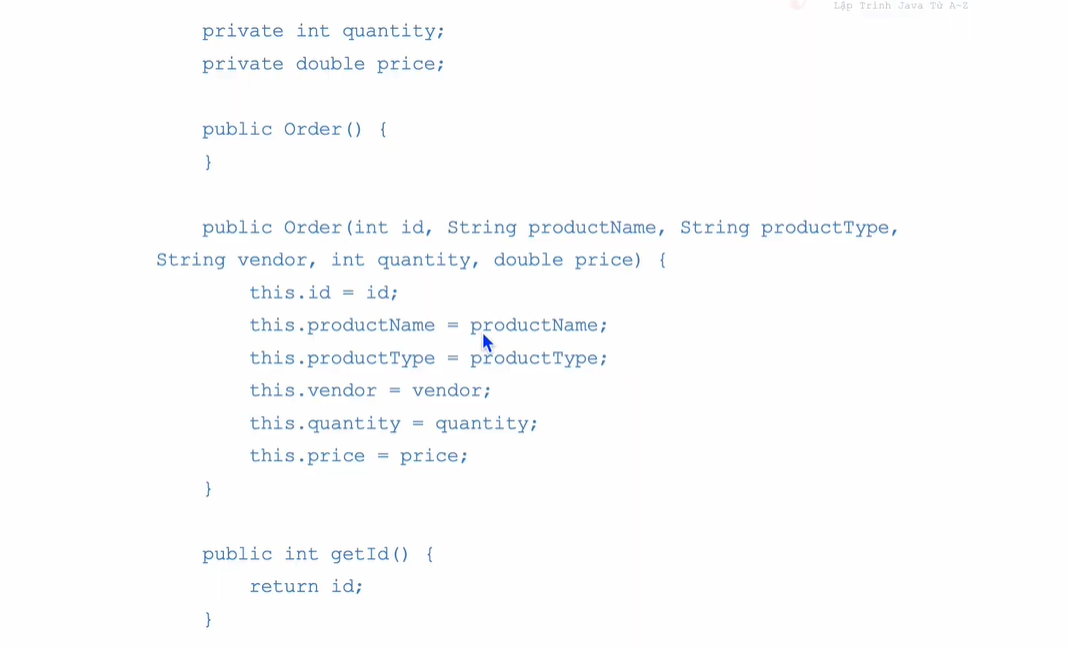


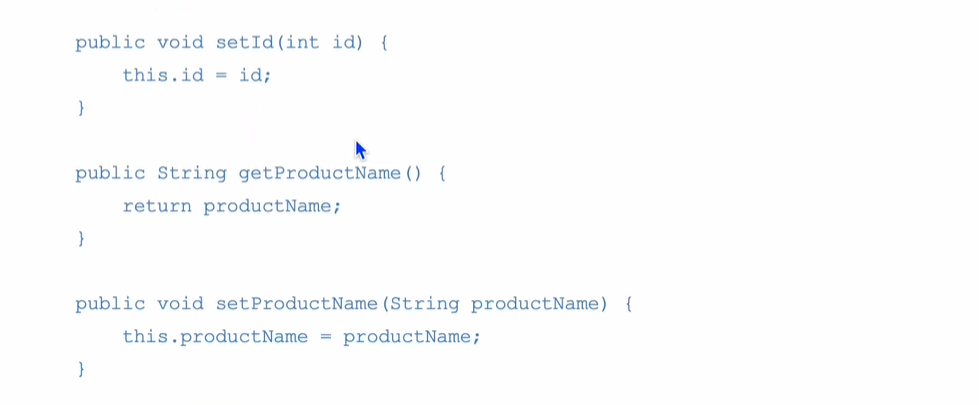


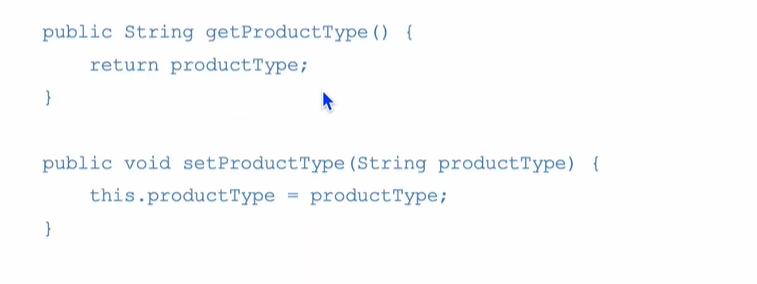












Get set các thuộc tính

