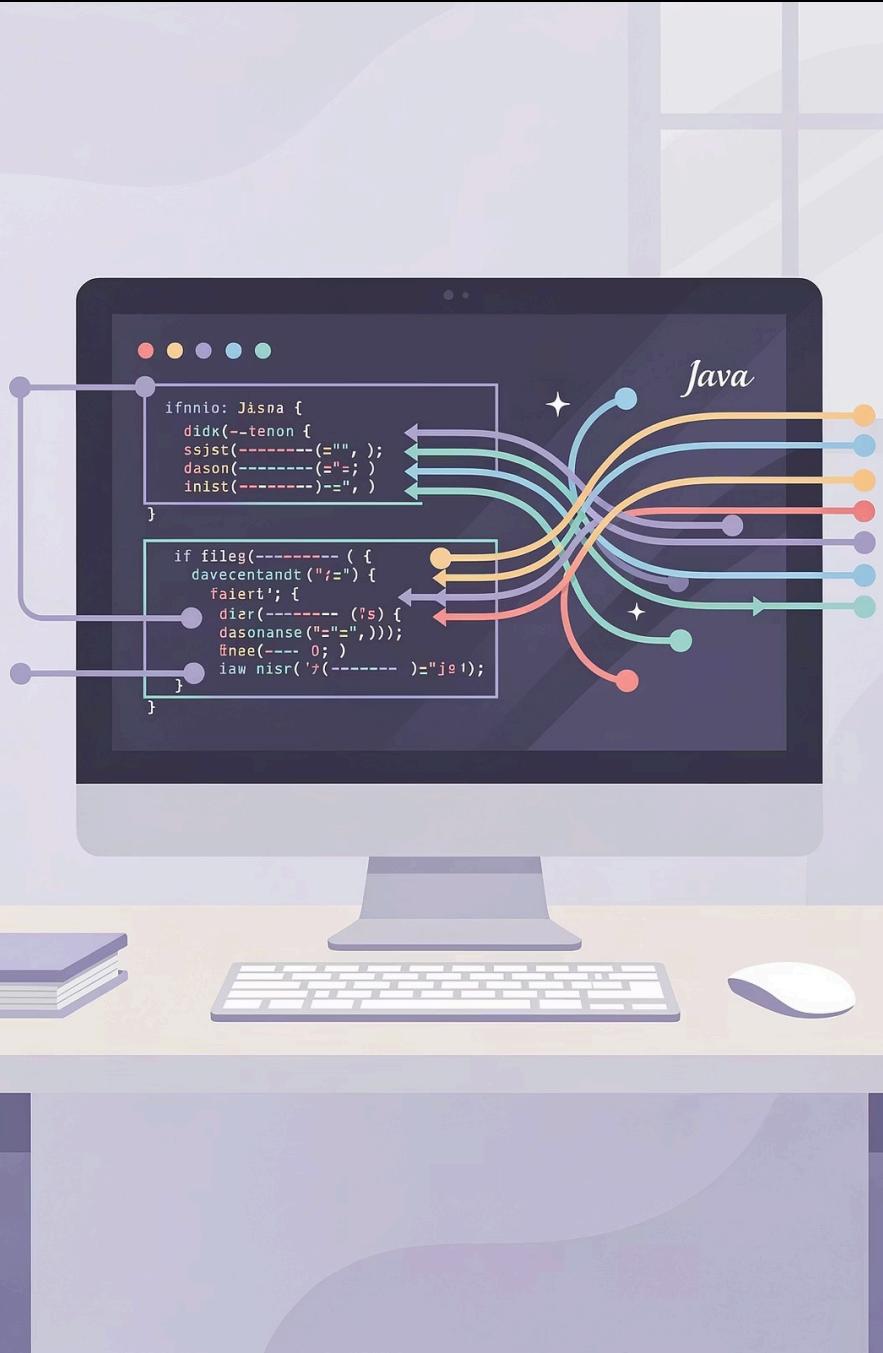


CHƯƠNG 6: COLLECTIONS & STREAM API – HIỆN ĐẠI HÓA JAVA

MỤC TIÊU HỌC TẬP

Sau khi hoàn thành chương này, bạn sẽ:

- Sử dụng thành thạo Java Collections Framework
- Hiểu và áp dụng Generics
- Viết Lambda Expressions
- Sử dụng Functional Interfaces
- Xử lý dữ liệu với Stream API
- Chuyển đổi từ Imperative sang Declarative programming





KIẾN THỨC CÂN CÓ (PREREQUISITES)

Trước khi học chương này, bạn cần nắm vững các kiến thức nền tảng sau:

1

Mảng (Arrays)

Chương 1, phần 0.7

- Khai báo, truy cập, duyệt mảng
- Hiểu hạn chế của mảng (kích thước cố định)

2

Wrapper Classes ★ QUAN TRỌNG

Chương 1, phần 0.13

- Integer, Double, String, Boolean, ...
- Autoboxing và Unboxing
- **Collections chỉ chấp nhận Wrapper Classes, không chấp nhận primitives!**

3

Interface

Chương 3, phần 3.4

- Định nghĩa và implement interface
- Functional Interfaces

4

Method

Chương 1, 2

- Cách định nghĩa và gọi method
- Tham số, return type

Lưu ý quan trọng: Nếu bạn chưa học **Wrapper Classes** ở Chương 1, hãy quay lại học ngay! Collections không thể dùng primitives. Chương này sẽ **hiện đại hóa** cách bạn xử lý dữ liệu trong Java.

6.1. JAVA COLLECTIONS FRAMEWORK

6.1.1. Collections là gì?

Định nghĩa

Collections Framework là tập hợp các interfaces và classes để lưu trữ và xử lý nhóm các objects một cách hiệu quả và có tổ chức.

Lợi ích chính

- ✓ Không cần tự implement cấu trúc dữ liệu
- ✓ Tối ưu hiệu suất
- ✓ Code chuẩn, dễ đọc
- ✓ Tiết kiệm thời gian phát triển

Nhắc lại từ Chương 1

- Mảng có kích thước cố định
- Khó thêm/xóa phần tử
- Collections giải quyết vấn đề này
- Chỉ chấp nhận Wrapper Classes

Tưởng tượng: Hộp dụng cụ (The Toolbox)



List

Giống **Danh sách đi chợ**. Thứ tự quan trọng (mua trước hay sửa sau), và có thể ghi 2 lần "sữa" (duplicate) nếu cần mua nhiều.



Set

Giống **Danh sách khách mời đám cưới**. Không được mời ai 2 lần (no duplicate), thứ tự không quan trọng.

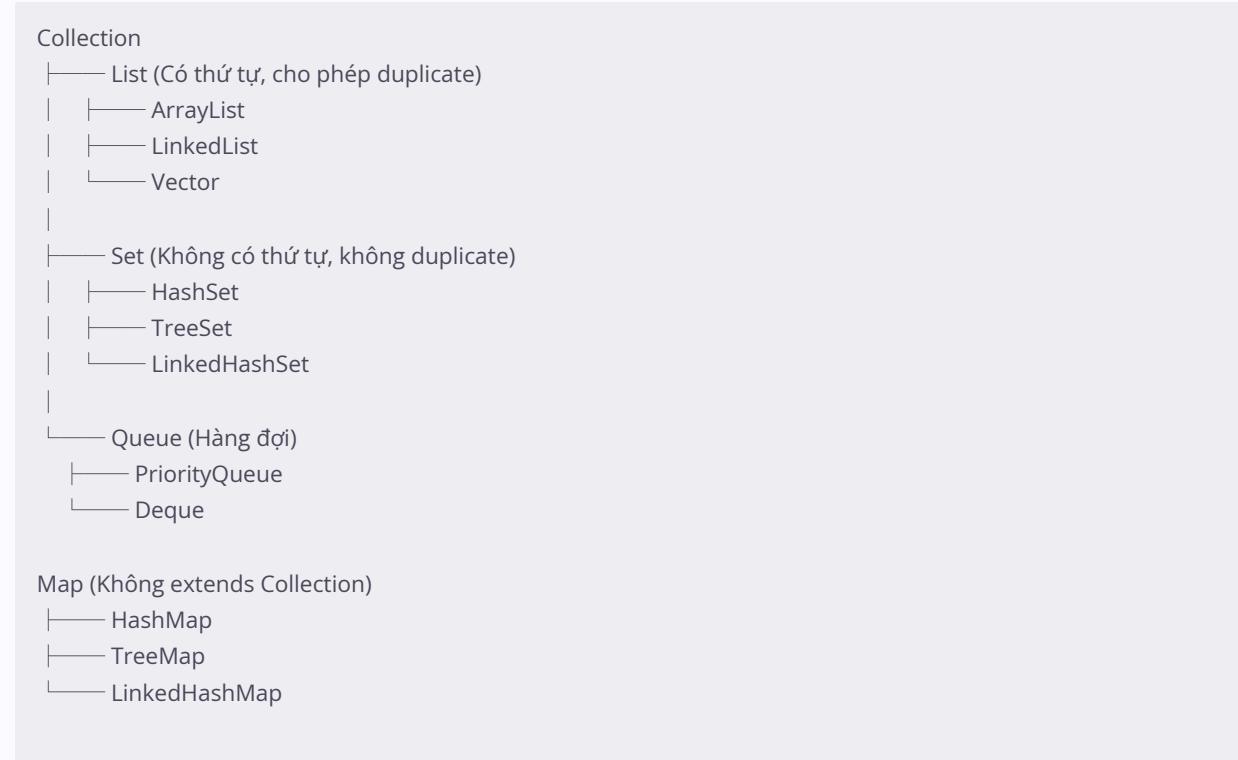


Map

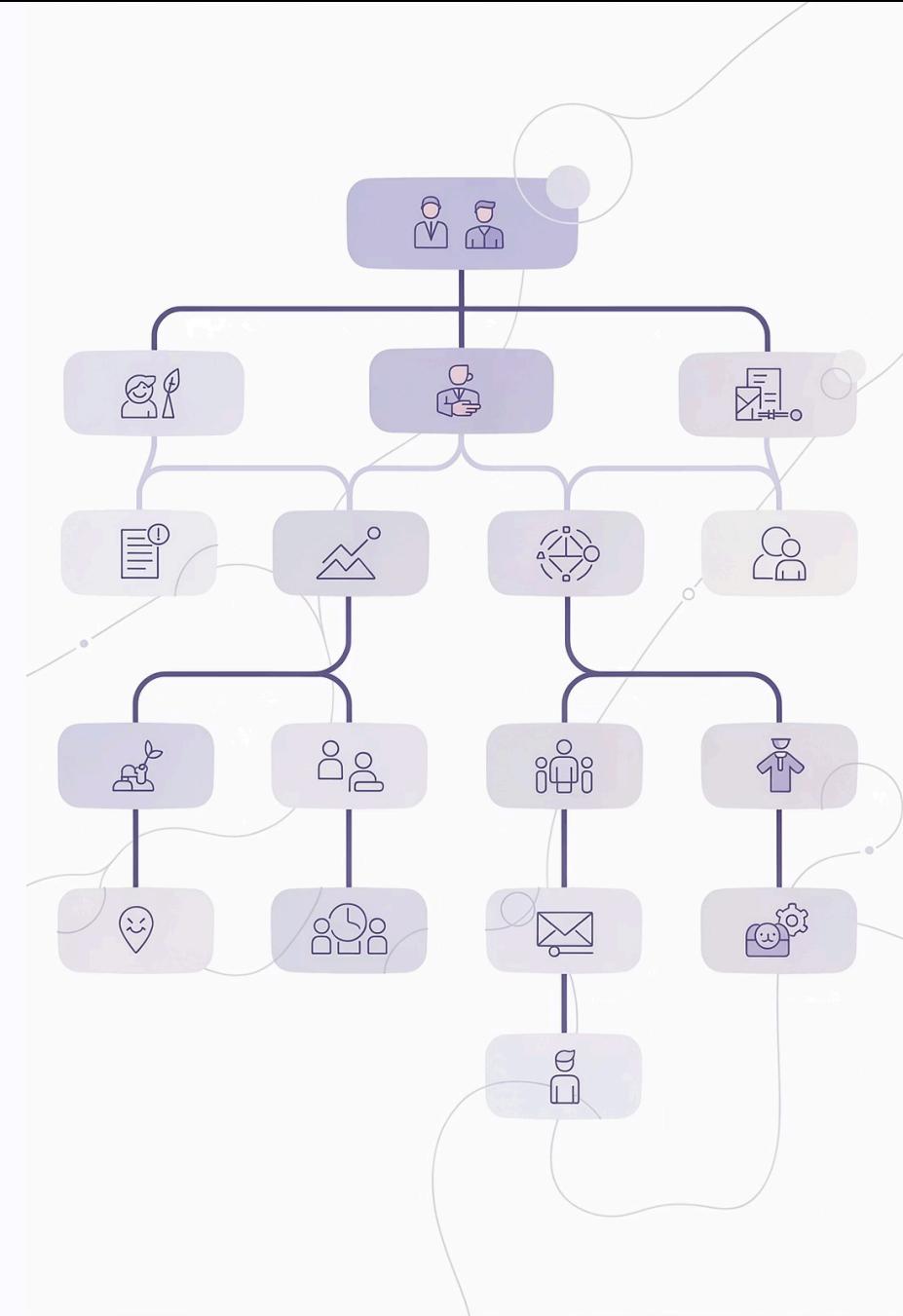
Giống **Danh bạ điện thoại**. Tra cứu bằng Tên (key) để ra Số điện thoại (value). Tên không được trùng, nhưng số điện thoại có thể trùng.

