VIETNAM AUSTRALIA
Vocational College

Slide-2.2: Lambda Expression, forEach, Stream API

Mentor: Nguyễn Bá Minh Đạo

Nội dung

1. Lambda Expressions Java 8
2. Phương thức forEach()
3. Stream API trong Java 8

Lambda Expressions trong Java 8

Lambda Expressions là gì?

- **Lambda Expressions (Biểu thức Lambda)** là một trong những tính năng mới được giới thiệu trong Java 8.
- **Trước** khi **Java 8**, mọi thứ chủ yếu là hướng đối tượng. Ngoại trừ các kiểu dữ liệu nguyên thủy (primitive type), mọi thứ trong java tồn tại dưới dạng đối tượng.
  => Tất cả các lời gọi đến các **method/function** sẽ được thực hiện bằng cách sử dụng các **class** hoặc **object**. Các **method/function** không tồn tại độc lập.

Lambda Expressions trong Java 8

Lambda Expressions là gì?

- Với **Java 8**, lập trình chức năng (functional programming) đã được giới thiệu. Vì vậy, chúng ta có thể dễ dàng sử dụng các chức năng ẩn danh (anonymous functions).
- Nó tạo điều kiện cho các lập trình viên lập trình **Functional** và phát triển ứng dụng đơn giản hơn rất nhiều so với những phiên bản trước đó => cung cấp một cách rõ ràng và ngắn ngọn để đại diện cho một **Functional Interface** sử dụng một biểu thức Lamda.

Lambda Expressions trong Java 8

Lambda Expressions là gì?

- **Lambda Expression** (biểu thức Lambda) có thể được định nghĩa là một hàm ẩn danh, cho phép người dùng chuyển các phương thức làm đối số. Điều này giúp loại bỏ rất nhiều mã soạn sẵn.
- **Lambda Expression** là một hàm không có tên và không thuộc bất kỳ lớp nào, không có phạm vi truy cập (private, public hoặc protected), không khai báo kiểu trả về.

Lambda Expressions trong Java 8

Tại sao nên sử dụng Lambda Expressions?

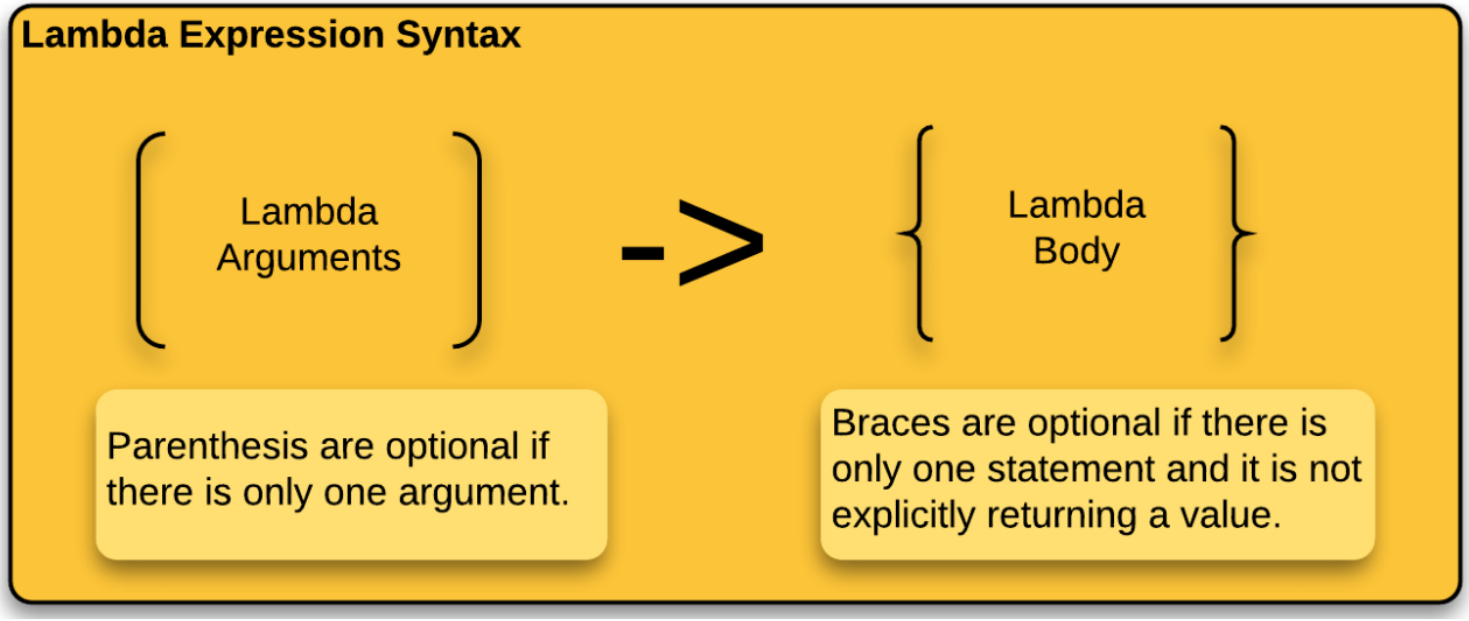
- ❑ Cung cấp bản implement cho Functional interface.
- ❑ Viết ít code hơn
- ❑ Hiệu quả hơn nhờ hỗ trợ thực hiện tuần tự (sequential) và song song (parallel) thông qua Stream API.

Lambda Expressions trong Java 8

Cú pháp Lambda Expressions

(argument-list) -> {body}

- Biểu thức **Lambda** bao gồm 3 thành phần sau:
  - **Argument-list:** có thể không có, có một hoặc nhiều tham số.
  - **Arrow-token:** được sử dụng để liên kết arguments-list và body của biểu thức.
  - **Body:** chứa các biểu thức và câu lệnh cho biểu thức lambda.



Lambda Expressions trong Java 8

Sử dụng Lambda Expressions

- Có thể viết **lambda expression** bằng nhiều cách tùy thuộc vào việc sử dụng:
  - **Tùy chọn khai báo kiểu dữ liệu:** Chúng ta không cần phải khai báo kiểu dữ liệu cho các **parameter** truyền vào. **Trình biên dịch** sẽ tự suy luận ra kiểu dữ liệu từ giá trị của các **parameter**.
  - **Tùy chọn sử dụng dấu ngoặc ():** Trong trường hợp bạn chỉ truyền vào một **parameter** duy nhất thì chúng ta có thể bỏ qua cặp dấu ngoặc **()**. Nếu như có nhiều **parameter** thì phải sử dụng dấu ngoặc.

```
// No argument and one-statement method body

() -> expression

// One argument and one-statement method body

(parameters) -> expression

// Arguments separated by commas and block body

(arg1, arg2, ...) -> {
    body-block
}

// With arguments and block body, return value

(arg1, arg2, ...) -> {
    body-block;
    return return-value;
}
```

Lambda Expressions trong Java 8

Sử dụng Lambda Expressions

- Có thể viết **lambda expression** bằng nhiều cách tùy thuộc vào việc sử dụng:
  - **Tùy chọn sử dụng dấu ngoặc {}**: Trong trường hợp phần **body code** của chúng ta chỉ thực hiện 1 **statement** duy nhất thì chúng ta cũng có thể loại bỏ luôn cặp dấu ngoặc **{}**.
  - **Tùy chọn sử dụng lệnh return**: Trong biểu thức **Lambda**, nếu chỉ có một câu lệnh, bạn có thể sử dụng hoặc không sử dụng từ khoá **return**. Bạn phải sử dụng từ khóa **return** khi **biểu thức lambda** chứa nhiều câu lệnh.

```
// No argument and one-statement method body

() -> expression

// One argument and one-statement method body

(parameters) -> expression

// Arguments separated by commas and block body

(arg1, arg2, ...) -> {
    body-block
}

// With arguments and block body, return value

(arg1, arg2, ...) -> {
    body-block;
    return return-value;
}
```

Lambda Expressions trong Java 8

So sánh Lambda Expressions và Method

❑ Một phương thức (**method/function**) gồm các phần chính sau:

- ➢ **Name:** tên phương thức.
- ➢ **Parameter list:** danh sách các tham số.
- ➢ **Body:** biểu thức, câu lệnh xử lý.
- ➢ **return type:** kiểu dữ liệu trả về.

