



Java Stream API

cuong@techmaster.vn

Tại sao cần nắm vững Java Stream?

- Phần lớn thời gian chúng ta làm việc, xử lý dữ liệu kiểu collection: List, Set, Map
- Code lặp đi lặp lại vừa dài, vừa mất thời gian, đôi khi không tối ưu tốc độ.
- Cần có thư viện để lập trình viên tái sử dụng các hàm dạng khai báo, nối chuỗi các bước xử lý lại để code gọn hơn, mà vẫn cần phải debug được.
- Nắm vững Java Stream giúp bạn code ứng dụng trong sáng hơn, gọn hơn. Java Stream với Java collections cũng như SQL với table

Bài toán đời thường

Có một túi bi gồm các viên bi xanh, đỏ, vàng, tím.

Bạn đều đặn lấy từng viên ra khỏi túi → stream

Nhóm các viên bi cùng màu vào cốc riêng → groupby

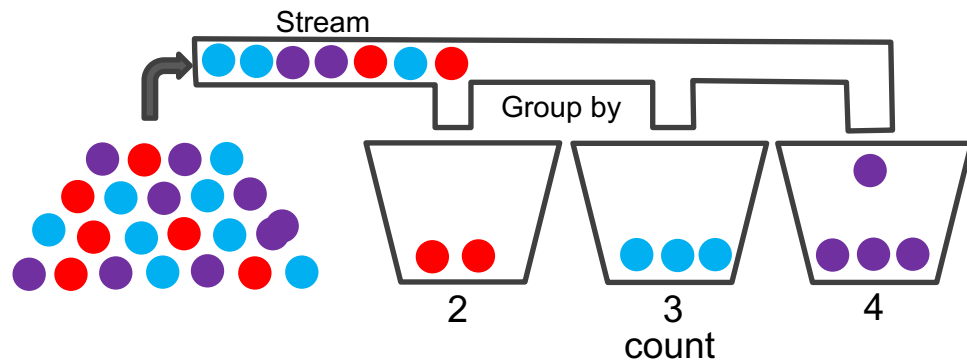
Sau đó đếm số lượng bi ở từng cốc → count