Collections Hierarchy

Java Collections Framework được tổ chức theo cấu trúc phân cấp rõ ràng, giúp bạn dễ dàng chọn đúng công cụ cho từng tình huống.



💡 **Ghi nhớ:** Map không phải là Collection, nhưng vẫn là một phần quan trọng của Collections Framework.
Map lưu trữ cặp key-value thay vì các phần tử đơn lẻ.



2. Item B

3. Item B

4. Item C

6.1.2. List Interface

Đặc điểm của List

List là một trong những cấu trúc dữ liệu được sử dụng nhiều nhất trong Java. Hiểu rõ đặc điểm của List sẽ giúp bạn sử dụng nó hiệu quả.

Có thứ tự (Ordered)

Các phần tử được lưu trữ theo thứ tự chèn vào. Phần tử đầu tiên thêm vào sẽ ở vị trí đầu tiên.

Cho phép Duplicate

Có thể chứa các phần tử giống nhau. Ví dụ: ["Apple", "Apple", "Banana"] là hoàn toàn hợp lệ.

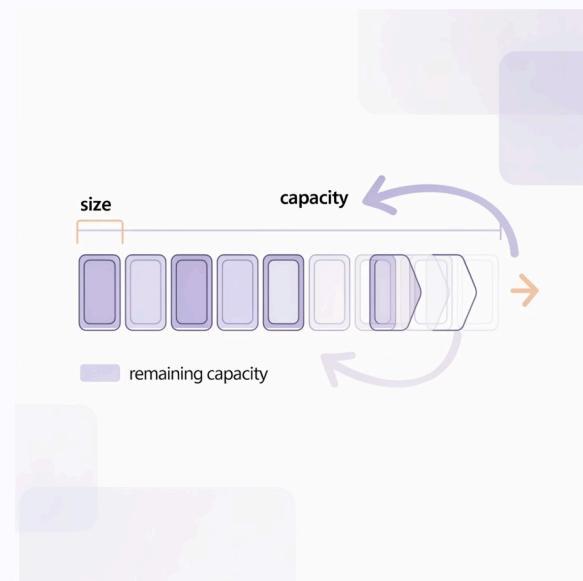
Có Index

Truy cập phần tử theo vị trí (0, 1, 2, ...). Giống như mảng, nhưng linh hoạt hơn nhiều.

ArrayList - Implementation phổ biến nhất

Đặc điểm nổi bật

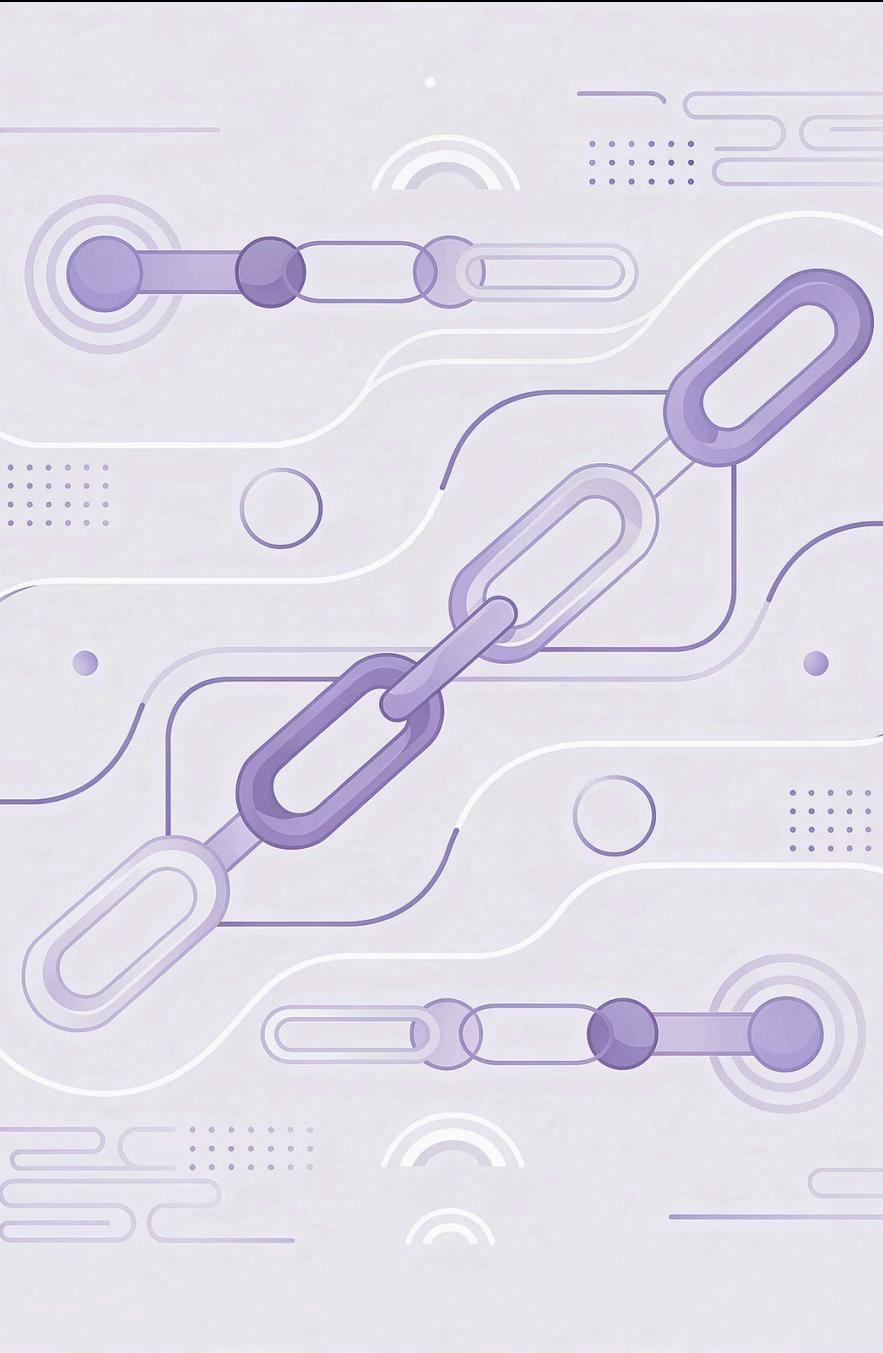
- Implement bằng mảng động
- ⚡ Truy cập cực nhanh O(1)
- 🍎 Thêm/xóa chậm ở giữa O(n)
- Tự động mở rộng khi cần



Ví dụ sử dụng

```
List<String> fruits = new ArrayList<>();  
  
// Thêm phần tử  
fruits.add("Apple");  
fruits.add("Banana");  
fruits.add("Orange");  
  
// Truy cập theo index  
String first = fruits.get(0); // "Apple"  
  
// Duyệt List - Cách 1  
for (String fruit : fruits) {  
    System.out.println(fruit);  
}  
  
// Duyệt List - Cách 2 (Modern)  
fruits.forEach(fruit ->  
    System.out.println(fruit)  
);  
  
// Kích thước  
System.out.println(fruits.size()); // 3
```

⚠💡 Tip: ArrayList là lựa chọn mặc định trong hầu hết trường hợp. Chỉ cân nhắc LinkedList khi bạn thường xuyên thêm/xóa phần tử ở đầu hoặc giữa danh sách.



LinkedList - Khi nào nên dùng?

Đặc điểm kỹ thuật

- Implement bằng doubly linked list
- ⚡ Thêm/xóa nhanh O(1)
- 🐛 Truy cập chậm O(n)
- Tốn bộ nhớ hơn (lưu pointer)

Ví dụ thực tế

```
List<String> tasks = new LinkedList<>();
```

```
tasks.add("Task 1");  
tasks.add("Task 2");
```

```
// Thêm vào đầu - Rất nhanh!  
tasks.addFirst("Urgent Task");
```

```
// Thêm vào cuối  
tasks.addLast("Low Priority");
```

```
// Xóa đầu/cuối - Rất nhanh!  
tasks.removeFirst();  
tasks.removeLast();
```

LinkedList đặc biệt hữu ích khi implement Queue hoặc Deque (hàng đợi hai đầu), nơi bạn thường xuyên thao tác ở đầu và cuối danh sách.

So sánh ArrayList vs LinkedList

Lựa chọn đúng cấu trúc dữ liệu có thể cải thiện hiệu suất ứng dụng đáng kể. Hãy xem bảng so sánh chi tiết:

Đặc điểm	ArrayList	LinkedList	Giải thích
Cấu trúc	Mảng động	Doubly Linked List	Cách tổ chức dữ liệu bên trong
Truy cập (get)	⚡ O(1) - Nhanh	👉 O(n) - Chậm	ArrayList truy cập trực tiếp, LinkedList phải duyệt
Thêm cuối (add)	⚡ O(1) - Nhanh	⚡ O(1) - Nhanh	Cả hai đều nhanh khi thêm cuối
Thêm/xóa giữa	👉 O(n) - Chậm	⚡ O(1) - Nhanh	ArrayList phải dịch chuyển mảng, LinkedList chỉ đổi pointer
Bộ nhớ	Ít hơn	Nhiều hơn	LinkedList tốn thêm bộ nhớ cho pointers
Khi nào dùng	Truy cập nhiều	Thêm/xóa nhiều	Tùy thuộc nhu cầu ứng dụng



Khuyến nghị

Dùng **ArrayList** trong hầu hết trường hợp vì đơn giản và hiệu quả cho các thao tác thông thường.



Khi nào dùng LinkedList

Chỉ dùng **LinkedList** khi bạn thường xuyên thêm/xóa phần tử ở giữa danh sách hoặc implement Queue/Deque.