❑ Một biểu thức Lambda (**Lambda Expression**) gồm các phần chính sau:

- ➢ **No Name:** không có tên phương thức - phương thức ẩn danh (anonymous method).
- ➢ **Parameter list:** danh sách các tham số.
- ➢ **Body:** biểu thức, câu lệnh xử lý.
- ➢ **No return type:** không có kiểu dữ liệu trả về tường minh, trình biên dich Java 8 có thể tự suy luận ra kiểu dữ liệu trả về dựa vào code thực thi.

Lambda Expressions trong Java 8

Sử dụng Lambda Expressions

- Để sử dụng biểu thức lambda, chúng ta cần tạo **Functional interface** của riêng mình hoặc sử dụng Functional interface do Java cung cấp.
- Sử dụng Functional interface:
  - Trước Java 8: chúng ta tạo anonymous inner classes.
  - Từ Java 8: sử dụng biểu thức lambda thay vì các anonymous inner classes.

Lambda Expressions trong Java 8

Ví dụ sử dụng Lambda Expressions

```java
public class SortBefore8Example {

    public static void main(String[] args) {

        List<String> languages = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        Collections.sort(languages, new Comparator<String>() {

            @Override
            public int compare(String o1, String o2) {
                return o1.compareTo(o2);
            }

        });

        for (String language : languages) {
            System.out.println(language);
        }
    }
}
```

```java
public class SortJava8Example {

    public static void main(String[] args) {

        List<String> languages = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        Collections.sort(languages, (String o1, String o2) -> {
            return o1.compareTo(o2);
        });

        for (String language : languages) {
            System.out.println(language);
        }
    }
}
```

Lambda Expressions trong Java 8

Ví dụ Lambda Expressions không có tham số và có 1 tham số duy nhất

```java
@FunctionalInterface
interface Sayable1 {
    public String say();
}

public class LambdaExpressionExample01 {
    public static void main(String[] args) {
        Sayable1 s = () -> {
            return "I have nothing to say.";
        };
        System.out.println(s.say());
    }
}
```

```java
@FunctionalInterface
interface Sayable2 {
    public String say(String name);
}

public class LambdaExpressionExample02 {
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable2 s1 = (name) -> {
            return "Hello, " + name;
        };
        System.out.println(s1.say("Java Coder"));

        // Shorter
        Sayable2 s2 = name -> {
            return "Hello, " + name;
        };
        System.out.println(s2.say("Java Coder"));

        // Shortest
        Sayable2 s3 = name -> "Hello, " + name;
        System.out.println(s3.say("Java Coder"));
    }
}
```

Lambda Expressions trong Java 8

Ví dụ Lambda Expressions có nhiều tham số, sử dụng hoặc không sử dụng từ khóa return

```java
@FunctionalInterface
interface Addable {
    int add(int a, int b);
}

public class LambdaExpressionExample03 {
    public static void main(String[] args) {

        // Multiple parameters with data type in lambda expression
        Addable ad1 = (int a, int b) -> (a + b);
        System.out.println(ad1.add(10, 20));

        // Multiple parameters in lambda expression
        Addable ad2 = (a, b) -> (a + b);
        System.out.println(ad2.add(10, 20));

        // Lambda expression without return keyword.
        Addable ad3 = (a, b) -> (a + b);
        System.out.println(ad3.add(10, 20));

        // Lambda expression with return keyword.
        Addable ad4 = (a, b) -> {
            return (a + b);
        };
        System.out.println(ad4.add(10, 20));

        // Lambda expression without return keyword.
        Addable ad5 = (a, b) -> (a + b);
        System.out.println(ad5.add(10, 20));

        // Lambda expression with multi-statement
        Addable ad6 = (a, b) -> {
            int sum = (a + b);
            return sum;
        };
        System.out.println(ad6.add(10, 20));
    }
}
```

Lambda Expressions trong Java 8

Ví dụ Lambda Expressions với vòng lặp forEach

```
public class LambdaExpressionExample04 {
    public static void main(String[] args) {

        List<String> languages = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        // Using Lambda expression
        languages.forEach(n -> System.out.println(n));
    }
}
```

Ví dụ Lambda Expressions tạo Thread

```
public class LambdaExpressionExample05 {
    public static void main(String[] args) {
        // Using an anonymous inner class
        Runnable r1 = new Runnable() {
            public void run() {
                System.out.println("Runnable 1");
            }
        };

        // Using Lambda Expression for Funcational Interface
        Runnable r2 = () -> System.out.println("Runnable 2");

        r1.run();
        r2.run();
    }
}
```

Lambda Expressions trong Java 8

Ví dụ Lambda Expressions với Filter Collection Data

```
class Product {
    int id;
    String name;
    float price;

    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}

public class LambdaExpressionExample06 {
    public static void main(String[] args) {

        List<Product> list = new ArrayList<>();
        list.add(new Product(1, "Samsung A5", 17000f));
        list.add(new Product(3, "Iphone 6S", 65000f));
        list.add(new Product(2, "Sony Xperia", 25000f));
        list.add(new Product(4, "Nokia Lumia", 15000f));
        list.add(new Product(5, "Redmi4 ", 26000f));
        list.add(new Product(6, "Lenevo Vibe", 19000f));

        // using lambda to filter data
        Stream<Product> filtered_data = list.stream().filter(p -> p.price > 20000);

        // using lambda to iterate through collection
        filtered_data.forEach(product -> System.out.println(product.name + ": " + product.price));
    }
}
```

Lambda Expressions trong Java 8

Phạm vi truy cập trong Lambda Expressions

- Việc truy cập các biến phạm vi bên ngoài từ các biểu thức lambda rất giống với các đối tượng ẩn danh (anonymous objects).
- Bạn có thể truy cập bất kỳ biến final, static hoặc biến chỉ được gán một lần.
- Biểu thức Lambda throw 1 lỗi biên dịch, nếu một biến được gán một giá trị lần thứ 2.

Lambda Expressions trong Java 8

Truy cập biến local – Accessing local variable

❑ Chúng ta có thể truy cập các biến final và biến chỉ được gán một lần

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}

public class Java8Scope01 {
    public static void doSomething1() {
        final int num = 1;
        Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);

        stringConverter.convert(2); // 3
    }

    public static void doSomething2() {
        int num = 1;
        Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);

        stringConverter.convert(2); // 3
    }

    public static void doSomething3() {
        int num = 1;
        Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);

        stringConverter.convert(2); // 3

        // Lambda expression will throw error at num varialbe.
        // Local variable num defined in an enclosing scope must be final or effectively
        // num = 3;
    }
}
```

Lambda Expressions trong Java 8

Truy cập fields và biến static – Accessing fields and static variable

❑ Chúng ta có thể truy cập và thay đổi các trường hoặc biến static.

```
public class Java8Scope02 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 1;
            return String.valueOf(from);
        };

        Converter<Integer, String> stringConverter2 = (from) -> {
            outerStaticNum = 1;
            return String.valueOf(from);
        };
    }
}
```

Truy cập với Default Method trong Interface

❑ **Default Methods** không thể truy cập bên trong **Lambda Expressions**

Phương thức forEach() trong Java 8

Giới thiệu

- Phương thức **forEach()** là một tính năng mới của Java 8.
- Nó là một **Default Method** được định nghĩa trong interface **Iterable** và **Stream**.
- Các lớp **Collection** extends từ interface **Iterable** có thể sử dụng vòng lặp **forEach()** để duyệt phần tử.