Tìm ở mỗi cốc viên bi có đường kính lớn nhất → maxby

Tính khối lượng trung bình của tất cả các viên bi màu xanh → average

Nhặt ra các viên bi thuỷ tinh trong suốt → collect + filterby

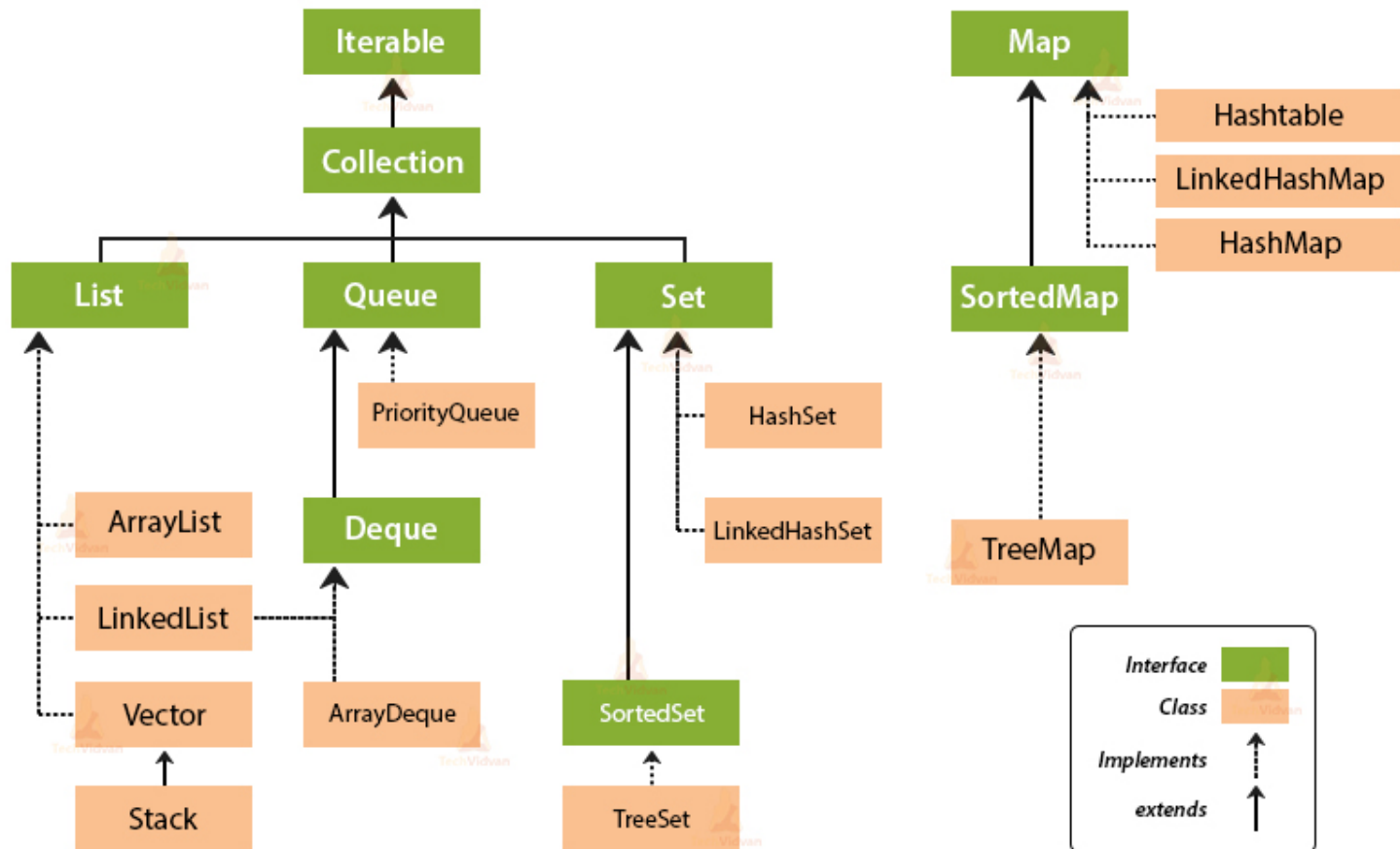




Java Stream dựa trên nền tảng công nghệ gì?

- Các cấu trúc dữ liệu căn bản của Java Core
- Interface
- Generic
- Lambda Expression
- Fluent API: nối chuỗi các bước xử lý

Collection Framework Hierarchy in Java



Interface kết hợp với Generic



```
public interface Stream<T> extends BaseStream<T, Stream<T>>
```

Các class giao tiếp với nhau qua interface giảm tightly coupling

Một thuật toán dùng có nhiều kiểu dữ liệu khác nhau

Lambda function

- Anonymous: không cần đặt tên cụ thể, chỉ cần viết logic
- Function: hàm, chứ không phải method gắn vào một class, object cụ thể
- Pass around: truyền như tham số và lưu như biến
- Concise: ngắn gọn, dùng chỗ nào viết luồn chỗ đó

```
Comparator<String> compareStringLength = new  
Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};
```

```
(s1, s2) -> s1.length() - s2.length()
```

Lambda function

Truyền tham số cho Lambda Expression

Fluent API – nối chuỗi các hàm xử lý

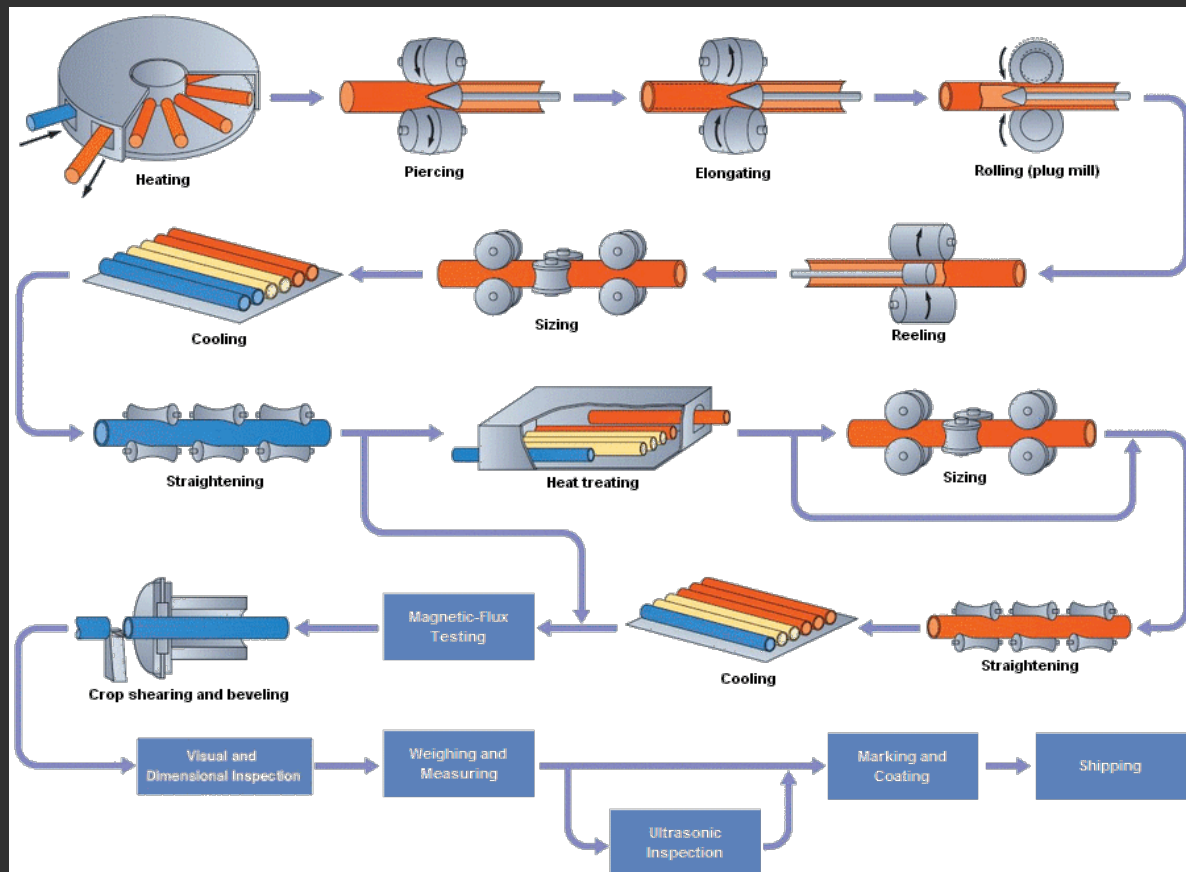
- Giảm thiểu việc khai báo biến
- Kết quả trả về hàm này là đối tượng thực thi hàm tiếp theo

```
List<String> names = List.of("John", "Adam", "Henry",  
"Anna", "Henry");  
  
names.stream().distinct().forEach(System.out::println);
```