6.1.3. Set Interface

Đặc điểm của Set

Set là cấu trúc dữ liệu đảm bảo tính duy nhất của các phần tử. Nó rất hữu ích khi bạn cần loại bỏ duplicates hoặc kiểm tra sự tồn tại nhanh chóng.

Không có thứ tự

Set không đảm bảo thứ tự các phần tử (trừ LinkedHashSet và TreeSet). Bạn không thể truy cập theo index.

Không cho phép duplicate

Mỗi phần tử chỉ xuất hiện một lần. Nếu thêm phần tử trùng, nó sẽ bị bỏ qua tự động.

Không có index

Không thể dùng `get(index)` như List. Phải duyệt hoặc dùng `contains()` để kiểm tra.

- 💡 **Khi nào dùng Set:** Khi bạn cần đảm bảo không có phần tử trùng lặp (ví dụ: danh sách ID, email, username) hoặc cần kiểm tra sự tồn tại rất nhanh.

HashSet - Nhanh và hiệu quả

Đặc điểm

- Không có thứ tự
- ⚡ Cực nhanh O(1) cho add/contains
- Sử dụng hashCode() và equals()
- Cho phép một giá trị null
- Hiệu quả nhất trong các loại Set



Ví dụ thực tế

```
Set<String> uniqueEmails = new HashSet<>();  
  
// Thêm email  
uniqueEmails.add("user1@email.com");  
uniqueEmails.add("user2@email.com");  
uniqueEmails.add("user1@email.com");  
// ↑ Bị bỏ qua (duplicate)  
  
System.out.println(uniqueEmails.size());  
// Output: 2  
  
// Kiểm tra tồn tại - Cực nhanh!  
if (uniqueEmails.contains("user1@email.com")) {  
    System.out.println("Email đã tồn tại!");  
}  
  
// Xóa phần tử  
uniqueEmails.remove("user2@email.com");
```

Use case phổ biến: Lưu trữ username, email, ID duy nhất; loại bỏ duplicates từ List; kiểm tra membership nhanh.

TreeSet - Set có thứ tự

Đặc điểm đặc biệt

- ✨ Có thứ tự (sorted tự động)
- 🐢 Chậm hơn HashSet - $O(\log n)$
- Dùng Comparable hoặc Comparator
- Không cho phép null
- Implement bằng Red-Black Tree

Ví dụ sắp xếp tự động

```
Set<String> sortedNames = new TreeSet<>();  
  
sortedNames.add("Charlie");  
sortedNames.add("Alice");  
sortedNames.add("Bob");  
  
// Tự động sắp xếp theo alphabet  
for (String name : sortedNames) {  
    System.out.println(name);  
}  
// Output:  
// Alice  
// Bob  
// Charlie  
  
// Ví dụ với số  
Set<Integer> numbers = new TreeSet<>();  
numbers.add(5);  
numbers.add(1);  
numbers.add(3);  
// Tự động sắp xếp: [1, 3, 5]
```

💡 **Khi nào dùng TreeSet:** Khi bạn cần Set có thứ tự (sorted) mà không cần sắp xếp thủ công. Ví dụ: leaderboard, danh sách ưu tiên.

LinkedHashSet - Kết hợp tốt nhất

LinkedHashSet là sự kết hợp hoàn hảo giữa HashSet và LinkedList, mang lại hiệu suất cao và duy trì thứ tự chèn.



Đặc điểm

- Giữ thứ tự chèn (insertion order)
- Nhanh như HashSet - O(1)
- Kết hợp HashSet + LinkedList
- Tốn bộ nhớ hơn HashSet một chút

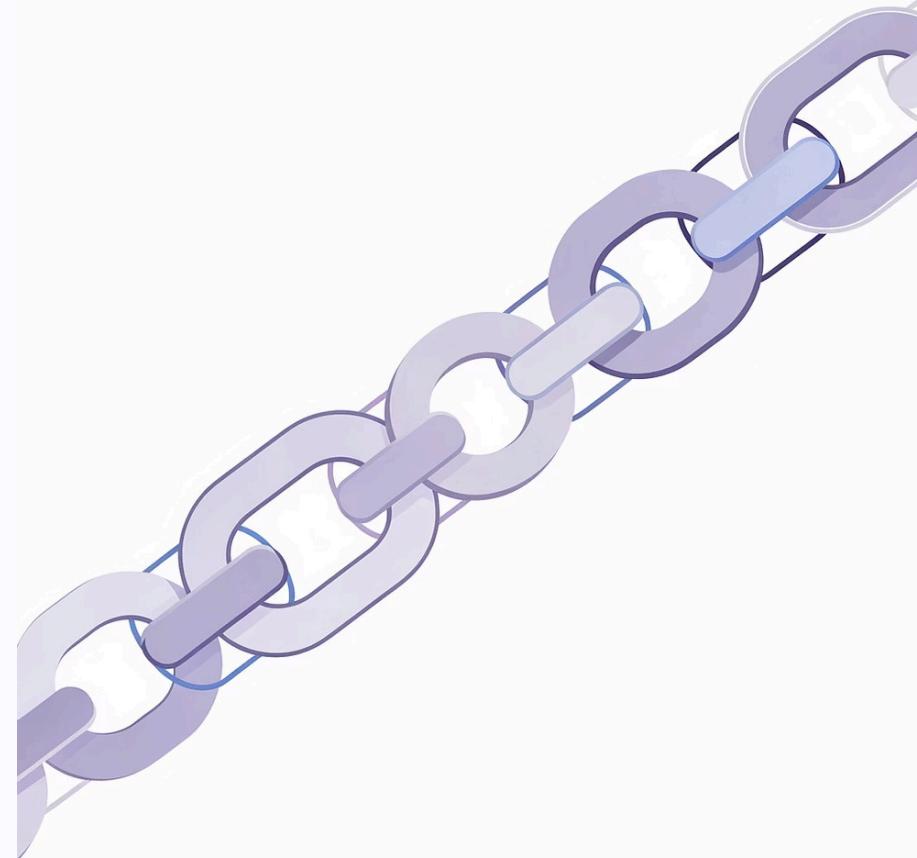
Ví dụ

```
Set<String> tasks = new LinkedHashSet<>();

tasks.add("Viết báo cáo");
tasks.add("Hợp team");
tasks.add("Code review");

// Giữ đúng thứ tự chèn
for (String task : tasks) {
    System.out.println(task);
}

// Output:
// Viết báo cáo
// Hợp team
// Code review
```



6.1.4. Map Interface

Đặc điểm của Map

Map là cấu trúc dữ liệu lưu trữ theo cặp key-value, giống như một cuốn từ điển hoặc danh bạ điện thoại trong thực tế.



Key-Value Pairs

Lưu trữ dữ liệu dưới dạng cặp khóa-giá trị. Mỗi key ánh xạ đến một value cụ thể.



Key không Duplicate

Mỗi key chỉ xuất hiện một lần. Nếu thêm key trùng, value cũ sẽ bị ghi đè.



Value có thể Duplicate

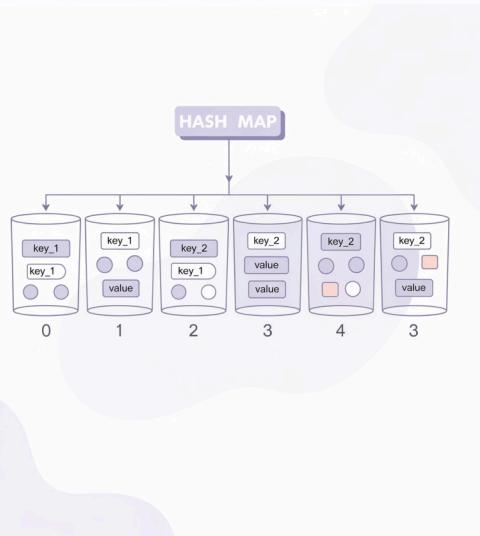
Nhiều key khác nhau có thể trỏ đến cùng một value. Value không bị ràng buộc duy nhất.

- ❑💡 **Tưởng tượng:** Map giống như danh bạ điện thoại - Tên (key) là duy nhất, nhưng nhiều người có thể có cùng số điện thoại (value). Bạn tra cứu bằng Tên để lấy Số điện thoại.

HashMap - Map phổ biến nhất

Đặc điểm

- Không có thứ tự
- ⚡ Cực nhanh O(1) cho get/put
- Cho phép một null key và nhiều null values
- Không thread-safe
- Hiệu quả nhất trong các loại Map



Ví dụ thực tế

```
Map<String, Integer> productPrices =  
    new HashMap<>();  
  
// Thêm cặp key-value  
productPrices.put("Apple", 10000);  
productPrices.put("Banana", 20000);  
productPrices.put("Orange", 15000);  
  
// Truy cập theo key - Cực nhanh!  
int applePrice = productPrices.get("Apple");  
// 10000  
  
// Kiểm tra key tồn tại  
if (productPrices.containsKey("Banana")) {  
    System.out.println("Có bán Chuối");  
}  
  
// Cập nhật value  
productPrices.put("Apple", 12000);  
// Ghi đè giá cũ
```

Duyệt HashMap

```
// Duyệt cả key và value  
for (Map.Entry<String, Integer> entry :  
    productPrices.entrySet()) {  
    System.out.println(entry.getKey() +  
        ":" + entry.getValue() + "đ");  
}  
  
// Chỉ duyệt keys  
for (String product : productPrices.keySet()) {  
    System.out.println(product);  
}  
  
// Chỉ duyệt values  
for (Integer price : productPrices.values()) {  
    System.out.println(price + "đ");  
}
```

TreeMap - Map có thứ tự

TreeMap tự động sắp xếp các entry theo key, rất hữu ích khi bạn cần dữ liệu có thứ tự.

Đặc điểm đặc biệt

- ⭐ Có thứ tự (sorted by key)
- 🐢 Chậm hơn HashMap - O(log n)
- Key phải implement Comparable
- Không cho phép null key
- Dùng Red-Black Tree

Ví dụ sắp xếp tự động

```
Map<String, Integer> scores =  
    new TreeMap<>();  
  
scores.put("Charlie", 85);  
scores.put("Alice", 95);  
scores.put("Bob", 90);  
  
// Tự động sắp xếp theo key (tên)  
for (String name : scores.keySet()) {  
    System.out.println(name +  
        ": " + scores.get(name));  
}  
// Output (sorted):  
// Alice: 95  
// Bob: 90  
// Charlie: 85
```

Use case: Leaderboard, từ điển (dictionary), bất kỳ dữ liệu nào cần hiển thị theo thứ tự key.

6.1.5. Tiêu chí Lựa chọn Cấu trúc Dữ liệu

Decision Tree - Cây quyết định

```
Cần duplicate?  
|   — Có → List  
|   |   — Truy cập nhiều? → ArrayList  
|   |   — Thêm/xóa giữa nhiều? → LinkedList  
|  
|   — Không → Set  
|   |   — Cần thứ tự?  
|   |   |   — Có → TreeSet (sorted)  
|   |   |   — LinkedHashSet (insertion order)  
|   |   — Không → HashSet  
|  
|   — Cần key-value? → Map  
|   |   — Cần thứ tự? → TreeMap  
|   |   — Không → HashMap
```

Phân tích Hiệu Năng (Performance Analysis)

Chọn đúng "Hộp dụng cụ" quyết định tốc độ chương trình của bạn:

Hành động	ArrayList	LinkedList	HashSet	HashMap
Truy cập (Get)	⚡ O(1)	🐢 O(n)	-	⚡ O(1)
Thêm/Xóa cuối	⚡ O(1)	⚡ O(1)	⚡ O(1)	⚡ O(1)
Thêm/Xóa giữa	🐢 O(n)	⚡ O(1)	-	-
Tìm kiếm	🐢 O(n)	🐢 O(n)	⚡ O(1)	⚡ O(1)

Kinh nghiệm 1

Nếu cần **tìm kiếm nhanh** (ví dụ: tìm User theo ID) → Đừng dùng List, hãy dùng Map hoặc Set.