```
50    /**
51     * Performs the given action for each element of the {@code Iterable}
52     * until all elements have been processed or the action throws an
53     * exception.  Actions are performed in the order of iteration, if that
54     * order is specified.  Exceptions thrown by the action are relayed to the
55     * caller.
56     * <p>
57     * The behavior of this method is unspecified if the action performs
58     * side-effects that modify the underlying source of elements, unless an
59     * overriding class has specified a concurrent modification policy.
60     *
61     * @implSpec
62     * <p>The default implementation behaves as if:
63     * <pre>{@code
64     *     for (T t : this)
65     *         action.accept(t);
66     * }</pre>
67     *
68     * @param action The action to be performed for each element
69     * @throws NullPointerException if the specified action is null
70     * @since 1.8
71     */
72    default void forEach(Consumer<? super T> action) {
73        Objects.requireNonNull(action);
74        for (T t : this) {
75            action.accept(t);
76        }
77    }
```

Phương thức forEach() trong Java 8

Ví dụ forEach() với Map/List

```java
import java.util.HashMap;
import java.util.Map;

public class ForEachMapExample {

    public static void main(String[] args) {
        Map<Integer, String> hmap = new HashMap<Integer, String>();
        hmap.put(1, "Java");
        hmap.put(2, "JavaScript");
        hmap.put(3, "PHP");
        hmap.put(4, "C#");
        hmap.put(5, "C++");

        // forEach to iterate and display each key and value pair of HashMap
        hmap.forEach((key, value) -> System.out.println(key + " - " + value));
    }
}
```

Ví dụ forEachOrdered()

```java
import java.util.Arrays;
import java.util.List;

public class ForEachOrderedExample {

    public static void main(String[] args) {
        List<String> languages = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        System.out.println("Iterating by passing lambda expression: ");
        languages.stream().forEachOrdered(lang -> System.out.println(lang));

        System.out.println("Iterating by passing method reference: ");
        languages.stream().forEachOrdered(System.out::println);
    }
}
```

```java
import java.util.Arrays;
import java.util.List;

public interface ForEachListExample {

    public static void main(String[] args) {
        List<String> languages = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        System.out.println("Iterating by passing lambda expression: ");
        languages.forEach(lang -> System.out.println(lang));

        System.out.println("Iterating by passing method reference: ");
        languages.forEach(System.out::println);
    }
}
```

Stream API trong Java 8

Giới thiệu về Stream API trong Java 8

- **Stream** (luồng) là một đối tượng mới của Java được giới thiệu từ phiên bản Java 8, giúp cho việc thao tác trên collection và array trở nên dễ dàng và tối ưu hơn.
- Một Stream đại diện cho một chuỗi các phần tử hỗ trợ các hoạt động tổng hợp tuần tự (sequential) và song song (parallel).
- Tất cả các class và interface của Stream API nằm trong gói **java.util.stream**

Stream API trong Java 8

Ví dụ sử dụng Stream API trong Java 8

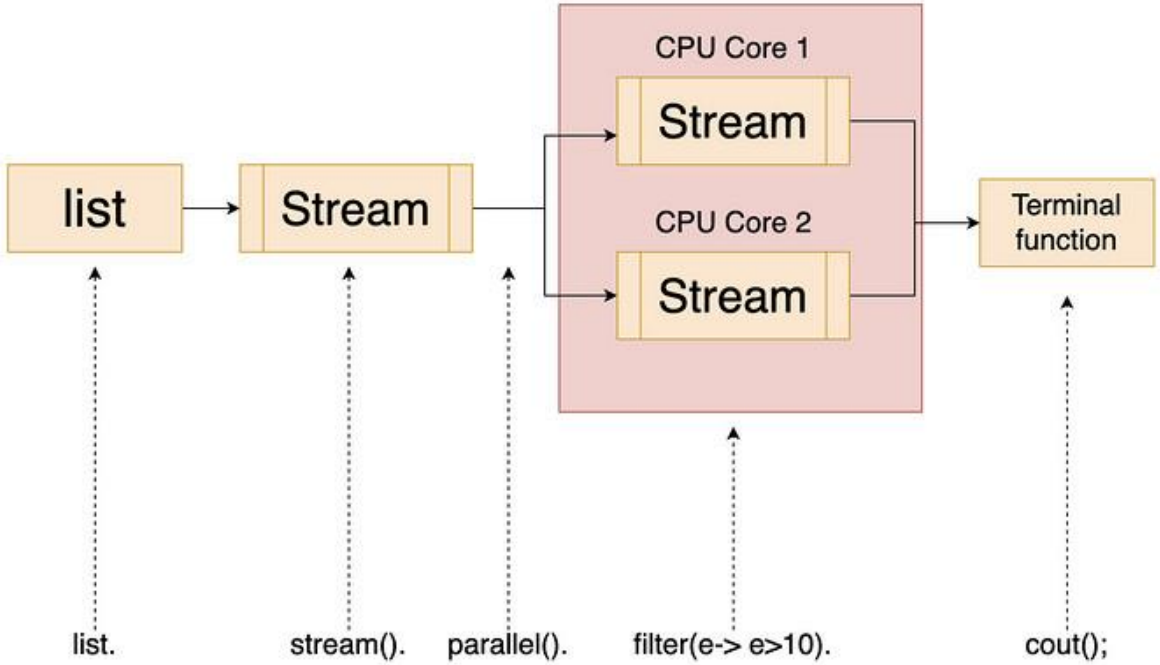
```
public class StreamExample01 {

    List<Integer> numbers = Arrays.asList(7, 2, 5, 4, 2, 1);

    public void withoutStream() {
        long count = 0;
        for (Integer number : numbers) {
            if (number % 2 == 0) {
                count++;
            }
        }
        System.out.printf("There are %d elements that are even", count);
    }

    public void withStream() {
        long count = numbers.stream().filter(num -> num % 2 == 0).count();
        System.out.printf("There are %d elements that are even", count);
    }