Cách học Java Stream hiệu quả nhất

- Hiểu cú pháp Generic, Interface, Fluent API
- Không cần viết lệnh stream api quá phức tạp. Nếu cần thiết kết hợp stream api và lập trình xử lý collection cổ điển.
- Google tìm ví dụ mẫu, copy lại thành một tập các ví dụ viết thành các hàm test. Khi nào cần mở code ra tham khảo.

Nhập môn Stream API



Có bao nhiêu cách để duyệt một List<>?

```
List<String> names = List.of("John", "Adam", "Henry", "Anna",  
"Tí", "Tèo");  
  
for (var name: names) { System.out.println(name); }  
for (int i = 0; i < names.size(); i++) {  
    System.out.println(names.get(i));  
}  
names.forEach(n -> System.out.println(n));  
names.forEach(System.out::println);  
  
names.stream().forEach(System.out::println);
```

stream() và Collectors là gì?

- stream: đưa từng phần tử dữ liệu vào một dây chuyền xử lý tuần tự.
Stream<E> là một generic interface

```
default Stream<E> stream() {  
    return StreamSupport.stream(spliterator(), false);  
}
```

```
public interface Stream<T> extends BaseStream<T, Stream<T>>
```

- Collectors: "Nhà sưu tầm" dùng các phương pháp các nhau để chọn lựa, phân loại, tìm kiếm, sắp xếp phần tử dữ liệu

Collectors

- CH_CONCURRENT_ID
- CH_CONCURRENT_NOID
- CH_ID
- CH_UNORDERED_ID
- CH_NOID
- CH_UNORDERED_NOID

tập các hàm xử lý

- Collectors()
- duplicateKeyException(Object, Object, Object) : Ille...
- uniqKeysMapMerger() <K, V, M extends Map<K, V>...
- uniqKeysMapAccumulator(Function<? super T, ? ext...
- castingIdentity() <I, R> : Function<I, R>
- CollectorImpl<T, A, R>
- toCollection(Supplier<C>) <T, C extends Collection...
- toList() <T> : Collector<T, ?, List<T>>
- toUnmodifiableList() <T> : Collector<T, ?, List<T>>
- toSet() <T> : Collector<T, ?, Set<T>>
- toUnmodifiableSet() <T> : Collector<T, ?, Set<T>>
- joining() : Collector<CharSequence, ?, String>
- joining(CharSequence) : Collector<CharSequence, ?, ...
- joining(CharSequence, CharSequence, CharSequen...
- mapMerger(BinaryOperator<V>) <K, V, M extends M...
- mapping(Function<? super T, ? extends U>, Collect...
- flatMapMapping(Function<? super T, ? extends Stream<...
- filtering(Predicate<? super T>, Collector<? super T, ...

```
public final class Collectors {
```

```
    static final Set<Collector.Characteristics> CH_CO  
        = Collections.unmodifiableSet(EnumSet.of(
```

```
    static final Set<Collector.Characteristics> CH_CO  
        = Collections.unmodifiableSet(EnumSet.of(
```

```
    static final Set<Collector.Characteristics> CH_ID  
        = Collections.unmodifiableSet(EnumSet.of(
```

```
    static final Set<Collector.Characteristics> CH_UN  
        = Collections.unmodifiableSet(EnumSet.of(
```

```
    static final Set<Collector.Characteristics> CH_NO
```

```
    static final Set<Collector.Characteristics> CH_UN  
        = Collections.unmodifiableSet(EnumSet.of(
```

```
    private Collectors() { }
```

```
/**
```

```
 * Construct an {@code IllegalStateException} with
```