Kinh nghiệm 2

Nếu cần **lưu trữ đơn giản** để duyệt → Dùng ArrayList, đơn giản và hiệu quả nhất.

Kinh nghiệm 3

Hạn chế dùng LinkedList trừ khi bạn hiểu rõ mình đang làm gì (thường ít khi thực sự cần).

Ví dụ thực tế - Chọn đúng Collection

Hãy xem các tình huống thực tế và lựa chọn Collection phù hợp:

1. Danh sách sinh viên

Có thể có nhiều sinh viên cùng tên, cần giữ thứ tự.

```
List<Student> students =  
    new ArrayList<>();
```

Dùng **ArrayList** vì cần duplicate và truy cập theo index.

2. Danh sách ID duy nhất

Mỗi ID chỉ xuất hiện một lần, không quan tâm thứ tự.

```
Set<String> uniquelds =  
    new HashSet<>();
```

Dùng **HashSet** vì cần đảm bảo unique và kiểm tra nhanh.

3. Dictionary (Từ điển)

Tra cứu từ (key) để lấy nghĩa (value).

```
Map<String, String>  
dictionary =  
    new HashMap<>();
```

Dùng **HashMap** vì cần tra cứu nhanh theo key.

4. Bảng xếp hạng

Cần lưu điểm số và tự động sắp xếp theo tên.

```
Map<String, Integer> scores  
=  
    new TreeMap<>();
```

Dùng **TreeMap** vì cần Map có thứ tự (sorted).

6.2. GENERICS

6.2.1. Generics là gì?

Vấn đề khi không dùng Generics

Trước khi có Generics (Java 5), code Java rất dễ gây lỗi runtime vì không có kiểm tra kiểu dữ liệu lúc compile.

Code cũ (không type-safe)

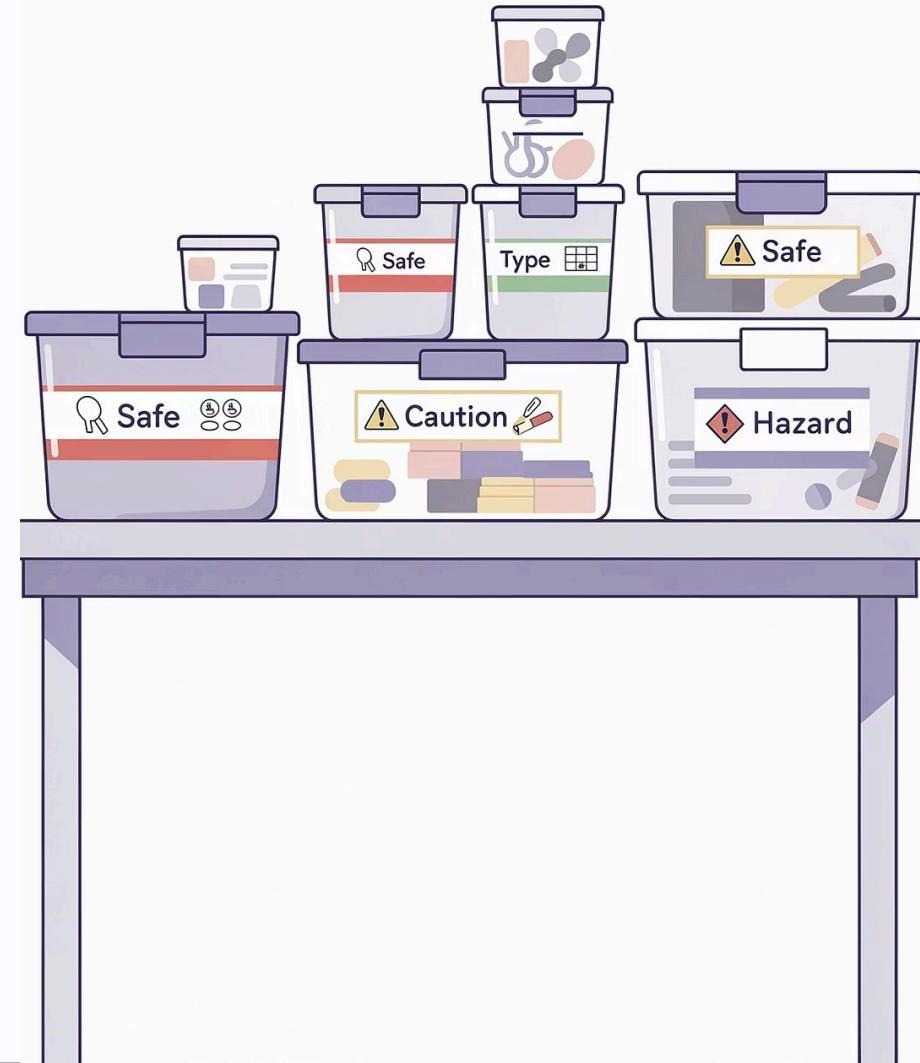
```
List list = new ArrayList();
list.add("Hello");
list.add(123);
list.add(new Date());

// Phải cast khi lấy ra
String str = (String) list.get(0);
//
```

Giải pháp: Generics (Type-Safe)

Code mới (type-safe) 

```
List<String> list = new ArrayList<>();  
  
list.add("Hello");  
list.add("World");  
  
// list.add(123);  
//
```



6.2.2. Cú pháp Generics - Generic Class

Generics cho phép bạn tạo các class linh hoạt, có thể làm việc với nhiều kiểu dữ liệu khác nhau mà vẫn đảm bảo type-safe.

Định nghĩa Generic Class

```
// Định nghĩa một Box generic
public class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}
```

Sử dụng với String

```
// T được thay thế bằng String
Box<String> stringBox = new Box<>();

stringBox.setItem("Hello");

// Không cần cast!
String value = stringBox.getItem();
//
```

Generic Method

Ngoài Generic Class, bạn cũng có thể tạo Generic Method - một method có thể làm việc với nhiều kiểu dữ liệu.

Ví dụ 1: Generic Method đơn giản

```
public class Utils {  
    // Generic method để in mảng  
    public static <T> void printArray(T[] array) {  
        for (T item : array) {  
            System.out.println(item);  
        }  
    }  
  
    // Sử dụng  
    String[] strings = {"Hello", "World", "Java"};  
    Utils.printArray(strings); //
```

Bounded Type Parameters

Đôi khi bạn muốn giới hạn kiểu dữ liệu mà Generic có thể chấp nhận. Đó là lúc Bounded Type Parameters phát huy tác dụng.

Giới hạn kiểu generic với extends

Định nghĩa với Bound

```
// Chỉ chấp nhận Number và subclass
public class NumberBox<T extends Number> {
    private T number;

    public void setNumber(T number) {
        this.number = number;
    }

    // Có thể gọi methods của Number
    public double getDoubleValue() {
        return number.doubleValue();
    }

    public int getIntValue() {
        return number.intValue();
    }
}
```

Sử dụng

```
//
```

6.2.3. Wildcards - Ký tự đại diện ? (Unbounded Wildcard)

Wildcard ? cho phép chấp nhận bất kỳ kiểu dữ liệu nào. Rất hữu ích khi bạn chỉ cần đọc dữ liệu, không cần biết kiểu cụ thể.

Ví dụ với Unbounded Wildcard

```
// Chấp nhận List của bất kỳ type nào
public void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}

// Sử dụng
List<String> strings =
    Arrays.asList("A", "B", "C");
printList(strings); //
```

? extends Type (Upper Bounded Wildcard)

? extends Type chấp nhận Type và tất cả các subclass của nó. Được dùng khi bạn muốn **đọc** dữ liệu từ collection.

Ví dụ thực tế

```
// Chấp nhận Number và subclass
// (Integer, Double, Float, etc.)
public double sum(List<? extends Number> numbers) {
    double sum = 0;
    for (Number num : numbers) {
        // Có thể gọi methods của Number
        sum += num.doubleValue();
    }
    return sum;
}

// Sử dụng
List<Integer> integers =
    Arrays.asList(1, 2, 3);
double sumInt = sum(integers);
//
```

? super Type (Lower Bounded Wildcard)

? super Type chấp nhận Type và tất cả các superclass của nó. Được dùng khi bạn muốn **ghi** dữ liệu vào collection.

Ví dụ thực tế

```
// Chấp nhận Integer và superclass  
// (Number, Object)  
public void addNumbers(  
    List<? super Integer> list  
) {  
    list.add(1);  
    list.add(2);  
    list.add(3);  
    //
```

6.3. LAMBDA EXPRESSIONS & FUNCTIONAL INTERFACES

6.3.1. Lambda Expressions là gì?

Vấn đề: Code dài dòng với Anonymous Inner Class

Trước Java 8, khi cần truyền một hành động đơn giản, chúng ta phải viết rất nhiều boilerplate code:

Code cũ (dài dòng) ❌

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

// Sắp xếp theo độ dài
Collections.sort(names,
    new Comparator<String>() {

        @Override
        public int compare(String a, String b) {
            return a.length() - b.length();
        }
});
```

Code mới (Lambda) ✓

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

// Lambda expression
Collections.sort(names,
    (a, b) -> a.length() - b.length()
);
```

Lợi ích:

- ✓ 1 dòng code!
- ✓ Ngắn gọn, rõ ràng
- ✓ Dễ đọc, dễ maintain
- ✓ Functional programming style

Vấn đề:

- ❌ 8 dòng code cho logic đơn giản
- ❌ Nhiều boilerplate
- ❌ Khó đọc, khó maintain

Giải pháp: Lambda Expression

Lambda Expression là một cách ngắn gọn để viết anonymous function (hàm ẩn danh). Nó giúp code ngắn hơn, dễ đọc hơn và có phong cách functional programming hiện đại.

So sánh Before & After

Before Lambda (Java 7)

```
// 1. Anonymous Inner Class - 7 dòng
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello");
    }
};

// 2. Comparator - 8 dòng
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
});
```

After Lambda (Java 8+)

```
// 1. Lambda - 1 dòng!
Runnable r = () -> System.out.println("Hello");

// 2. Lambda - 1 dòng!
Collections.sort(names,
    (a, b) -> a.compareTo(b)
);
```

- ☐💡 **Kết luận:** Lambda giảm code từ **7-8 dòng xuống 1 dòng**, giữ nguyên chức năng nhưng dễ đọc và maintain hơn nhiều!