}
```

- **2 phương thức trên** đều cho kết quả giống nhau nhưng sử dụng **stream** ngắn gọn hơn
- Với hàm **withoutStream()** lặp toàn bộ danh sách không xử lý song song.
- Với hàm **withStream()**, các phương thức **stream()**, **filter()**, **count()** đang xảy ra song song => sử dụng Stream API sẽ cải thiện hiệu suất chương trình.

Stream API trong Java 8

Một số phương thức của Stream

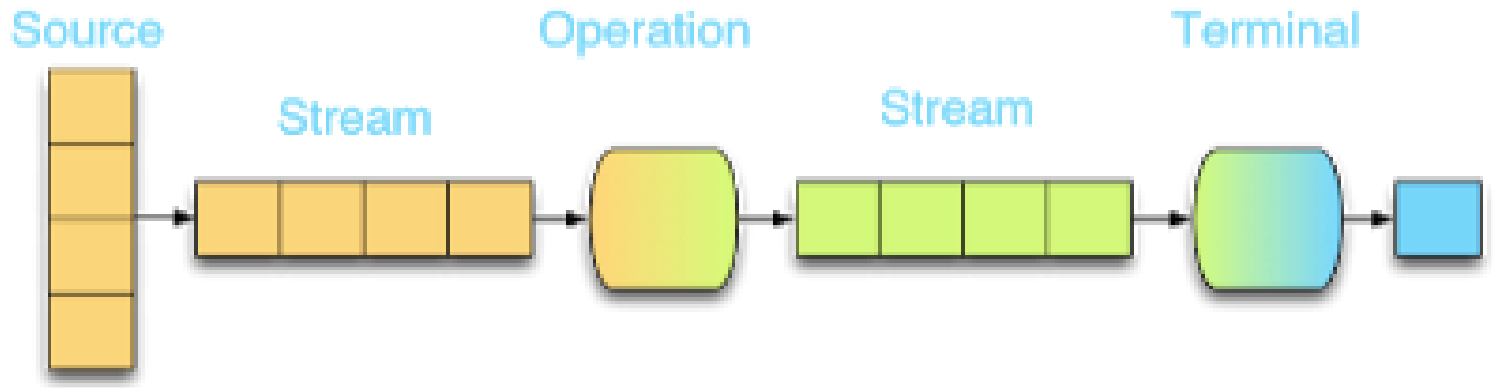
- Trong Java 8, **Collection interface** được hỗ trợ bởi 2 phương thức để tạo ra **Stream**:
  - **stream()**: trả về 1 stream sẽ được xử lý theo tuần tự.
  - **parallelStream()**: trả về 1 stream song song, các xử lý sau đó sẽ thực hiện song song



Stream API trong Java 8

Các đặc điểm của Stream

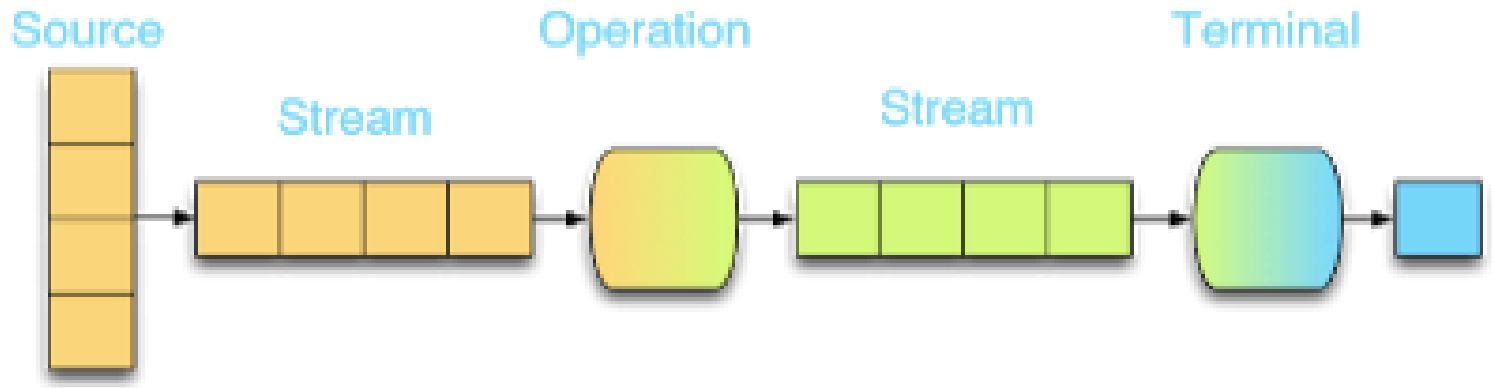
- **Stream** không lưu trữ các phần tử của **collection** hay **array**.
- **Stream** chỉ thực hiện các phép toán tổng hợp (ví dụ: **filter()** và **count()** mà chúng ta đã thấy trong ví dụ trên để có được **stream** dữ liệu mong muốn).
- **Stream** không phải là một cấu trúc dữ liệu (**data structure**)
- **Stream** là immutable object. Các hoạt động tổng hợp mà chúng ta thực hiện trên **Collection**, **Array** không làm thay đổi dữ liệu của nguồn, chúng chỉ trả lại **stream** mới.



Stream API trong Java 8

Các đặc điểm của Stream

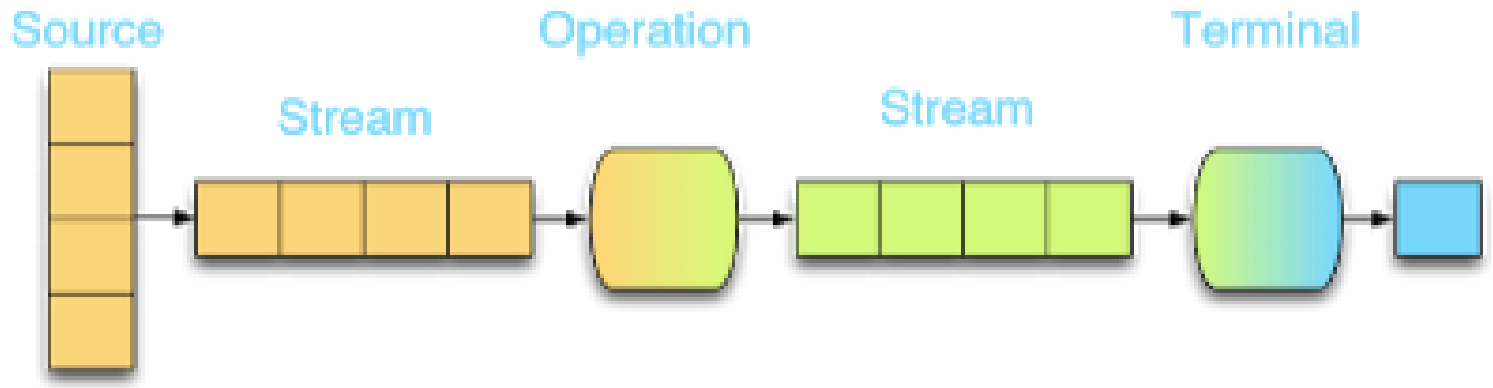
- Tất cả các hoạt động **stream** là lazy (lười biếng), có nghĩa là chúng không được thực hiện cho đến khi cần thiết.
- Để hiện thực cơ chế lazy, hầu hết các thao tác với **Stream** đều return lại một Stream mới, giúp tạo một mắc xích bao gồm một loạt các thao tác nhằm thực thi các thao tác đó một cách tối ưu nhất.
- Mắt xích này còn được gọi là **pipeline**.



Stream API trong Java 8

Các đặc điểm của Stream

- **Các phần tử** của **luồng** chỉ được truy cập một lần trong suốt vòng đời của **Stream**. Giống **Iterator**, một **Stream** mới được tạo ra để duyệt lại các phần tử dữ liệu nguồn.
- **Stream** không dùng lại được, nghĩa là một khi đã sử dụng nó xong, chúng ta không thể gọi nó lại để sử dụng lần nữa.
- Chúng ta không thể dùng **index** để truy xuất các phần tử trong **Stream**.
- **Stream** hỗ trợ thao tác song song các phần tử trong **Collection** hay **Array**.



Stream API trong Java 8

So sánh Streams với Collections

Stream	Collection
- Không phải là một cấu trúc dữ liệu - Stream là một luồng thực hiện tính toán các phần tử theo yêu cầu	- Collection là một cấu trúc dữ liệu chứa các phần tử trong bộ nhớ - Các phần tử trong Collection sẽ được tính toán trước khi được thêm vào Collection
Các Collection có các yếu tố tính tức thời (eager)	Các Stream có yếu tố tính lười biếng (lazy)
Mặc dù chúng ta có thể tạo **Stream** từ Collection và sử dụng một số phương thức trên Collection. Tuy nhiên, Collection gốc vẫn không thay đổi. Do đó, *Stream không thể thay đổi dữ liệu*.	Và một đặc điểm quan trọng của **Stream** là chúng *có thể chuyển đổi dữ liệu*, vì các hoạt động trên Stream có thể tạo ra một cấu trúc dữ liệu khác

Stream API trong Java 8

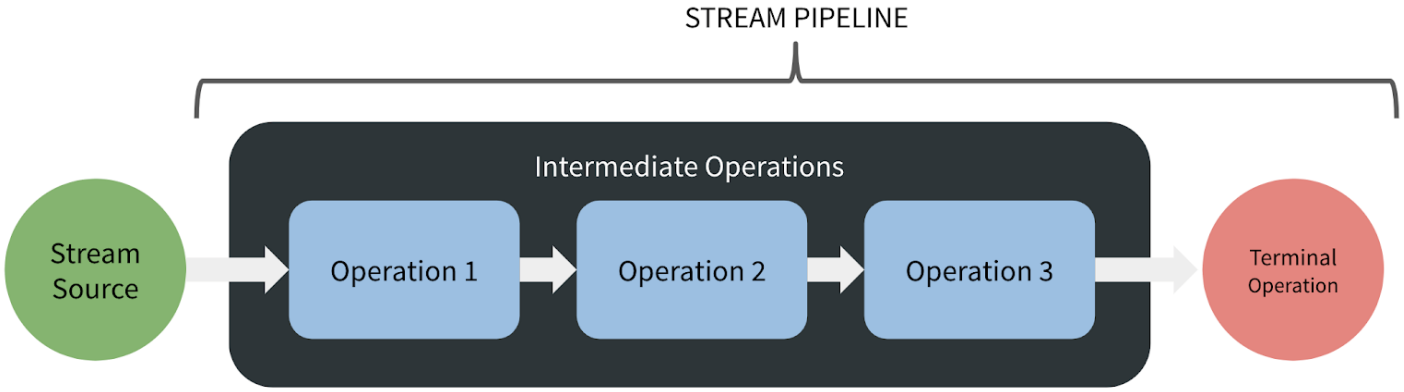
Cách làm việc với Stream

❑ Hoạt động của luồng được giải thích theo ba giai đoạn:

➢ Tạo stream (**stream source**).

➢ Thực hiện các thao tác trung gian (**intermediate operations**) trên stream ban đầu để chuyển đổi nó thành một stream khác và tiếp tục thực hiện các hoạt động trung gian khác. Có thể có nhiều hoạt động trung gian.

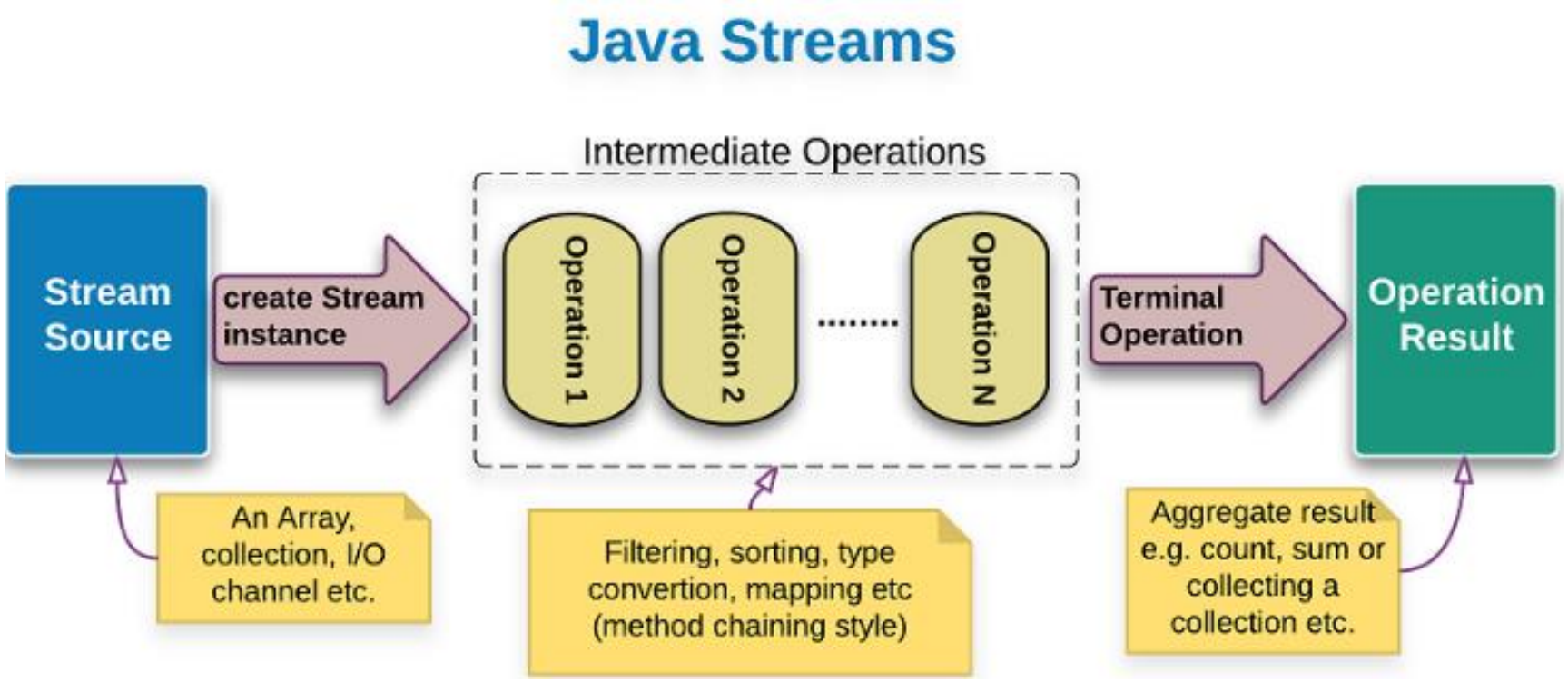
➢ Thực hiện thao tác đầu cuối (**terminal operation**) trên stream cuối cùng để nhận kết quả và sau đó không thể sử dụng lại chúng.



Stream API trong Java 8

Cách làm việc với Stream

- Một Stream pipeline bao gồm: 1 stream source, 0 hoặc nhiều intermediate operation, và 1 terminal operation.



Stream API trong Java 8

Ví dụ tạo Stream cho những kiểu primitive

- **Interface Stream** trong package **java.util.stream** là interface đại diện cho một Stream. Interface này chỉ làm việc với kiểu dữ liệu là **Object**.
- Với các kiểu primitive thì các bạn có thể sử dụng các đối tượng Stream dành cho những kiểu primitive đó, ví dụ như **IntStream**, **LongStream** hay **DoubleStream**.