predicate

Trong Java 8, **Predicate<T>** là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó. Predicate<T> sẽ trả về giá trị true/false của một tham số kiểu **T** mà bạn đưa vào có thỏa với điều kiện của Predicate đó hay không, cụ thể là điều kiện được viết trong phương thức **test()**.

```
p -> (p.contains("Hen") && (p.length()>3))
```

```
p -> p.getSalary() > 600
```


predicate

@FunctionalInterface

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

- Hàm test trả về giá trị boolean
- Dùng trong các hàm filter, anyMatch, allMatch, noneMatch, takeWhile, dropWhile, partitionBy ...
- Có thể nối and, or, negate

comparator

@FunctionalInterface

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- Hàm compare trả về int
- sort, min, max, minBy, maxBy
- reversed() để nghịch đảo chiều sắp xếp
- thenCompared()

stream().parallel() xử lý song song dùng nhiều thread

- parallel() sẽ không đảm bảo được thứ tự xử lý. Mạnh ai nấy chạy
- Muốn đảm bảo được thứ tự phải dùng **forEachOrdered**

```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Tí", "Tèo");
```

Anna

Tèo

John

```
names.stream().parallel().forEach(System.out::println);
```



Henry

Tí

Adam

John

Adam

Henry

```
names.stream().parallel().forEachOrdered(System.out::println);
```



Anna

Tí

Tèo

map để ánh xạ từng phần tử

```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Henry");
```

```
names.stream().map(p -> p.toUpperCase()).forEach(System.out::println);
```

Lambda function

Function reference

JOHN

ADAM

HENRY

ANNA

HENRY

distinct chọn duy nhất, loại bỏ phần tử lặp

```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Henry");  
names.stream().distinct().forEach(System.out::println);
```

John

Adam

Henry

Anna

filter lọc sử dụng predicate

```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Henry");
```

```
names.stream().filter(p -> (p.contains("Hen") && (p.length()>3)))  
    .forEach(System.out::println);
```

Predicate

Henry

```
names.stream().filter(p -> p.contains("Hen")).filter(x -> x.length()>3)
```

takeWhile tiếp tục chừng nào predicate đúng

```
List<Integer> ints = List.of(4, 4, 4, 5, 6, 7, 8, 4, 10);  
ints.stream().takeWhile(number -> (number/4 == 1)).forEach(System.out::println);
```

4

4

4

5

6

7

dropWhile bỏ qua chừng nào predicate còn đúng

```
List<Integer> ints = List.of(4, 4, 4, 5, 6, 7, 8, 9, 10);  
ints.stream().dropWhile(number -> number % 2 == 0).forEach(System.out::println);
```

5
6
7
8
9
10

Bỏ qua chừng nào điều kiện `number % 2 == 0` còn đúng
4, 4, 4 đều thoả mãn `number % 2 == 0` nên bị bỏ qua drop
Nhưng đến 5 không còn thoả mãn

allMatch trả về true nếu tất cả phần tử thoả mãn Predicate

```
List<Integer> ints = List.of(2, 3, 5, 7, 11, 13, 17);  
Boolean res = ints.stream().allMatch(p -> InStreamTest.isPrime(p));  
if (res) {  
    System.out.println("Tập chứa toàn các số nguyên tố");  
}
```


anyMatch trả về true khi ít nhất một phần tử thoả mãn predicate

```
List<Integer> ints = List.of(4, 4, 4, 5, 6, 7, 8, 4, 10);  
Boolean res = ints.stream().anyMatch(p -> p % 5 == 0);  
if (res) {  
    System.out.println("Tập có chứa ít nhất một số chia hết cho 5");  
}
```

noneMatch true nếu không có bất kỳ phần tử nào thoả mãn predicate

```
List<Integer> ints = List.of(4, 6, 8, 9, 12, 14, 15, 16, 18);  
Boolean res = ints.stream().noneMatch(p ->  
    InStreamTest.isPrime(p));  
if (res) {  
    System.out.println("Tập không chứa bất kỳ số nguyên tố nào");  
}
```

max - min

```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Henry");  
var result = names.stream().max(Comparator.naturalOrder());  
System.out.println(result); // John
```

```
List<Invoice> invoices = List.of(  
    new Invoice("A01", 200, 2), // 400  
    new Invoice("A02", 100, 5), // 500  
    new Invoice("A03", 150, 2)); // 300  
var invoice_with_max_price =  
invoices.stream().max(Comparator.comparing(Invoice::getPrice));  
System.out.println(invoice_with_max_price);  
//BasicStreamTest.Invoice(invoiceNo=A01, price=200, qty=2)
```