6.3.2. Cú pháp Lambda

Cú pháp cơ bản

```
(parameters) -> expression
```

```
// Hoặc với nhiều statements
```

```
(parameters) -> {  
    statements;  
    return value;  
}
```

Các dạng Lambda Expression

01

Không tham số

```
// Không có parameter, dùng ()  
Runnable r = () -> System.out.println("Hello");  
r.run(); // Output: Hello
```

02

Một tham số (bỏ dấu ngoặc)

```
// Một parameter, bỏ được ()  
Function<String, Integer> length = s -> s.length();  
int len = length.apply("Hello"); // 5
```

03

Nhiều tham số

```
// Nhiều parameters, cần ()  
BiFunction<Integer, Integer, Integer> add =  
    (a, b) -> a + b;  
int sum = add.apply(5, 3); // 8
```

04

Nhiều dòng code (block)

```
// Nhiều statements, cần {} và return  
Function<String, String> process = s -> {  
    String upper = s.toUpperCase();  
    String result = "Processed: " + upper;  
    return result;  
};
```

Ví dụ thực tế với Lambda

Hãy xem Lambda Expression được áp dụng như thế nào trong các tình huống thực tế:

1. Duyệt List với forEach

Cách cũ (for loop)

```
List<String> names = Arrays.asList(  
    "Alice", "Bob", "Charlie"  
>;  
  
for (String name : names) {  
    System.out.println(name);  
}
```

Cách mới (Lambda)

```
List<String> names = Arrays.asList(  
    "Alice", "Bob", "Charlie"  
>;  
  
// Lambda expression  
names.forEach(name ->  
    System.out.println(name)  
>;  
  
// Hoặc method reference  
names.forEach(System.out::println);
```

2. Sắp xếp với Comparator

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");  
  
// Sắp xếp theo độ dài (ngắn đến dài)  
names.sort((a, b) -> a.length() - b.length());  
// Output: [Bob, Alice, David, Charlie]  
  
// Sắp xếp theo alphabet (A-Z)  
names.sort((a, b) -> a.compareTo(b));  
// Output: [Alice, Bob, Charlie, David]  
  
// Sắp xếp ngược (Z-A)  
names.sort((a, b) -> b.compareTo(a));  
// Output: [David, Charlie, Bob, Alice]
```

3. Lọc với removeIf

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);  
  
// Xóa tất cả số chẵn  
numbers.removeIf(n -> n % 2 == 0);  
// Output: [1, 3, 5, 7]
```

6.3.3. Functional Interfaces

Định nghĩa

Functional Interface là interface chỉ có **đúng một abstract method**. Đây là nền tảng để Lambda Expression hoạt động.

Cú pháp

```
@FunctionalInterface  
public interface MyFunction {  
    // Chỉ có 1 abstract method  
    void doSomething();  
  
    // Có thể có default methods  
    default void doMore() {  
        System.out.println("More");  
    }  
  
    // Có thể có static methods  
    static void help() {  
        System.out.println("Help");  
    }  
}
```

Sử dụng với Lambda

```
// Cách cũ (Anonymous class)  
MyFunction func1 = new MyFunction() {  
    @Override  
    public void doSomething() {  
        System.out.println("Done");  
    }  
};  
  
// Cách mới (Lambda)  
MyFunction func2 = () ->  
    System.out.println("Done");  
  
func2.doSomething(); // Done
```

💡 **Annotation @FunctionalInterface:** Không bắt buộc nhưng nên dùng. Nó giúp compiler kiểm tra xem interface có đúng 1 abstract method hay không.

Các Functional Interfaces phổ biến trong Java

Java cung cấp sẵn nhiều Functional Interfaces trong package `java.util.function`. Bạn không cần tự tạo, chỉ cần biết cách dùng!

1. Predicate<T>

Chức năng: Kiểm tra điều kiện, trả về boolean

```
Predicate<String> isLong =  
    s -> s.length() > 5;  
  
boolean result = isLong.test("Hello World");  
// true  
boolean result2 = isLong.test("Hi");  
// false
```

2. Consumer<T>

Chức năng: Nhận input, không trả về gì

```
Consumer<String> print =  
    s -> System.out.println(s);  
  
print.accept("Hello");  
// In "Hello"  
print.accept("World");  
// In "World"
```

3. Function<T, R>

Chức năng: Nhận T, trả về R

```
Function<String, Integer> length =  
    s -> s.length();  
  
int len = length.apply("Hello");  
// 5  
int len2 = length.apply("Java");  
// 4
```

4. Supplier<T>

Chức năng: Không nhận gì, trả về T

```
Supplier<String> greeting =  
    () -> "Hello World";  
  
String msg = greeting.get();  
// "Hello World"
```

5. BiFunction<T, U, R>

Chức năng: Nhận T và U, trả về R

```
BiFunction<Integer, Integer, Integer>  
    add = (a, b) -> a + b;  
  
int sum = add.apply(5, 3);  
// 8  
int sum2 = add.apply(10, 20);  
// 30
```

6. UnaryOperator<T>

Chức năng: Nhận T, trả về T

```
UnaryOperator<Integer> square =  
    n -> n * n;  
  
int result = square.apply(5);  
// 25
```

Ví dụ sử dụng Functional Interfaces

Hãy xem cách kết hợp các Functional Interfaces trong thực tế:

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

// 1. Predicate: Lọc tên dài hơn 4 ký tự
Predicate<String> longName = name -> name.length() > 4;

List<String> longNames = names.stream()
    .filter(longName) // Dùng Predicate
    .collect(Collectors.toList());
// Result: ["Alice", "Charlie", "David"]

// 2. Function: Chuyển thành chữ hoa
Function<String, String> toUpper = String::toUpperCase;

List<String> upperNames = names.stream()
    .map(toUpper) // Dùng Function
    .collect(Collectors.toList());
// Result: ["ALICE", "BOB", "CHARLIE", "DAVID", "EVE"]

// 3. Consumer: In từng phần tử
Consumer<String> print = System.out::println;

names.forEach(print); // Dùng Consumer
// Output:
// Alice
// Bob
// Charlie
// David
// Eve
```

Kết hợp nhiều Functional Interfaces

```
// Chain nhiều operations
names.stream()
    .filter(name -> name.length() > 3) // Predicate
    .map(name -> name.toUpperCase()) // Function
    .forEach(name -> System.out.println(name)); // Consumer

// Output:
// ALICE
// CHARLIE
// DAVID
```

6.4. STREAM API (TƯ DUY DECLARATIVE)

6.4.1. Imperative vs Declarative

Imperative (Mệnh lệnh) - "HOW" (Làm thế nào)

Imperative programming tập trung vào **cách thức** thực hiện từng bước một. Bạn phải mô tả chi tiết "máy tính phải làm gì ở mỗi bước".

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

List<String> result = new ArrayList<>();

// Imperative: Mô tả từng bước
for (String name : names) {
    if (name.length() > 4) {    // Bước 1: Lọc
        String upper = name.toUpperCase(); // Bước 2: Chuyển đổi
        result.add(upper);      // Bước 3: Thêm vào list
    }
}

System.out.println(result);
// [ALICE, CHARLIE, DAVID]
```

Đặc điểm

- ✗ Code dài (8 dòng)
- ✗ Phải quản lý state (biến result)
- ✗ Khó đọc, khó hiểu
- ✗ Nhiều chi tiết thừa
- ✗ Dễ lỗi (quên thêm vào list)

Declarative (Khai báo) - "WHAT" (Làm gì)

Declarative programming tập trung vào **kết quả** mong muốn. Bạn chỉ cần nói "tôi muốn gì", không cần nói "làm thế nào".

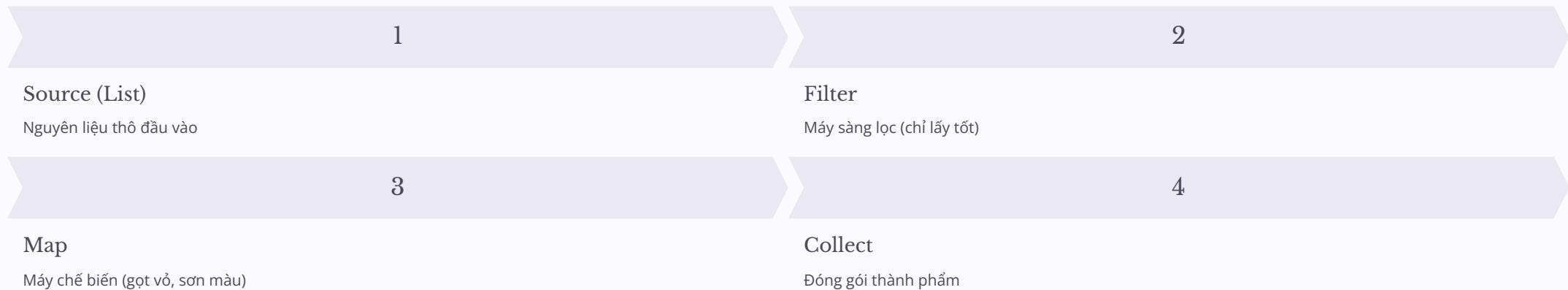
```
List<String> names = Arrays.asList(  
    "Alice", "Bob", "Charlie", "David"  
);  
  
// Declarative: Mô tả kết quả mong muốn  
List<String> result = names.stream()  
.filter(name -> name.length() > 4) // Lọc  
.map(String::toUpperCase)      // Chuyển đổi  
.collect(Collectors.toList());   // Thu thập  
  
System.out.println(result);  
// [ALICE, CHARLIE, DAVID]
```

Đặc điểm

- ✅ Code ngắn (3 dòng)
- ✅ Không cần quản lý state
- ✅ Dễ đọc như câu văn
- ✅ Logic rõ ràng
- ✅ Ít lỗi hơn

🏭 Tưởng tượng: Dây chuyên sản xuất (Assembly Line)

Stream API biến việc xử lý dữ liệu thành một dây chuyền nhà máy tự động:



💡 **Tại sao hay hơn Loop:** Bạn không cần tạo biến tạm (result), không cần viết if/else lồng nhau. Logic cực kỳ rõ ràng "từ trái sang phải". Đọc code giống như đọc câu văn tiếng Việt!

6.4.2. Stream API là gì?

Định nghĩa

Stream API là API để xử lý collections theo phong cách functional programming. Nó không phải là một cấu trúc dữ liệu, mà là một công cụ để xử lý dữ liệu.

Không lưu trữ dữ liệu

Stream không phải là cấu trúc dữ liệu. Nó chỉ là một "pipeline" để xử lý dữ liệu từ source (List, Array, ...).

Functional & Immutable

Stream không thay đổi source gốc. Mỗi operation tạo ra một stream mới, giữ nguyên dữ liệu ban đầu.

Lazy Evaluation

Các intermediate operations chỉ thực thi khi có terminal operation. Tối ưu hiệu năng!

Có thể Parallel

Dễ dàng chạy song song với .parallelStream() để tận dụng đa nhân CPU, tăng tốc xử lý dữ liệu lớn.