```
public class PrimitiveStreamExample {

    public static void main(String[] args) {
        IntStream.range(1, 4).forEach(System.out::println); // 1 2 3

        IntStream.of(1, 2, 3).forEach(System.out::println); // 1 2 3

        DoubleStream.of(1, 2, 3).forEach(System.out::println); // 1.0 2.0 3.0

        LongStream.range(1, 4).forEach(System.out::println); // 1 2 3

        LongStream.of(1, 2, 3).forEach(System.out::println); // 1 2 3
    }
}
```

Stream API trong Java 8

Ví dụ tạo Stream từ các cấu trúc dữ liệu khác

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;
import java.util.stream.Stream;

public class ConvertToStreamExample {

    // Generate Streams from Arrays using .stream or Stream.of
    public static void streamFromArray() {

    // Generate Streams from Collections
    public static void streamFromCollection() {

    // Generate Streams using Stream.generate()
    public static void streamUsingGenerate() {

    // Generate Streams using Stream.iterate()
    public static void streamUsingIterate() {

    // Generate Streams from APIs like Regex
    public static void streamUsingRegex() {
}
```

```java
// Generate Streams from Arrays using .stream or Stream.of
public static void streamFromArray() {
    String[] languages = { "Java", "C#", "C++", "PHP", "Javascript" };

    // Get Stream using the Arrays.stream
    Stream<String> testStream1 = Arrays.stream(languages);
    testStream1.forEach(x -> System.out.println(x));

    // Get Stream using the Stream.of
    Stream<String> testStream2 = Stream.of(languages);
    testStream2.forEach(x -> System.out.println(x));
}
```

```java
// Generate Streams from Collections
public static void streamFromCollection() {
    List<String> items = new ArrayList<>();
    items.add("Java");
    items.add("C#");
    items.add("C++");
    items.add("PHP");
    items.add("Javascript");

    items.stream().forEach(item -> System.out.println(item));
}
```