Custom comparator

```
List<Invoice> invoices = List.of(
    new Invoice("A01", 200, 2), // 400
    new Invoice("A02", 100, 5), // 500
    new Invoice("A03", 150, 2)); // 300
Comparator<Invoice> compareSubTotal = new Comparator<Invoice>() {
    public int compare(Invoice inv1, Invoice inv2) {
        return inv1.getPrice() * inv1.getQty() - inv2.getPrice() * inv2.getQty();
    }
};
var invoice_with_max_subtotal = invoices.stream().max(compareSubTotal);
System.out.println(invoice_with_max_subtotal);

//Cách viết tắt
var invoice_with_max_subtotal2 = invoices.stream()
    .max((inv1, inv2) -> inv1.getPrice() * inv1.getQty() - inv2.getPrice() * inv2.getQty());
System.out.println(invoice_with_max_subtotal2);
```

Yêu cầu tham số truyền vào max là một đối tượng tuân thủ interface Comparator kiểu truyền vào kế thừa Invoice

```
max(Comparator<? super Invoice> comparator)
```

Kiểu kế thừa từ Invoice

Cách 1: Tạo một đối tượng comparator tham số kiểu Invoice

```
Comparator<Invoice> compareSubTotal = new Comparator<Invoice>() {  
    public int compare(Invoice inv1, Invoice inv2) {  
        return inv1.getPrice() * inv1.getQty() - inv2.getPrice() * inv2.getQty();  
    }  
};
```

Cách 2: Dùng lambda function

```
((inv1, inv2) -> inv1.getPrice() * inv1.getQty() - inv2.getPrice() * inv2.getQty())
```

sorted sử dụng comparator

```
List<String> names = List.of("CCC", "A", "EEEE", "BB", "DDDD");  
Comparator<String> compareStringLength = new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};
```

```
names.stream()  
    .sorted(compareStringLength)  
    .forEach(System.out::println);
```

//Cách viết tắt

```
names.stream()  
    .sorted(((s1, s2) -> s1.length() - s2.length()))  
    .forEach(System.out::println);
```

A
BB
CCC
DDDD
EEEE
A
BB
CCC
DDDD
EEEE

Nối nhiều hàm Comparator với nhau

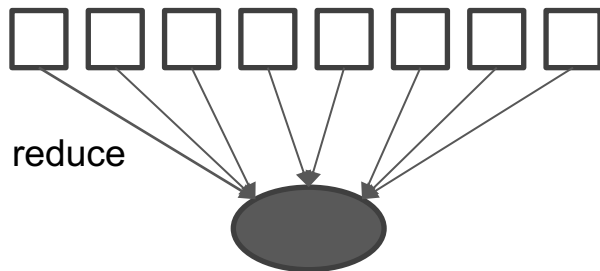
```
List<Emp> emps = List.of(  
    new Emp("John", 20, 2000),  
    new Emp("Anna", 20, 1900),  
    new Emp("Bill", 21, 2100),  
    new Emp("Geogre", 21, 2300),  
    new Emp("Tom", 18, 1500),  
    new Emp("Bob", 30, 1200),  
    new Emp("Jane", 30, 2600)  
);  
emps.stream()  
    .sorted(Comparator.comparing(Emp::getAge)  
        .thenComparing(Comparator.comparing(Emp::getSalary).reversed()))  
    .limit(100)  
    .forEach(System.out::println);
```

```
name=Tom, age=18, salary=1500  
name=John, age=20, salary=2000  
name=Anna, age=20, salary=1900  
name=Geogre, age=21, salary=2300  
name=Bill, age=21, salary=2100  
name=Jane, age=30, salary=2600  
name=Bob, age=30, salary=1200
```

limit(n) giới hạn n phần tử đầu tiên.
skip(n) bỏ qua n phần tử đầu tiên



Reduce từ nhiều phần tử tính toán thu về một kết quả



```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Henry");  
String combinedString = names.stream().reduce("", (result, element) ->  
    result + element + " ");  
System.out.println(combinedString);
```

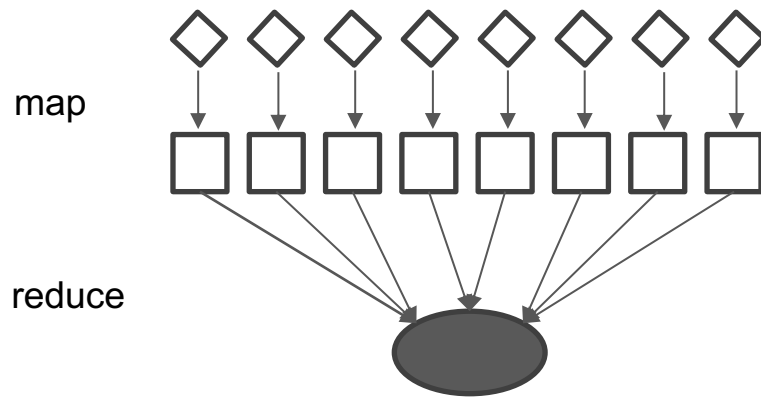
John Adam Henry Anna Henry

Reduce tìm từ dài nhất

```
List<String> names = List.of("John", "Adamson", "Thierry Henry",  
"Annaka", "Tí", "Tèo");  
String longestString = names.stream().reduce("", (word1, word2) ->  
word1.length() > word2.length() ? word1 : word2);  
System.out.println(longestString);
```