Tạo Stream

Có nhiều cách để tạo Stream từ các nguồn dữ liệu khác nhau:

01

Từ Collection

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();

// Hoặc parallel stream
Stream<String> parallelStream = list.parallelStream();
```

02

Từ Array

```
String[] array = {"a", "b", "c"};
Stream<String> stream = Arrays.stream(array);

// Hoặc với index
Stream<String> stream2 = Arrays.stream(array, 0, 2);
// ["a", "b"]
```

03

Từ values

```
// Tạo stream trực tiếp từ values
Stream<String> stream = Stream.of("a", "b", "c");

// Empty stream
Stream<String> empty = Stream.empty();
```

04

Từ range (số)

```
// Tạo dãy số từ 1 đến 9 (không bao gồm 10)
IntStream range = IntStream.range(1, 10);

// Tạo dãy số từ 1 đến 10 (bao gồm 10)
IntStream rangeClosed = IntStream.rangeClosed(1, 10);
```

 **Lưu ý:** Một Stream chỉ có thể dùng một lần. Sau khi có terminal operation, stream sẽ bị "tiêu thụ" (consumed) và không thể dùng lại. Nếu cần, tạo stream mới!

6.4.3. Các Toán tử Stream

1. filter() - Lọc phần tử

filter() giữ lại các phần tử thỏa mãn điều kiện, loại bỏ phần tử không thỏa mãn.

Cú pháp

```
Stream<T> filter(Predicate<T> predicate)
```

Tham số: Predicate (điều kiện trả về true/false)

Trả về: Stream mới chứa các phần tử thỏa điều kiện

Ví dụ

```
List<Integer> numbers =  
    Arrays.asList(1, 2, 3, 4, 5, 6);  
  
// Lọc số chẵn  
List<Integer> evens = numbers.stream()  
    .filter(n -> n % 2 == 0)  
    .collect(Collectors.toList());  
  
System.out.println(evens);  
// [2, 4, 6]
```

Ví dụ thực tế: Lọc sinh viên giỏi

```
List<Student> students = getStudents();  
  
// Lọc sinh viên có GPA >= 3.5  
List<Student> excellentStudents = students.stream()  
    .filter(s -> s.getGpa() >= 3.5)  
    .collect(Collectors.toList());  
  
// Lọc sinh viên tuổi từ 18-22  
List<Student> youngStudents = students.stream()  
    .filter(s -> s.getAge() >= 18 && s.getAge() <= 22)  
    .collect(Collectors.toList());
```

2. map() - Chuyển đổi phần tử

map() chuyển đổi mỗi phần tử thành một phần tử khác (có thể cùng hoặc khác kiểu).

Ví dụ 1: Chuyển thành chữ hoa

```
List<String> names = Arrays.asList(  
    "alice", "bob", "charlie"  
)  
  
// Chuyển thành chữ hoa  
List<String> upper = names.stream()  
    .map(String::toUpperCase)  
    .collect(Collectors.toList());  
  
System.out.println(upper);  
// [ALICE, BOB, CHARLIE]
```

Ví dụ 2: Đổi kiểu dữ liệu

```
List<String> names = Arrays.asList(  
    "alice", "bob", "charlie"  
)  
  
// String → Integer (độ dài)  
List<Integer> lengths = names.stream()  
    .map(String::length)  
    .collect(Collectors.toList());  
  
System.out.println(lengths);  
// [5, 3, 7]
```

Ví dụ thực tế: Chuyển đổi objects

```
List<Student> students = getStudents();  
  
// Lấy danh sách tên  
List<String> names = students.stream()  
    .map(Student::getName) // Student → String  
    .collect(Collectors.toList());  
  
// Lấy danh sách điểm  
List<Double> gpas = students.stream()  
    .map(Student::getGpa) // Student → Double  
    .collect(Collectors.toList());  
  
// Tạo DTO mới  
List<StudentDTO> dtos = students.stream()  
    .map(s -> new StudentDTO(s.getName(), s.getGpa()))  
    .collect(Collectors.toList());
```

3. sorted() - Sắp xếp phần tử

sorted() sắp xếp các phần tử trong stream theo thứ tự tự nhiên hoặc theo Comparator.

Sắp xếp tự nhiên

```
List<String> names = Arrays.asList(  
    "Charlie", "Alice", "Bob"  
>;  
  
// Sắp xếp alphabet (A-Z)  
List<String> sorted = names.stream()  
    .sorted()  
    .collect(Collectors.toList());  
  
System.out.println(sorted);  
// [Alice, Bob, Charlie]  
  
// Sắp xếp số  
List<Integer> numbers =  
    Arrays.asList(5, 2, 8, 1, 9);  
List<Integer> sortedNums =  
    numbers.stream()  
    .sorted()  
    .collect(Collectors.toList());  
// [1, 2, 5, 8, 9]
```

Sắp xếp tùy chỉnh

```
List<String> names = Arrays.asList(  
    "Charlie", "Alice", "Bob"  
>;  
  
// Sắp xếp theo độ dài  
List<String> sortedByLength =  
    names.stream()  
    .sorted((a, b) ->  
        a.length() - b.length()  
)  
    .collect(Collectors.toList());  
  
System.out.println(sortedByLength);  
// [Bob, Alice, Charlie]  
  
// Sắp xếp ngược (Z-A)  
List<String> reversed =  
    names.stream()  
    .sorted((a, b) -> b.compareTo(a))  
    .collect(Collectors.toList());  
// [Charlie, Bob, Alice]
```

4. distinct() - Loại bỏ duplicate

distinct() loại bỏ các phần tử trùng lặp, chỉ giữ lại các phần tử duy nhất.

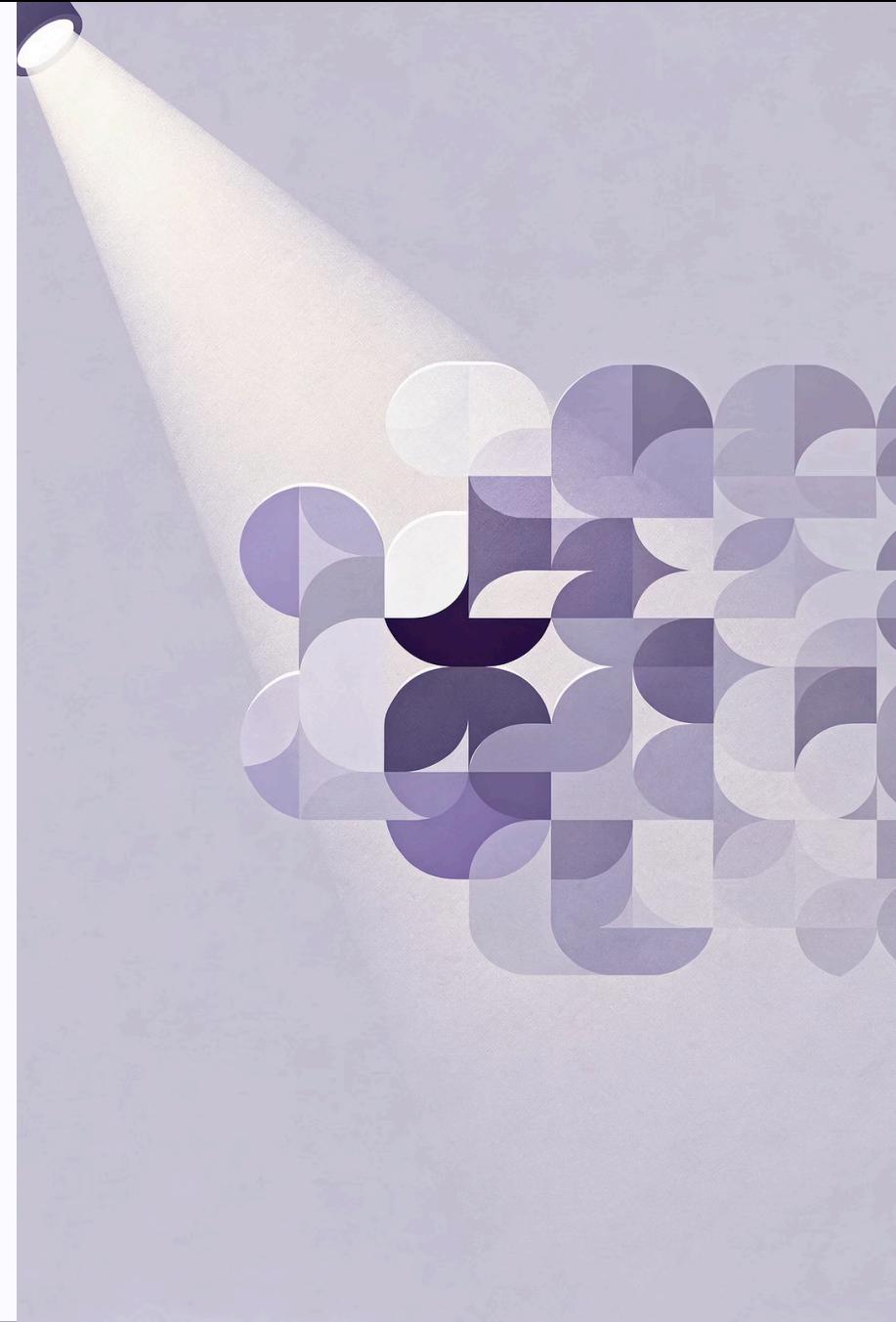
Ví dụ với số

```
List<Integer> numbers = Arrays.asList(  
    1, 2, 2, 3, 3, 3, 4, 5, 5  
);  
  
List<Integer> unique = numbers.stream()  
    .distinct()  
    .collect(Collectors.toList());  
  
System.out.println(unique);  
// [1, 2, 3, 4, 5]
```

Ví dụ với String

```
List<String> words = Arrays.asList(  
    "apple", "banana", "apple",  
    "orange", "banana"  
);  
  
List<String> uniqueWords =  
    words.stream()  
    .distinct()  
    .collect(Collectors.toList());  
  
System.out.println(uniqueWords);  
// [apple, banana, orange]
```

❑ **Lưu ý:** distinct() dùng equals() và hashCode() để so sánh. Đối với custom objects, đảm bảo đã override hai methods này đúng cách!



5. limit() và skip()

limit(n) giới hạn lấy tối đa n phần tử đầu tiên. **skip(n)** bỏ qua n phần tử đầu tiên.

limit() - Lấy n phần tử đầu

```
List<Integer> numbers = Arrays.asList(  
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
);  
  
// Lấy 3 phần tử đầu  
List<Integer> first3 = numbers.stream()  
    .limit(3)  
    .collect(Collectors.toList());  
  
System.out.println(first3);  
// [1, 2, 3]  
  
// Top 5  
List<Integer> top5 = numbers.stream()  
    .limit(5)  
    .collect(Collectors.toList());  
// [1, 2, 3, 4, 5]
```

skip() - Bỏ qua n phần tử đầu

```
List<Integer> numbers = Arrays.asList(  
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
);  
  
// Bỏ qua 3 phần tử đầu  
List<Integer> skip3 = numbers.stream()  
    .skip(3)  
    .collect(Collectors.toList());  
  
System.out.println(skip3);  
// [4, 5, 6, 7, 8, 9, 10]  
  
// Bỏ qua 5 phần tử đầu  
List<Integer> skip5 = numbers.stream()  
    .skip(5)  
    .collect(Collectors.toList());  
// [6, 7, 8, 9, 10]
```

Kết hợp limit() và skip() - Pagination

```
List<Integer> numbers = IntStream.rangeClosed(1, 100)  
    .boxed()  
    .collect(Collectors.toList());  
  
int page = 2; // Trang thứ 2  
int pageSize = 10; // Mỗi trang 10 phần tử  
  
// Lấy dữ liệu trang 2  
List<Integer> pageData = numbers.stream()  
    .skip((page - 1) * pageSize) // Bỏ qua 10 phần tử đầu  
    .limit(pageSize) // Lấy 10 phần tử  
    .collect(Collectors.toList());  
  
System.out.println(pageData);  
// [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

6. collect() - Thu thập kết quả

collect() là terminal operation để thu thập kết quả từ stream vào một cấu trúc dữ liệu cụ thể (List, Set, Map, String, ...).

To List

```
List<String> list = stream  
.collect(Collectors.toList());
```

To Set

```
Set<String> set = stream  
.collect(Collectors.toSet());
```

To Map

```
Map<String, Integer> map = stream  
.collect(Collectors.toMap(  
    name -> name, // Key  
    name -> name.length() // Value  
));
```

Các cách thu thập phổ biến

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

// 1. To List (phổ biến nhất)

```
List<String> list = names.stream()  
.collect(Collectors.toList());
```

// 2. To Set (loại bỏ duplicate)

```
Set<String> set = names.stream()  
.collect(Collectors.toSet());
```

// 3. To Map

```
Map<String, Integer> nameLength = names.stream()  
.collect(Collectors.toMap(  
    name -> name, // Key: tên  
    name -> name.length() // Value: độ dài  
));  
// {"Alice": 5, "Bob": 3, "Charlie": 7}
```

// 4. Joining thành String

```
String joined = names.stream()  
.collect(Collectors.joining(", "));  
// "Alice, Bob, Charlie"
```

// 5. Joining với prefix/suffix

```
String formatted = names.stream()  
.collect(Collectors.joining(", ", "[", "]"));  
// "[Alice, Bob, Charlie]"
```

7. reduce() - Tổng hợp (Aggregation)

reduce() tổng hợp tất cả phần tử trong stream thành một giá trị duy nhất bằng cách áp dụng một hàm kết hợp (accumulator).