Stream API trong Java 8

Ví dụ tạo Stream từ các cấu trúc dữ liệu khác

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;
import java.util.stream.Stream;

public class ConvertToStreamExample {

    // Generate Streams from Arrays using .stream or Stream.of
    public static void streamFromArray() {

    // Generate Streams from Collections
    public static void streamFromCollection() {

    // Generate Streams using Stream.generate()
    public static void streamUsingGenerate() {

    // Generate Streams using Stream.iterate()
    public static void streamUsingIterate() {

    // Generate Streams from APIs like Regex
    public static void streamUsingRegex() {
}
```

```java
// Generate Streams using Stream.generate()
public static void streamUsingGenerate() {
    Stream<String> stream = Stream.generate(() -> "javacoder").limit(3);
    String[] testStrArr = stream.toArray(String[]::new);
    System.out.println(Arrays.toString(testStrArr));
    // [javacoder, javacoder, javacoder]
}

// Generate Streams using Stream.iterate()
public static void streamUsingIterate() {
    Stream<Long> iterateNumbers = Stream.iterate(1L, n -> n + 1).limit(5);
    iterateNumbers.forEach(System.out::print); // 12345
}

// Generate Streams from APIs like Regex
public static void streamUsingRegex() {
    String str = "Welcome,to,javacoder";
    Pattern.compile(",").splitAsStream(str).forEach(System.out::print);
    // Welcometojavacoder
}
```

Stream API trong Java 8

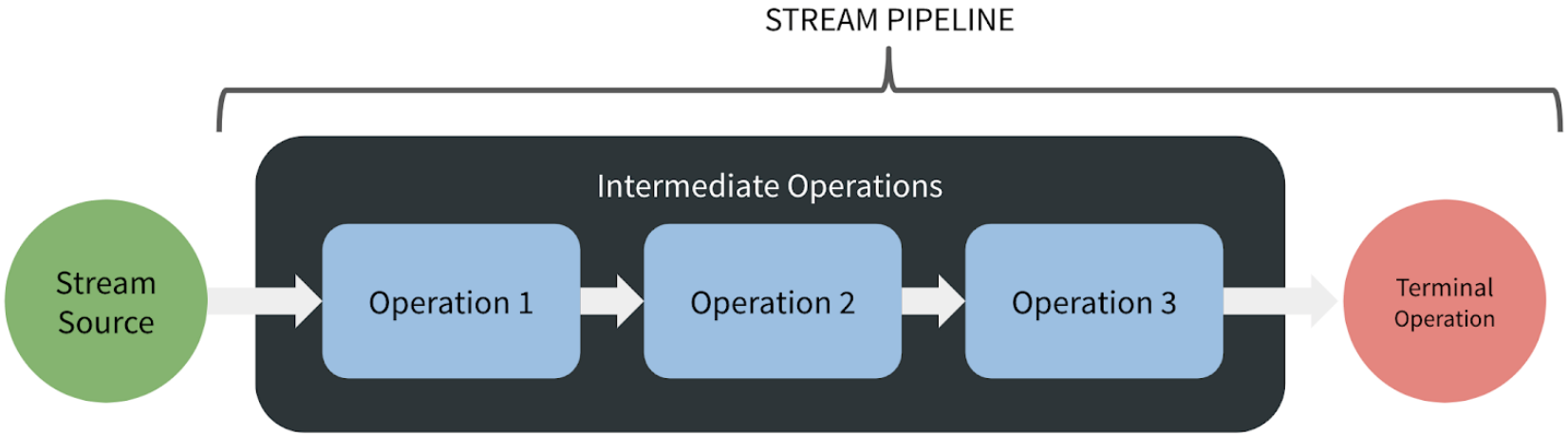
Ví dụ tạo Stream sang các cấu trúc dữ liệu khác

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ConvertFromStreamExample {

    // Get Collections using stream.collect(Collectors.toList())
    public static void getCollectionUsingStreamCollection() {
        Stream<String> stream = Stream.of("Java", "C#", "C++", "PHP", "Javascript");
        List<String> languages = stream.collect(Collectors.toList());
        System.out.println(languages);
    }

    // Get arrays using stream.toArray(EntryType[]::new)
    public static void getArrayUsingStreamToArray() {
        Stream<String> stream = Stream.of("Java", "C#", "C++", "PHP", "Javascript");
        String[] languages = stream.toArray(String[]::new);
        System.out.println(Arrays.toString(languages));
    }
}
```

Stream API trong Java 8

Ví dụ Intermediate Operations

- Có thể sử dụng **0** hoặc **nhiều intermediate operations** để chuyển đổi Stream ban đầu thành những Stream mới.
- Mặc dù chúng ta có thể định nghĩa nhiều **intermediate operation** nhưng chúng không thực thi các thao tác đó ngay lập tức, chỉ khi **terminal operation** được gọi thì toàn bộ các thao tác đó mới được thực thi.



Stream API trong Java 8

Ví dụ sử dụng filter()

- Stream **filter()** giúp loại bỏ các phần tử dựa trên các tiêu chí nhất định.
- Ví dụ: sử dụng filter() để lọc các số chia hết cho 3.

```
import java.util.stream.Stream;

public class FilterStreamExample {

    // filter() operation helps eliminate elements based on certain criteria
    public static void main(String[] args) {
        Stream.iterate(1, count -> count + 1) //
                .filter(number -> number % 3 == 0) //
                .limit(6) //
                .forEach(System.out::println);
    }

}
```

Stream API trong Java 8

Ví dụ sử dụng skip(), limit()

- Ý nghĩa của Stream **skip()**, **limit()** hoàn toàn tương tự với **OFSET** và **LIMIT** trong SQL.
- Stream **limit()** được sử dụng để loại bỏ các phần tử n đầu tiên của Stream . Nếu Stream này chứa ít hơn n phần tử thì luồng trống sẽ được trả lại.
- Stream **limit()** được sử dụng để cắt giảm kích thước của Stream. Kết quả trả về các phần tử của Stream đã được cắt giảm để không vượt quá **maxSize** (tham số đầu vào của phương thức).

```
import java.util.Arrays;
import java.util.List;

public class LimitStreamExample {

    // limit() Returns a stream consisting of the elements of this stream, truncated
    // to be no longer than maxSize in length.
    public static void main(String[] args) {
        List<String> data = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        data.stream() //
                .skip(1) //
                .limit(3) //
                .forEach(System.out::print); // C#C++PHP
    }
}
```

Stream API trong Java 8

Ví dụ sử dụng map()

- Stream **map()** giúp ánh xạ các phần tử tới các kết quả tương ứng.

```
public class MapStreamExample {

    // map() operation helps map elements to the corresponding results
    public static void main(String[] args) {
        List<String> data = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        data.stream() //
                .map(String::toUpperCase) // convert each element to upper case
                .forEach(System.out::println);
    }
}
```

Ví dụ sử dụng sorted()