Thierry Henry

Map – Reduce ~ Ánh xạ rồi thu gọn



```
@Data
@AllArgsConstructor
class Invoice {
    String invoiceNo;
    int price;
    int qty;
}

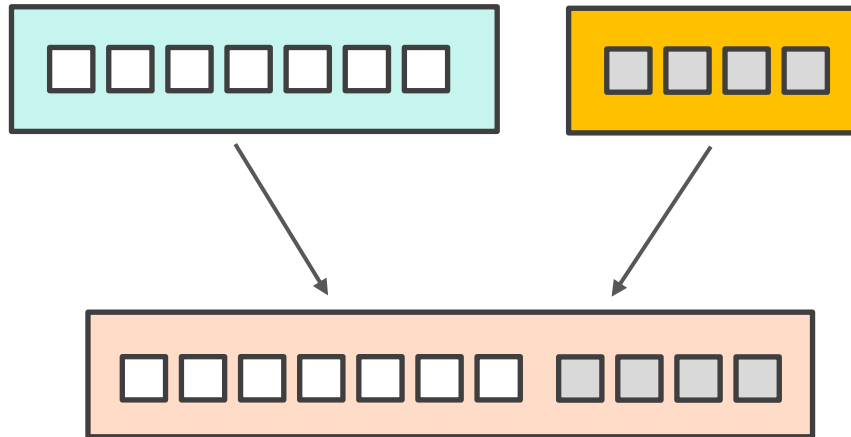
@Test
public void map_reduce() {
    List<Invoice> invoices = List.of(
        new Invoice("A01", 200, 2), // 400
        new Invoice("A02", 100, 5), // 500
        new Invoice("A03", 150, 2)); // 300
```

```
    int result = invoices.stream().map(invoice ->
invoice.getPrice() * invoice.getQty()) // Tính subtotal
    .reduce(0, (total, subTotal) -> total + subTotal);
```

```
    System.out.println(result);
}
```

$$\begin{array}{r} 200 * 2 \\ + 100 * 5 \\ 150 * 2 \\ \hline 1200 \end{array}$$

flatMap: phẳng hoá các collection lồng nhau



```
Developer o1 = new Developer();
o1.setName("mkyong");
o1.addBook("Java 8 in Action");
o1.addBook("Spring Boot in Action");
o1.addBook("Effective Java (3rd Edition)");

Developer o2 = new Developer();
o2.setName("zilap");
o2.addBook("Learning Python, 5th Edition");
o2.addBook("Effective Java (3rd Edition)");
```

```
List<Developer> devs = List.of(o1, o2);

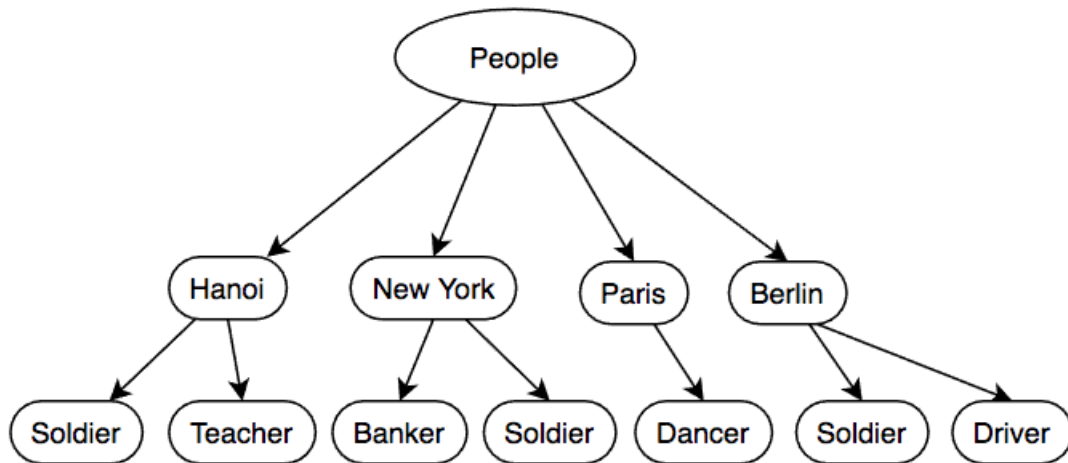
devs.stream().map(x -> x.getBook()).forEach(System.out::println);

Set<String> collect = devs.stream().map(x -> x.getBook()) // Stream<Set<String>>
    .flatMap(x -> x.stream()) // Stream<String>
    .filter(x -> !x.toLowerCase().contains("python")) // filter python book
    .collect(Collectors.toSet()); // remove duplicated

collect.forEach(System.out::println);
```

groupBy

- Group By Then Aggregate (Max, Min, Sum, Count, Average)
- Group By Then Group By



```
List<String> names = List.of("John", "Adam", "Henry", "Anna", "Henry", "Anna",  
"Anna", "Ted");  
names.stream()  
.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))  
.entrySet()  
.forEach(System.out::println);
```

```
Function.identity() == t -> t
```