Tính tổng

```
List<Integer> numbers = Arrays.asList(  
    1, 2, 3, 4, 5  
);  
  
// Tính tổng - Cách 1  
int sum = numbers.stream()  
    .reduce(0, (a, b) -> a + b);  
System.out.println(sum); // 15
```

```
// Tính tổng - Cách 2 (ngắn gọn)  
int sum2 = numbers.stream()  
    .reduce(0, Integer::sum);  
System.out.println(sum2); // 15
```

Tính tích

```
List<Integer> numbers = Arrays.asList(  
    1, 2, 3, 4, 5  
);  
  
// Tính tích  
int product = numbers.stream()  
    .reduce(1, (a, b) -> a * b);  
System.out.println(product); // 120  
  
// 1 * 1 = 1  
// 1 * 2 = 2  
// 2 * 3 = 6  
// 6 * 4 = 24  
// 24 * 5 = 120
```

Tìm Max/Min

```
List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 9, 3);  
  
// Tìm Max  
Optional<Integer> max = numbers.stream()  
    .reduce(Integer::max);  
max.ifPresent(System.out::println); // 9  
  
// Tìm Min  
Optional<Integer> min = numbers.stream()  
    .reduce(Integer::min);  
min.ifPresent(System.out::println); // 1  
  
// Hoặc dùng trực tiếp  
int maxValue = numbers.stream()  
    .max(Integer::compareTo)  
    .orElse(0); // 9
```

8. forEach() - Duyệt và thực hiện hành động

forEach() là terminal operation để duyệt qua từng phần tử và thực hiện một hành động (Consumer).

In từng phần tử

```
List<String> names = Arrays.asList(  
    "Alice", "Bob", "Charlie"  
>;  
  
// Cách 1: Lambda  
names.stream()  
.forEach(name ->  
    System.out.println(name)  
>;  
  
// Cách 2: Method reference (ngắn gọn)  
names.stream()  
.forEach(System.out::println);  
  
// Output:  
// Alice  
// Bob  
// Charlie
```

Thực hiện hành động phức tạp

```
List<Student> students = getStudents();  
  
// In thông tin chi tiết  
students.stream()  
.forEach(student -> {  
    System.out.println("Name: " +  
        student.getName());  
    System.out.println("GPA: " +  
        student.getGpa());  
    System.out.println("---");  
>;  
  
// Gửi email cho từng sinh viên  
students.stream()  
.forEach(student ->  
    emailService.send(student.getEmail())  
>;
```

 **Lưu ý:** forEach() là **terminal operation**, kết thúc stream pipeline. Khác với filter/map là intermediate operations có thể chain tiếp.

6.4.4. Chain of Operations - Kết hợp nhiều toán tử

Sức mạnh thực sự của Stream API là khả năng chain (nối) nhiều operations lại với nhau, tạo thành một pipeline xử lý dữ liệu mạnh mẽ.

Ví dụ: Lọc → Chuyển đổi → Sắp xếp

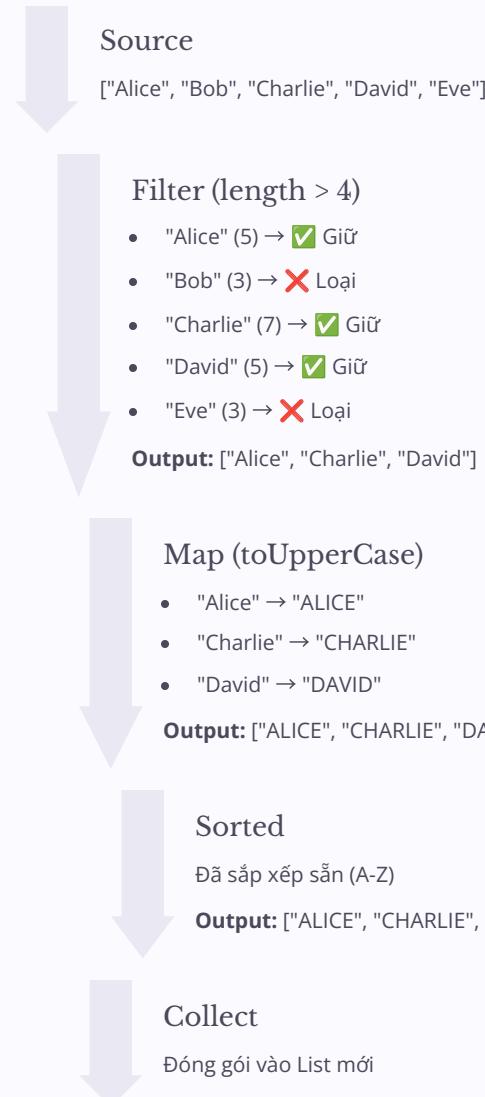
```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");

List<String> result = names.stream()
    .filter(name -> name.length() > 4) // Lọc: ["Alice", "Charlie", "David"]
    .map(String::toUpperCase)        // Chuyển đổi: ["ALICE", "CHARLIE", "DAVID"]
    .sorted()                      // Sắp xếp: ["ALICE", "CHARLIE", "DAVID"]
    .collect(Collectors.toList());   // Thu thập

System.out.println(result);
// [ALICE, CHARLIE, DAVID]
```

Phân tích Dây chuyền (Pipeline Trace)

Hãy nhìn dữ liệu chạy qua băng chuyền như thế nào:



Ví dụ thực tế: Xử lý dữ liệu phức tạp

Hãy xem một ví dụ thực tế với bài toán xử lý danh sách sinh viên:

Bài toán: Tìm Top 3 sinh viên xuất sắc

Yêu cầu: Từ danh sách sinh viên, tìm sinh viên có GPA ≥ 3.5, sắp xếp theo GPA giảm dần, lấy top 3.

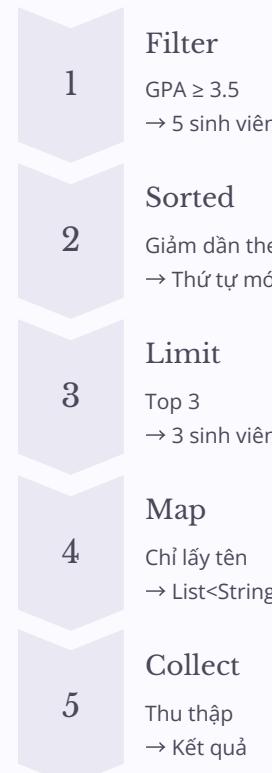
Dữ liệu đầu vào

```
List<Student> students = Arrays.asList(  
    new Student("Alice", 3.8),  
    new Student("Bob", 3.2),  
    new Student("Charlie", 3.9),  
    new Student("David", 3.6),  
    new Student("Eve", 3.4),  
    new Student("Frank", 3.7),  
    new Student("Grace", 3.5)  
>;
```

Xử lý với Stream

```
List<String> topStudents = students.stream()  
    .filter(s -> s.getGpa() >= 3.5) // Lọc GPA ≥ 3.5  
    .sorted((a, b) ->  
        Double.compare(b.getGpa(), a.getGpa()))  
    ) // Sắp xếp giảm dần  
    .limit(3) // Lấy top 3  
    .map(Student::getName) // Lấy tên  
    .collect(Collectors.toList());  
  
System.out.println(topStudents);  
// [Charlie, Alice, Frank]
```

Pipeline Breakdown



6.4.5. Terminal vs Intermediate Operations

Hiểu sự khác biệt giữa Terminal và Intermediate Operations là chìa khóa để sử dụng Stream API hiệu quả.

Phân loại Operations

Intermediate Operations (Trung gian)

- ✓ Trả về Stream
- ✓ Lazy - chỉ thực thi khi có terminal operation
- ✓ Có thể chain nhiều operations
- ✓ Không tiêu thụ stream

Ví dụ:

- filter()
- map()
- sorted()
- distinct()
- limit()
- skip()

Terminal Operations (Kết thúc)

- ✓ Trả về kết quả cụ thể (không phải Stream)
- ✓ Eager - thực thi toàn bộ pipeline
- ✗ Kết thúc stream, không thể chain tiếp
- ✓ Tiêu thụ stream

Ví dụ:

- collect()
- forEach()
- reduce()
- count()
- findFirst()
- anyMatch()

Ví dụ minh họa Lazy Evaluation

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

// Intermediate operations (CHƯA thực thi!)
Stream<Integer> stream = numbers.stream()
    .filter(n -> {
        System.out.println("Filtering: " + n);
        return n % 2 == 0;
    }) // Intermediate
    .map(n -> {
        System.out.println("Mapping: " + n);
        return n * 2;
    }); // Intermediate

System.out.println("Stream created, but not executed yet!");

// Terminal operation (BẮT ĐẦU thực thi!)
List<Integer> result = stream.collect(Collectors.toList()); // Terminal

// Output:
// Stream created, but not executed yet!
// Filtering: 1
// Filtering: 2
// Mapping: 2
// Filtering: 3
// Filtering: 4
// Mapping: 4
// Filtering: 5
```

💡 **Lazy Evaluation:** Intermediate operations không thực thi cho đến khi có terminal operation. Điều này giúp tối ưu hiệu năng - chỉ xử lý những gì cần thiết!

Ví dụ: Pipeline hoàn chỉnh

Hãy xem một ví dụ phân tích rõ ràng sự khác biệt giữa Intermediate và Terminal operations:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// ====== INTERMEDIATE OPERATIONS (Lazy) ======
Stream<Integer> processedStream = numbers.stream()
    .filter(n -> n % 2 == 0)      // Intermediate: Lọc số chẵn
    .map(n -> n * 2)            // Intermediate: Nhân đôi
    .sorted()                   // Intermediate: Sắp xếp
    .distinct();                // Intermediate: Loại trùng

// Đến đây CHƯA có gì được thực thi!
System.out.println("Stream pipeline created!");

// ====== TERMINAL OPERATION (Eager) ======
List<Integer> result = processedStream.collect(Collectors.toList());
// BẮT ĐẦU thực thi TẤT CẢ operations từ trên xuống!

System.out.println(result);
// [4, 8, 12, 16, 20]
```

So sánh Terminal Operations phổ biến

Operation	Trả về	Ví dụ	Use case
collect()	Collection	List, Set, Map	Thu thập kết quả
forEach()	void	In ra, gửi email	Thực hiện hành động
reduce()	T hoặc Optional<T>	Tổng, tích, max	Tổng hợp
count()	long	Đếm phần tử	Thống kê
findFirst()	Optional<T>	Phần tử đầu tiên	Tìm kiếm
anyMatch()	boolean	Có tồn tại không	Kiểm tra điều kiện



TÓM TẮT CHƯƠNG 6

Kiến thức đã học

1

Java Collections Framework

List, Set, Map và cách chọn cấu trúc dữ liệu phù hợp

2

So sánh Implementations

ArrayList vs LinkedList, HashSet vs TreeSet, HashMap vs TreeMap

3

Generics

Type Safety, Generic Class, Generic Method, Wildcards

4

Lambda Expressions

Cú pháp ngắn gọn cho anonymous functions

5

Functional Interfaces

Predicate, Consumer, Function, Supplier, BiFunction

6

Stream API

filter, map, sorted, collect, reduce và các operations

7

Programming Paradigms

Chuyển đổi từ Imperative sang Declarative programming

Kỹ năng đã có

- Chọn cấu trúc dữ liệu phù hợp
- Sử dụng Generics đúng cách
- Viết Lambda Expressions ngắn gọn
- Áp dụng Functional Interfaces
- Xử lý dữ liệu với Stream API
- Code theo phong cách Declarative
- Tối ưu hiệu năng với đúng Collection
- Viết code hiện đại, dễ đọc

🎯 BÀI TẬP CHƯƠNG 6

Bài 1: Collections cơ bản

Mục tiêu: Làm quen với List, Set, Map và các operations cơ bản.

