LAMBDA VÀ STREAM

Nội dung trong chương

Lambda và Stream được giới thiệu cùng với các tính năng của Java 8 là các công cụ hữu ích giúp giảm số lượng dòng lệnh cần phải viết, đặc biệt là khi làm việc với dữ liệu.

Nội dung của chương sẽ đề cập đến:

- *Giao diện hàm (Functional Interface),*
- *Biểu thức Lambda (Lambda Expression)*
- *Phương thức tham chiếu (Method References)*
- *Tùy chọn (Optional)*
- *Giao diện lập trình Stream (Stream API)*

Giao diện hàm

Giao diện hàm (Functional interface) là một interface có duy nhất 1 abstract method, có thể không có hoặc có nhiều default/static method.

```
@FunctionalInterface
public interface Flyable {
    void fly();
    default boolean alive() .
        return true;
    }
}
```

Chú thích **@FunctionalInterface** là không bắt buộc, tuy nhiên nên có để tránh sai sót (ví dụ khi khai báo nhiều hơn một phương thức **abstract**). Functional interface còn được gọi là SAM type (**S**ingle **A**bstract **M**ethod).

Với một số ngôn ngữ lập trình các hàm có thể đưa vào làm tham số đầu vào cho một hàm khác, điều này tạo lên sự linh hoạt. Ví dụ:

```
function printResult(calculate) {
    console.log('Result is', calculate())
}
printResult(function () {
    return 3.14
})
```

Tuy nhiên Java không cho phép thực hiện điều này. Tham số của một phương thức trong Java chỉ có thể là các kiểu dữ liệu cơ sở, hoặc đối tượng. Với cách viết như trên (JavaScript), Java không thực hiện được. Phuwong thức trong Java không thể truyền vào một phương thức khác dưới dạng đối số. Do đó, nhóm phát triển ngôn ngữ Java đã nghĩ ra một cách tiếp cận khác để giải quyết vấn đề này. Thay vì truyền trực tiếp function B vào function A, thì phải "gói" function B vào một object (có dạng của functional interface). Sau đó mới truyền object đó vào cho function A. Khi đó, Function A chỉ cần lấy ra và sử dụng.

Ví dụ:

```java
// Định nghĩa khuôn mẫu cho phương thức truyền vào
@FunctionalInterface
interface Calculable {
    double calculate();
}

public class Program {
    public static void printResult(Calculable func) {
        System.out.println("Result: " + func.calculate());
    }
    public static void main(String[] args) {
        printResult(new Calculable() {
            @Override
            public double calculate() {
                return 3.14;
            }
        });
    }
}
```

Kết quả khi chạy:

```
Result: 3.14
```

Trong một giao diện hàm có thể khai báo phương thức mặc định và phương thức tĩnh, là phương thức có từ khóa **default** hoặc **static** và có phần thân hàm. Ví dụ:

```java
@FunctionalInterface
interface Sayable{
   // Phương thức mặc định
   default void say(){
       System.out.println("Hello, this is default method");
   }
   // Phương thức trừu tượng
   void sayMore(String msg);
   // Phương thức tĩnh
   static void sayLouder(String msg){
       System.out.println(msg);
   }
}
public class Program implements Sayable{
   public void sayMore(String msg){
       System.out.println(msg);
   }
   public static void main(String[] args) {
       Program dm = new Program();
       dm.say();
       dm.sayMore("Work is worship");
       Sayable.sayLouder("Helloooo...");
```

```
23      }
24  }
```

Biểu thức Lambda

Trước tiên ta cần tìm hiểu tại sao cần sử dụng biểu thức **Lambda**. Mục đích chính của việc sử dụng Lambda là nhằm rút gọn các câu lệnh cần phải viết. Cú pháp của biểu thức Lambda như sau:

```
(argument-list) -> {body}
```

Xét trường hợp không sử dụng Lambda:

```
1  interface Drawable {
2     void draw();
3  }
```

Khi đó để tạo một đối tượng từ giao diện Drawable có hai cách, cách thứ nhất là tạo ra lớp Circle triển khai từ giao diện Drawable

```
 1  class Circle implements Drawable {
 2     @Override
 3     public void draw() {
 4         System.out.println("Circle");
 5     }
 6  }
 7  public class Program {
 8     public static void main(String[] args) {
 9         Circle c = new Circle();
10         c.draw();
11     }
12  }
```