Áp dụng với phần tử dữ liệu đơn giản, không chứa thuộc tính bên trong

```
Adam=1  
Ted=1  
John=1  
Henry=2  
Anna=3
```


Nhóm theo thành phố

```
//One level grouping  
var cities_population =  
people.stream().collect(Collectors.groupingBy(Person::getCity,  
Collectors.counting()));
```

Nhóm theo thành phố, sau đó theo job

```
//Multi level grouping  
Map<String, Map<String, Long>> result = people.stream().collect(  
Collectors.groupingBy(Person::getCity,  
Collectors.groupingBy(Person::getJob,  
Collectors.counting())));
```

collectingAndThen: collect bước 1 sau đó collect bước 2

Chọn ra nhân viên lương trên 600 rồi mới phân loại phòng ban

```
List<Employee> employees = List.of(
    new Employee("John", "sales", 1000),
    new Employee("Anna", "sales", 1200),
    new Employee("Bob", "tech", 800),
    new Employee("Rock", "hr", 700),
    new Employee("Bill", "tech", 200),
    new Employee("Van", "sales", 300)
);

var res = employees.stream()
    .collect(
        Collectors.collectingAndThen(
            Collectors.filtering(p -> p.getSalary() > 600, Collectors.toList()),
            p -> p.stream().collect(Collectors.groupingBy(Employee::getDept)))
    );
```

Tuy nhiên có cách đơn giản hơn

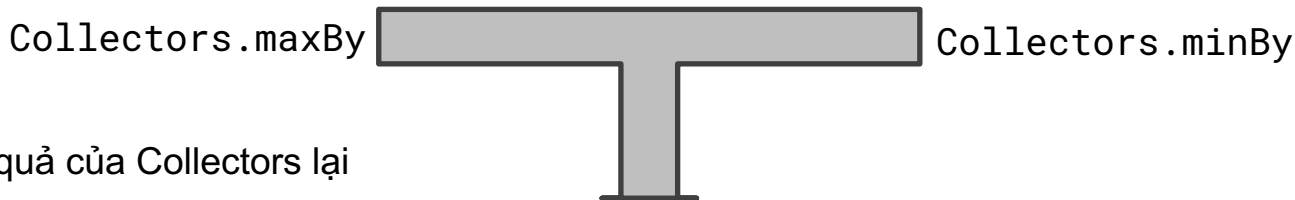
```
List<Employee> employees = List.of(  
    new Employee("John", "sales", 1000),  
    new Employee("Anna", "sales", 1200),  
    new Employee("Bob", "tech", 800),  
    new Employee("Rock", "hr", 700),  
    new Employee("Bill", "tech", 200),  
    new Employee("Van", "sales", 300)  
);  
  
var res2 = employees.stream()  
    .filter(p -> p.getSalary() > 600) //Lọc  
    .collect(Collectors.groupingBy(Employee::getDept)); //Rời phân nhóm
```

partitionBy: chia thành 2 nhóm theo predicate

```
List<Employee> employees = List.of(
    new Employee("John", "sales", 1000),
    new Employee("Anna", "sales", 1200),
    new Employee("Bob", "tech", 800),
    new Employee("Rock", "hr", 700),
    new Employee("Bill", "tech", 200),
    new Employee("Van", "sales", 300)
);
/*
Chia thành 2 nhóm nhân viên: lương lớn 600 và phần còn lại
*/
var res = employees.stream().collect(
    Collectors.partitioningBy(p -> p.getSalary() > 600 ));
```

tee

```
HashMap<String, Employee> result = employees.stream().collect(
    Collectors.teeing(
        Collectors.maxBy(Comparator.comparing(Employee::getSalary)),
        Collectors.minBy(Comparator.comparing(Employee::getSalary)),
        (e1, e2) -> {
            HashMap<String, Employee> map = new HashMap();
            map.put("MAX", e1.get());
            map.put("MIN", e2.get());
            return map;
        }
    ));
```



tee : ghép 2 kết quả của Collectors lại

(e1, e2) -> { }

Dùng tee tính tỷ lệ nam/nữ trong mỗi thành phố

```
List<Person> people = personRepository.getAll();
var maleVsFemalePerCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity,
    Collectors.teeing(
        Collectors.filtering(e -> e.getGender().equals("Male"),
        Collectors.counting()),
        Collectors.filtering(e -> e.getGender().equals("Female"),
        Collectors.counting()),
        (male, female) -> {
            return (double)male / female;
        }
    ))));
```