Yêu cầu

1. Tạo List<String> chứa tên 5 sinh viên
2. Tạo Set<Integer> chứa 5 ID duy nhất
3. Tạo Map<String, Double> chứa điểm số (tên → điểm) của 5 sinh viên
4. Duyệt và in tất cả các collections
5. Thực hiện các thao tác: thêm, xóa, kiểm tra tồn tại

Gợi ý

```
List<String> students = new ArrayList<>();
students.add("Alice");
// ... thêm 4 sinh viên nữa

Set<Integer> ids = new HashSet<>();
// ...

Map<String, Double> scores = new HashMap<>();
// ...
```

Bài 2: So sánh ArrayList vs LinkedList

Mục tiêu: Hiểu rõ sự khác biệt về hiệu năng giữa ArrayList và LinkedList.

Yêu cầu

1. Tạo ArrayList và LinkedList, mỗi cái chứa 10,000 phần tử Integer
2. Đo thời gian thực hiện các operations:
 - o Thêm 1,000 phần tử vào **đầu** list
 - o Thêm 1,000 phần tử vào **cuối** list
 - o Truy cập phần tử ở giữa (index 5000) 10,000 lần
 - o Xóa 1,000 phần tử ở giữa list
3. So sánh kết quả và giải thích tại sao

Gợi ý

```
long startTime = System.nanoTime();
// ... thực hiện operation
long endTime = System.nanoTime();
long duration = (endTime - startTime) / 1_000_000; // milliseconds
System.out.println("Time: " + duration + "ms");
```

Bài 3: Generics

Mục tiêu: Thực hành tạo và sử dụng Generic Class, Generic Method.

Yêu cầu

01

Generic Class: Pair<T, U>

Tạo class Pair lưu 2 giá trị khác kiểu

```
public class Pair<T, U> {  
    private T first;  
    private U second;  
  
    // Constructor, getters, setters  
    // toString()  
}
```

02

Generic Method: swap()

Tạo method đổi vị trí 2 phần tử trong mảng

```
public static <T> void swap(  
    T[] array, int i, int j  
) {  
    // TODO: Implement  
}
```

03

Bounded Generic: NumberBox<T>

Tạo class chỉ chấp nhận Number và subclass

```
public class NumberBox<T extends Number> {  
    private T number;  
  
    public double getDoubleValue() {  
        return number.doubleValue();  
    }  
}
```

Test code

```
Pair<String, Integer> pair = new Pair<>("Age", 25);  
System.out.println(pair); // (Age, 25)
```

```
String[] names = {"Alice", "Bob"};  
swap(names, 0, 1);  
System.out.println(Arrays.toString(names)); // [Bob, Alice]
```

```
NumberBox<Integer> intBox = new NumberBox<>();  
intBox.setNumber(42);  
System.out.println(intBox.getDoubleValue()); // 42.0
```

Bài 4: Lambda Expressions

Mục tiêu: Thực hành viết Lambda thay cho Anonymous Inner Class.

Yêu cầu

Viết Lambda Expressions cho các tình huống sau:

1. Sắp xếp List

Sắp xếp List<String> theo độ dài (ngắn → dài)

```
List<String> names = Arrays.asList(  
    "Alice", "Bob", "Charlie", "David"  
,  
    );  
// TODO: Sort by length
```

2. Lọc List

Lọc List<Integer> chỉ lấy số chẵn

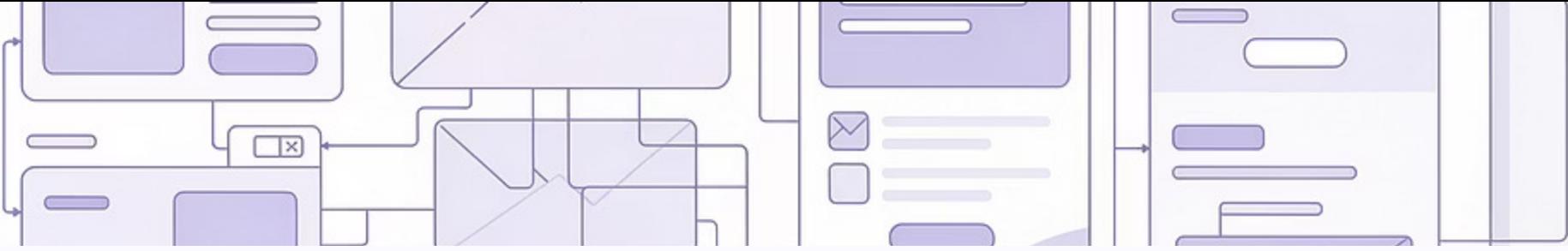
```
List<Integer> numbers =  
    Arrays.asList(1, 2, 3, 4, 5, 6);  
// TODO: Filter even numbers
```

3. Chuyển đổi

Chuyển List<String> thành chữ hoa

```
List<String> names =  
    Arrays.asList("alice", "bob");  
// TODO: Convert to uppercase
```

- ☐ **Lưu ý:** Tất cả phải viết bằng Lambda, không dùng vòng lặp for!



Bài 5: Functional Interfaces

Mục tiêu: Hiểu và sử dụng thành thạo các Functional Interfaces.

Yêu cầu

Tạo và sử dụng các Functional Interface sau:

1

Predicate: Kiểm tra số nguyên tố

```
Predicate<Integer> isPrime = n -> {  
    // TODO: Implement  
    // Trả về true nếu n là số nguyên tố  
};  
  
System.out.println(isPrime.test(7)); // true  
System.out.println(isPrime.test(10)); // false
```

2

Function: Đếm số từ

```
Function<String, Integer> countWords = s -> {  
    // TODO: Implement  
    // Đếm số từ trong chuỗi  
};  
  
System.out.println(  
    countWords.apply("Hello World Java")  
); // 3
```

3

Consumer: In với format đặc biệt

```
Consumer<String> printFormatted = s -> {  
    // TODO: Implement  
    // In với format: "*** [text] ***"  
};  
  
printFormatted.accept("Hello");  
// Output: *** Hello ***
```

Bài 6: Stream API - Xử lý dữ liệu

Mục tiêu: Thành thạo các operations của Stream API.

Dữ liệu đầu vào

```
class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Constructor, getters, setters  
}  
  
List<Student> students = Arrays.asList(  
    new Student("Alice", 20, 3.8),  
    new Student("Bob", 21, 3.2),  
    new Student("Charlie", 19, 3.9),  
    new Student("David", 22, 3.6),  
    new Student("Eve", 20, 3.4),  
    new Student("Frank", 19, 3.7)  
);
```

Yêu cầu

Sử dụng Stream API để:

1. Tìm sinh viên có GPA ≥ 3.5
2. Sắp xếp theo GPA giảm dần
3. Lấy top 3 sinh viên
4. Tính GPA trung bình của tất cả sinh viên
5. Nhóm sinh viên theo tuổi (Map<Integer, List<Student>>)
6. Tìm sinh viên có GPA cao nhất
7. Đếm số sinh viên có tuổi = 20

Bài 7: Tổng hợp - Hệ thống Quản lý Sản phẩm

Mục tiêu: Áp dụng toàn bộ kiến thức đã học vào một project thực tế.

Yêu cầu

Tạo hệ thống quản lý sản phẩm bao gồm:

1. Class Product

```
public class Product {  
    private String id;  
    private String name;  
    private double price;  
    private String category;  
  
    // Constructor, getters, setters, toString()  
}
```

2. Class ProductManager

Sử dụng Collections để lưu trữ và Stream API để xử lý. Implement các methods:

findByCategory

Tìm sản phẩm theo danh mục

```
List<Product> findByCategory(  
    String category  
)
```

findExpensiveProducts

Tìm sản phẩm đắt hơn giá cho trước

```
List<Product> findExpensiveProducts(  
    double minPrice  
)
```

getAveragePrice

Tính giá trung bình

```
double getAveragePrice()
```

getProductsByPriceRange

Tìm trong khoảng giá

```
List<Product> getProductsByPriceRange(  
    double min, double max  
)
```

getTopExpensiveProducts

Lấy n sản phẩm đắt nhất

```
List<Product> getTopExpensiveProducts(  
    int n  
)
```

groupByCategory

Nhóm theo danh mục

```
Map<String, List<Product>>  
groupByCategory()
```

Điều kiện bắt buộc

- ✓ Sử dụng Generics
- ✓ Tất cả methods dùng Stream API
- ✓ Code theo phong cách Declarative (không dùng vòng lặp for)
- ✓ Viết Unit Test cho mỗi method



TÀI LIỆU THAM KHẢO

Để học sâu hơn về các chủ đề trong chương này, bạn có thể tham khảo các nguồn tài liệu chính thức và sách chuyên sâu sau:

Java Collections Tutorial

Tài liệu chính thức từ Oracle về Java Collections Framework, bao gồm chi tiết về List, Set, Map và các implementations.

<https://docs.oracle.com/javase/tutorial/collections/>

Java Stream API Documentation

API reference đầy đủ về Stream API, bao gồm tất cả các operations, ví dụ và best practices.

[Package java.util.stream](#)

Effective Java (3rd Edition)

Cuốn sách kinh điển của Joshua Bloch. **Item 26-37** về Generics và **Item 42-48** về Lambda và Stream là must-read!

Đây là nguồn tài liệu tốt nhất để học best practices khi dùng Generics và Stream API.

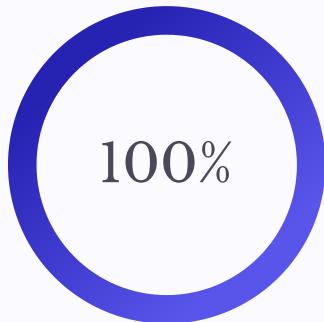
- ❑ **Khuyến nghị:** Sau khi hoàn thành chương này, hãy đọc kỹ các tài liệu trên để củng cố kiến thức và học các best practices từ các chuyên gia!

Chúc bạn học tốt!



Bạn đã hoàn thành Chương 6 - một trong những chương quan trọng nhất trong việc hiện đại hóa kỹ năng lập trình Java của bạn!

🎉 Những gì bạn đã đạt được



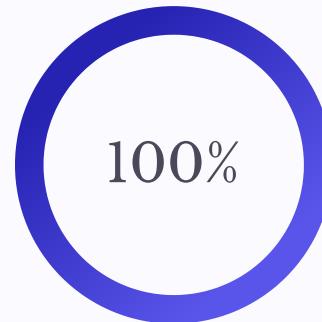
Collections Framework

Thành thạo List, Set, Map



Generics

Code type-safe và reusable



Lambda & Stream

Declarative programming

🚀 Bước tiếp theo

Hãy thực hành các bài tập để củng cố kiến thức. Đừng ngại thử nghiệm và sáng tạo với Stream API - đó là cách tốt nhất để học!

Tips để nhớ lâu

- Làm hết bài tập
- Viết code mỗi ngày
- Đọc code người khác
- Tham gia coding challenges

Liên hệ & Hỗ trợ

Nếu có thắc mắc, đừng ngại hỏi giảng viên hoặc tham gia các cộng đồng Java để được hỗ trợ!

Happy Coding! 🚀✨