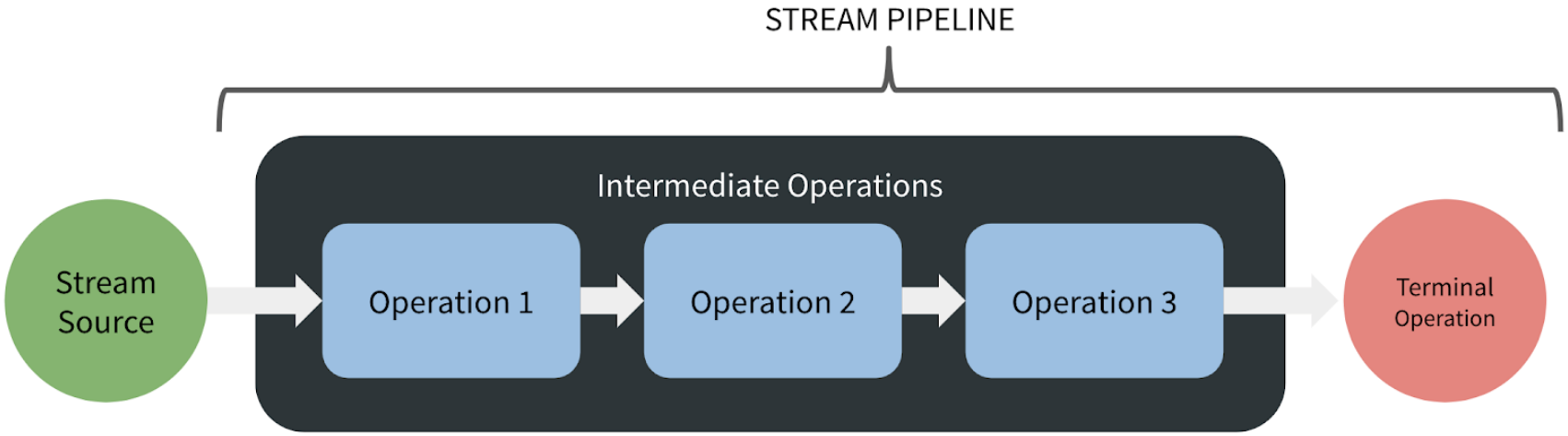
- Stream **sorted()** giúp sắp xếp các phần tử theo một thứ tự xác định.

```
public class SortedStreamExample {

    // sorted() operation helps sort elements based on certain criteria
    public static void main(String[] args) {
        List<String> data = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");

        // sorted according to natural order
        data.stream() //
                .sorted() //
                .forEach(System.out::println);

        // sorted according to the provided Comparator
        data.stream() //
                .sorted((s1, s2) -> s1.length() - s2.length()) //
                .forEach(System.out::println);
    }
}
```

Stream API trong Java 8

Ví dụ Terminal Operations

- Sử dụng **terminal operation** lấy về kết quả từ những **intermediate operations** đã được định nghĩa.
- Chúng ta có thể dễ dàng xác định đâu là **intermediate operation**, đâu là **terminal operation** bởi vì **terminal operation** sẽ trả về **void** hoặc **non-Stream** object.
- Sau khi **terminal operation** được gọi, **Stream** sẽ không sử dụng được nữa.



Stream API trong Java 8

Ví dụ sử dụng forEach(), collect()

- Phương thức **forEach()** giúp duyệt qua các phần tử của Stream.

```
public class ForEachStreamExample {

    // foreach operations helps iterate the elements of the Stream
    public static void main(String[] args) {
        Stream.iterate(1, count -> count + 1) //
                .filter(number -> number % 3 == 0) //
                .limit(6) //
                .forEach(System.out::println);
    }
}
```

- Phương thức **collect()** giúp thu thập kết quả Stream sang một Collection.

```
public class CollectStreamExample {

    // collect() operation helps to collect the stream result in a collection like
    // list
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("Java", "C#", "C++", "PHP", "Javascript");
        List<String> languages = stream.collect(Collectors.toList());
        System.out.println(languages);
    }
}
```

Stream API trong Java 8

Ví dụ sử dụng anyMatch(), allMatch(), noneMath()

- Phương thức **anyMatch()** trả về một Boolean tùy thuộc vào điều kiện được áp dụng trên Stream dữ liệu. Phương thức này trả về true ngay khi phần tử đầu tiên thỏa mãn điều kiện, những phần tử còn lại sẽ không được kiểm tra.

```
public class MatchStreamExample {

    // match() operation returns a boolean depending upon the condition applied on
    // stream data
    public static void main(String[] args) {
        List<String> data = Arrays.asList("Java", "C#", "C++", "PHP", "Javascript");
        boolean result = data.stream().anyMatch((s) -> s.equalsIgnoreCase("Java"));
        System.out.println(result); // true
    }
}
```

- **allMatch()**: Phương thức này trả về true nếu tất cả phần tử đều thỏa mãn điều kiện. Nếu một phần tử không thỏa mãn, những phần tử còn lại sẽ không được kiểm tra.
- **noneMatch()**: Ngược lại với **allMatch()**, phương thức này trả về true nếu tất cả phần tử đều không thỏa mãn điều kiện. Nếu một phần tử thỏa điều kiện, những phần tử còn lại sẽ không được kiểm tra.

Stream API trong Java 8

Ví dụ sử dụng count(), summaryStatistics()

- Phương thức **count()** trả về tổng số phần tử cho dữ liệu luồng.

```
public class CountStreamExample {

    // count() operation return the aggregate count for stream data
    public static void main(String[] args) {

        List<Integer> data = Arrays.asList(2, 3, 5, 4, 6);

        long count = data.stream().filter(num -> num % 3 == 0).count();
        System.out.println("Count = " + count);
    }
}
```

- Phương thức **summaryStatistics()** được sử dụng để lấy giá trị count, min, max, sum và average với tập dữ liệu số.

```
public class IntSummaryStatisticsExample {

    public static void main(String[] args) {
        List<Integer> primes = Arrays.asList(2, 3, 5, 7, 10);

        IntSummaryStatistics stats = primes.stream().mapToInt((x) -> x).summaryStatistics();
        System.out.println("Count: " + stats.getCount());
        System.out.println("Max: " + stats.getMax());
        System.out.println("Min: " + stats.getMin());
        System.out.println("Sum: " + stats.getSum());
        System.out.println("Average: " + stats.getAverage());
    }
}
```

Stream API trong Java 8

Ví dụ sử dụng min(), max() với các class Wrapper của kiểu dữ liệu nguyên thủy (primitive type)

- Phương thức **Stream.min()**, **Stream.max()** chấp nhận đối số là một **Comparator** sao cho các item trong stream có thể so sánh với nhau để tìm tối thiểu (**min**), tối đa (**max**).

```
public class MinMaxStreamExample01 {

    public static void main(String[] args) {
        Integer []numbers = {1, 8, 3, 4, 5, 7, 9, 6};

        // Find max, min with Array ===================

        // Max = 9
        Integer maxNumber = Stream.of(numbers)
                .max(Comparator.comparing(Integer::valueOf))
                .get();

        // Min = 1
        Integer minNumber = Stream.of(numbers)
                .min(Comparator.comparing(Integer::valueOf))
                .get();

        // Find max, min with Collection ====================
        List<Integer> listOfIntegers = Arrays.asList(numbers);

        // Max = 9
        Integer max = listOfIntegers.stream()
                .mapToInt(v -> v)
                .max()
                .getAsInt();

        // Min = 1
        Integer min = listOfIntegers.stream()
                .mapToInt(v -> v)
                .min()
                .getAsInt();
    }
}
```

Stream API trong Java 8

Ví dụ sử dụng min(), max() với các class Object

```java
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

class Programing {
    private String name;
    private int exp;

    public Programing(String name, int age) {
        super();
        this.name = name;
        this.exp = age;
    }

    public String getName() {
        return name;
    }

    public int getExp() {
        return exp;
    }
}
```

```java
public class MinMaxStreamExample02 {
    public static void main(String[] args) {

        List<Programing> students = Arrays.asList( //
                new Programing("Java", 5), //
                new Programing("PHP", 2), //
                new Programing("C#", 1) //
        );

        // Max = 5
        Programing maxByExp = students.stream()
                .max(Comparator.comparing(Programing::getExp))
                .get();

        // Min = 1
        Programing minByExp = students.stream()
                .min(Comparator.comparing(Programing::getExp))
                .get();
    }
}
```