Viết Unit Test với AssertJ

Viết Test cho REST API

- Bản chất của back end đó là xử lý dữ liệu và trả về REST do đó viết unit test cho Repository và Service để kiểm tra hợp lệ, tính đúng đắn của code là cần thiết nhưng không thể máy móc.
- Có 2 cách:
 - C1: Viết code chức năng rồi mới viết unit test
 - C2: Xen kẽ code chức năng và viết unit test

C1: Code chức năng xong code unit test

- Biểu hiện xu hướng viết tighgly coupling
- Unit test chỉ còn ý nghĩa hậu kiểm, hoặc dùng khi code chức năng thay đổi cần chạy lại unit test để kiểm tra xem có chỗ nào broken
- Khi chức năng đã xong, cách mà lập trình viên unit test chỉ để tạo cảm giác yên tâm là tôi đã viết đủ unit test. Chứ unit test không có tác dụng đảm bảo tính đúng đắn của chức năng.

Khi bạn đã có được bằng tốt nghiệp đại học, liệu bạn có muốn quay lại trường để làm bài kiểm tra không?

C2: Code chức năng xong xong với code unit test

- Nếu cấu trúc phần mềm bạn viết thực sự loosely coupling, ít ràng buộc, bạn có thể chia nhỏ chức năng để viết những đoạn code thử nghiệm trong unit test
- Code đến đâu, viết unit test đến đó. Thậm chí viết unit test để thử nghiệm trước giúp bạn tự tin hơn nhiều khi tích hợp nhiều hàm nhỏ nhỏ lại.

Hãy tận dụng tối đa Assertj + FluentAPI để kiểm thử

@Autowired

private FilmRepository filmRepo;

@Test

```
public void getAll() {  
    List<Film> filmList = filmRepo.getAll();  
    assertEquals(filmList.size(), 30);  
}
```

@Test

```
public void getFilmByCountry() {  
    Map<String, List<Film>> filmListByCountry = filmRepo.getFilmByCountry();  
    assertEquals(filmListByCountry.get("China").size(), 9);  
    assertEquals(filmListByCountry.get("Germany").size(), 1);  
}
```

@Test

```
public void getcountryMakeMostFilms() {  
    Entry<String, Integer> counter = filmRepo.getcountryMakeMostFilms();  
    assertEquals(counter.getKey(), "China");  
    assertEquals(counter.getValue(), 9);  
}
```

Cách viết unit test

quá phụ thuộc vào dữ liệu đầu vào,
không có tính tổng quát

Hàm unit test càng ít phụ thuộc vào biến thiên dữ liệu đầu vào càng tốt

```
@Test
@DisplayName("sortPeopleByFullNameReversed: sắp xếp theo tên thứ tự Z-A")
public void sortPeopleByFullNameZA() {
    List<Person> people = personRepository.sortPeopleByFullNameReversed();
    personRepository.printListPeople(people);
    assertThat(people).isSortedAccordingTo(
        Comparator.comparing(Person::getFullname).reversed());
}
```

Chủ động cấu hình dữ liệu ngay trong hàm test

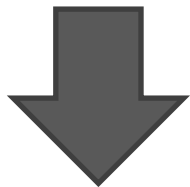
Chỉ đúng khi ngày test giữa
20/11/2021 và 27/11/2021

```
@Test
//Chỉ dùng để kiểm tra hàm tính tuổi
public void testGetAge() {
    Person john = new Person();
    john.setBirthday("1975/11/27");
    assertThat(john.getAge()).isEqualTo(45);

    Person jim = new Person();
    jim.setBirthday("1975/11/20");
    assertThat(jim.getAge()).isEqualTo(46);
}
```

Chọn cách lập trình khác để kiểm tra lại code

```
public Map<String, List<Person>> groupPeopleByCity() {  
    return people.stream().collect(Collectors.groupingBy(Person::getCity));  
}
```



Trên dùng groupingBy thì dưới dùng distinct
thậm chí viết code kiểu khác để kiểm tra

```
@Test  
public void groupPeopleByCity() {  
    var city_people = personRepository.groupPeopleByCity();  
    List<Person> people = personRepository.getAll();  
    List<String> cities = people.stream()  
        .map(Person::getCity) //ánh xạ trường City ra stream  
        .distinct() //chọn duy nhất  
        .peek(System.out::println) //Dùng peek để in ra debug  
        .toList(); //chuyển sang dạng List  
    assertThat(cities).containsAll(city_people.keySet());  
}
```