Stream API trong Java 8

Stream API với I/O

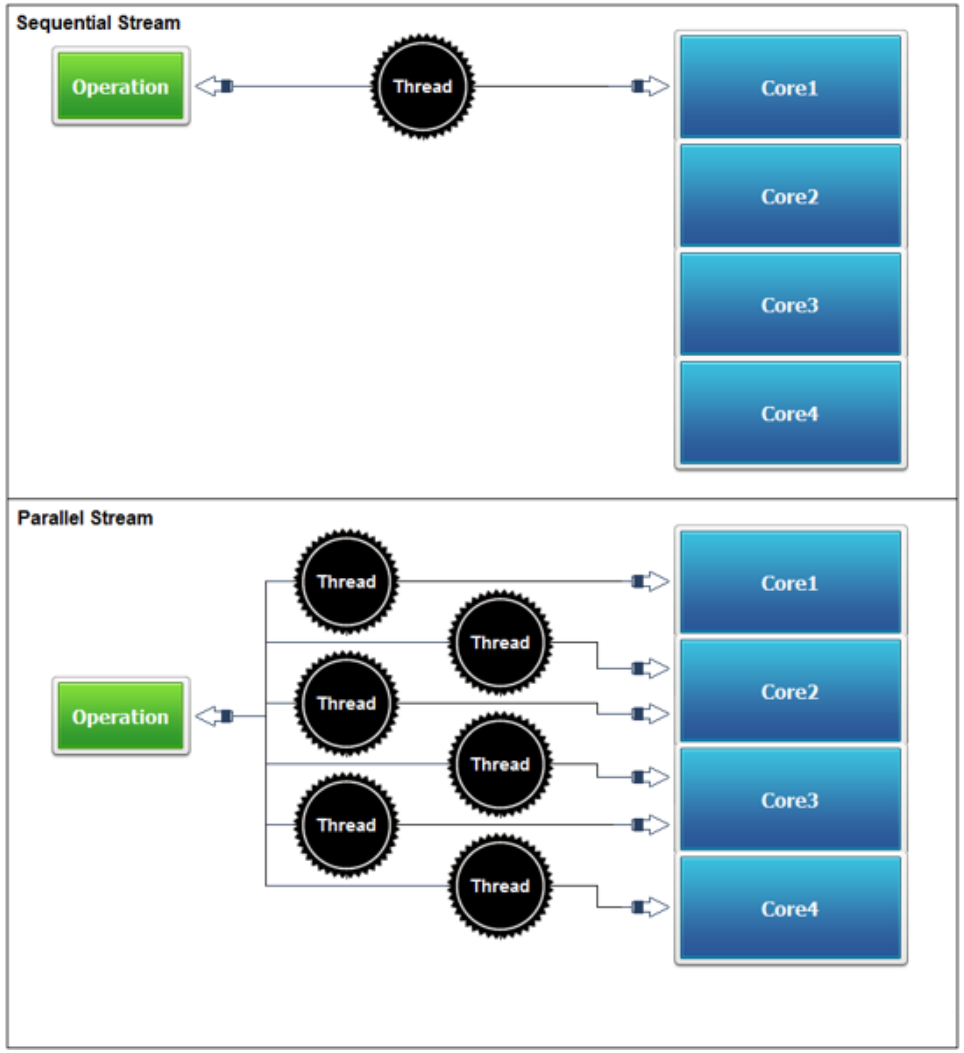
- **Stream API** không chỉ hỗ trợ các thao tác với các **Collection**, **Array** mà còn hỗ trợ các hoạt động I/O trên tập tin như đọc file văn bản.
- Phương thức **onClose()** trong đoạn code trên được gọi khi phương thức **close()** được gọi trên Stream.

```
public class ReadFileWithStreamExample {

    public static void main(String args[]) {

        String fileName = "[your-path]\\lines.txt";

        // read file into stream, try-with-resources
        try (Stream<String> stream = Files.lines(Paths.get(fileName))) {

            stream //
                    .onClose(() -> System.out.println("Done!")) //
                    .filter(s -> s.startsWith("line3")) //
                    .forEach(System.out::println);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Stream API trong Java 8

Luồng song song – Parallel Streams

- Các stream có thể là tuần tự (**sequential**) hoặc song song (**parallel**).
- Các thao tác trên các stream tuần tự được thực hiện trên một luồng đơn (**single thread**) trong khi các phép toán trên các stream song song được thực hiện đồng thời trên nhiều luồng (**multi-thread**).
- Chúng ta thường sử dụng **Parallel Streams** trong môi trường **multi-thread** khi mà chúng ta cần hiệu suất xử lý nhanh.



Stream API trong Java 8

Luồng song song – Parallel Streams

- Ví dụ **Sequential Stream** vs **Parallel Stream**

```java
public class SequentialStreamExample {

    public static void main(String[] args) {
        List<String> values = createDummyData();

        long startTime = System.nanoTime();

        long count = values.stream().sorted().count();
        System.out.println(count);

        long endTime = System.nanoTime();

        long millis = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);

        System.out.println(String.format("sequential sort took: %d ms", millis));
        // sequential sort took: 804 ms

    }

    public static List<String> createDummyData() {
        int max = 1000000;
        List<String> values = new ArrayList<>(max);
        for (int i = 0; i < max; i++) {
            UUID uuid = UUID.randomUUID();
            values.add(uuid.toString());
        }
        return values;
    }
}
```

```java
public class ParallelStreamExample {

    public static void main(String[] args) {
        List<String> values = createDummyData();

        long startTime = System.nanoTime();

        long count = values.parallelStream().sorted().count();
        System.out.println(count);

        long endTime = System.nanoTime();

        long millis = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);

        System.out.println(String.format("parallel sort took: %d ms", millis));
        // parallel sort took: 489 ms

    }

    public static List<String> createDummyData() {
        int max = 1000000;
        List<String> values = new ArrayList<>(max);
        for (int i = 0; i < max; i++) {
            UUID uuid = UUID.randomUUID();
            values.add(uuid.toString());
        }
        return values;
    }
}
```

- Như quan sát, 2 đoạn code gần như giống nhau **nhưng** sắp xếp song song nhanh hơn gần 50%. Chúng ta chỉ cần thay **stream()** thành **parallelStream()**

Stream API trong Java 8

Hạn chế của Streams

- **Stream** không thể tái sử dụng một khi đã gọi **Terminal Operations**

```
public class ErrorStreamExample {

    public static void main(String[] args) {
        Stream<String> stream = Stream.of("Java", "C#", "C++", "PHP", "Javascript")
                .filter(s -> s.startsWith("J"));

        stream.anyMatch(s -> true); // ok
        stream.noneMatch(s -> true); // exception
    }
}
```

Thực thi chương trình bên, ta nhận được thông báo lỗi như sau:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
        at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
        at java.base/java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:642)
        at com.vu.stream.reuse.ErrorStreamExample.main(ErrorStreamExample.java:12)
```

Để khắc phục hạn chế này, chúng ta phải tạo một Stream mới cho mọi hoạt động của thiết bị đầu cuối mà chúng ta muốn thực thi.

```
public class WithoutErrorStreamExample {

    public static void main(String[] args) {
        Supplier<Stream<String>> streamSupplier = //
                () -> Stream.of("Java", "C#", "C++", "PHP", "Javascript")
                        .filter(s -> s.startsWith("J"));

        streamSupplier.get().anyMatch(s -> true); // ok
        streamSupplier.get().noneMatch(s -> true); // ok
    }
}
```

Tổng kết nội dung bài học

- ❑ Lambda Expression Java 8
- ❑ Phương thức forEach()
- ❑ Stream API trong Java 8