Cách thứ hai là tạo ra đối tượng Circle từ giao diện Drawable, khi đó sẽ cần triển khai hàm draw khi tạo đối tượng:

```
 1  interface Drawable {
 2      void draw();
 3  }
 4  public class Program {
 5     public static void main(String[] args) {
 6         Drawable circle = new Drawable() {
 7             @Override
 8             public void draw() {
 9                 System.out.println("Circle");
10             }
11         };
12         circle.draw();
13     }
14  }
```

Với việc sử dụng biểu thức Lambda mã lệnh có thể rút gọn như sau:

```java
@FunctionalInterface  //It is optional
interface Drawable {
    void draw();
}
public class Program {
    public static void main(String[] args) {
        Drawable circle = () -> System.out.println("Circle");
        circle.draw();
    }
}
```

Trường hợp hàm draw có một tham số, cú pháp để truyền tham số như sau:

```java
interface Drawable {
    void draw(int r);
}
public class Program {
    public static void main(String[] args) {
        Drawable circle = r -> {
            System.out.println("Circle " + r);
        };
        circle.draw(25);
    }
}
```

Trường hợp hàm draw có nhiều tham số, danh sách các tham số bắt buộc phải để trong dấu ngoặc

```java
interface Drawable {
    void draw(int r, String color);
}
public class Program {
    public static void main(String[] args) {
        Drawable circle = (r, color) -> {
            System.out.println("Circle " + r + ", " + color);
        };
        circle.draw(25, "Blue");
    }
}
```

Đối với trường hợp hàm có giá trị trả về, nếu trong phần thân hàm chỉ có một câu lệnh return, có thể bỏ qua luôn từ khóa return này. Ví dụ:

```java
interface Addable {
    int add(int a, int b);
}
public class Program {
    public static void main(String[] args) {
        Addable addable = (a, b) -> a + b;
        System.out.println(addable.add(2, 10));
    }
}
```

Trong trường hợp phần thân hàm có nhiều câu lệnh, bắt buộc phải có dấu ngoặc kép kèm theo lệnh return.

Xét một ví dụ khác, trong hàm foreach để thực hiện việc duyệt một danh sách, nếu không sử dụng lambda, trong phương thức forEach() cần tạo một đối tượng Consumer và viết câu lệnh cho phương thức accept như sau:

```
public static void main(String[] args) {
   List<String> list = new ArrayList<>();
   list.add("Apple");
   list.add("Banana");
   list.add("Orange");
   list.add("Grape");
   list.forEach(new Consumer<String>() {
                    @Override
                    public void accept(String s) {
                        System.out.println(s);
                    }
                }
   );
}
```

Tuy nhiên khi sử dụng Lambda, chỉ cần khai báo biểu thức Lambda như sau:

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Orange");
list.add("Grape");
list.forEach(s -> System.out.println(s));
```

Phương thức tham chiếu

Phương thức tham chiếu (Method reference) được sử dụng để tham chiếu đến phương thức khi sử dụng biểu thức lambda, điều này giúp viết biểu thức lambda được rút gọn hơn.

Tham chiếu đến phương thích tĩnh

Để tham chiếu đến một phương thức tĩnh, sử dụng cú pháp sau:

```
ContainingClass::staticMethodName
```

Ví dụ, trong trường hợp sau biểu thức lambda tham chiếu đến phương thức tĩnh saySomething() thuộc lớp Program:

```
@FunctionalInterface
interface Sayable {
    void say();
}

public class Program {
    public static void saySomething() {
```

```
        System.out.println("Hello, this is static method.");
    }

    public static void main(String[] args) {
        Sayable sayable = () -> Program.saySomething();
        sayable.say();
    }
}
```

Tại đây biểu thức lambda có thể sử đến phương thức tham chiếu tĩnh để viết rút gọn lại như sau:

```
@FunctionalInterface  //It is optional
interface Sayable {
   void say();
}
public class Program {
   public static void saySomething(){
       System.out.println("Hello, this is static method.");
   }
   public static void main(String[] args) {
       Sayable sayable = Program::saySomething;
       sayable.say();
   }
}
```

Xét ví dụ khác, sử dụng phương thức tham chiếu để tạo hai tiểu trình thread1 và thread2 thực hiện công việc được viết trong hai phương thức tĩnh runMethod1 và runMethod2:

```
public class Program {
   public static void runMethod1() {
       System.out.println("Thread is running... doing 123");
   }
   public static void runMethod2() {
       System.out.println("Thread is running... doing abc");
   }
   public static void main(String[] args) {
       Thread thread1 = new Thread(Program::runMethod1);
       thread1.start();
       Thread thread2 = new Thread(Program::runMethod2);
       thread2.start();
   }
}
```

Tham chiếu tới một phương thức thường

Giống như phương thức tĩnh, có thể tham chiếu đến một phương thức bình thường bằng cách sử dụng cú pháp sau:

```
containingObject::instanceMethodName
```

Ví dụ sau tham chiếu đến phương thức saySomething thuộc lớp Program:

```java
interface Sayable {
   void say();
}
public class Program {
   public void saySomething() {
       System.out.println("Hello, this is non-static method.");
   }
   public static void main(String[] args) {
       // Referring non-static method using reference
       Sayable sayable = new Program()::saySomething;
       // Calling interface method
       sayable.say();
   }
}
```

Ví dụ tương tự với việc tạo tiểu trình để thực hiện công việc printMsg() được khai báo trong lớp Program:

```java
public class Program {
   public void printnMsg() {
       System.out.println("Hello, this is instance method");
   }
   public static void main(String[] args) {
       Thread t = new Thread(new Program()::printnMsg);
       t.start();
   }
}
```

Tham chiếu tới một hàm tạo

Để tham chiếu đến hàm tạo, sử dụng cú pháp đơn giản sau, trong đó chỉ cần một từ khóa new và không cần viết tên hàm tạo:

```
ClassName::new
```

Ví dụ khi tạo ra đối tượng hello, thay vì viết biểu thức lambda: msg -> new Message(msg) thì có thể viết ngắn gọn hơn như sau:

```java
interface Messageable {
   Message getMessage(String msg);
}
class Message {
   Message(String msg) {
       System.out.print(msg);
   }
}
public class Program {
   public static void main(String[] args) {
       Messageable hello = Message::new;
       hello.getMessage("Hello");
```

```
13      }
14  }
```

Các giao diện hàm đã được định nghĩa

Java định nghĩa nhiều **Giao diện hàm** (Functional Interface) để có thể sử dụng trong nhiều trường hợp, dưới đây là danh sách các giao diện hàm đã được định nghĩa sẵn. Chức năng có thể được chia thành năm nhóm:

- **Function** chấp nhận đối số và đưa ra kết quả;
- **Operator** tạo ra kết quả cùng kiểu với đối số của chúng (một trường hợp đặc biệt của hàm);
- **Predicate** chấp nhận đối số và trả về giá trị boolean (hàm có giá trị boolean);
- **Supplier** (cung cấp) không nhận nhận đối số và trả lại giá trị;
- **Consumer** (tiêu dùng) nhận các đối số và không trả lại kết quả.

Nhờ các quy ước đặt tên của các giao diện, có thể dễ dàng hiểu một đặc điểm giao diện chỉ bằng cách nhìn vào tiền tố tên của nó, ví dụ:

- Tiền tố **Bi** chỉ ra rằng một hàm, một Predicate hoặc một Consumer chấp nhận hai tham số. Tương tự với Bi tiền tố, tiền tố Unary và Binary chỉ ra rằng một toán tử chấp nhận một và hai tham số tương ứng.
- Tiền tố **Double, Long, Int,** và **Obj** cho biết loại giá trị đầu vào. Ví dụ, giao diện IntPredicate đại diện cho một Predicate chấp nhận giá trị của một kiểu Integer.
- Tiền tố **ToDouble, ToLong** và **ToInt** cho biết loại giá trị đầu ra. Ví dụ, giao diện ToIntFunction<T> đại diện cho một hàm trả về giá trị của một kiểu Integer.

Các giao diện này đều được định nghĩa trong java.util.function:

Giao diện hàm	Mô tả
BiConsumer<T,U>	Đại diện cho một hoạt động chấp nhận hai đối số đầu vào và không trả về kết quả nào.
Consumer<T>	Đại diện cho một phép toán chấp nhận một đối số duy nhất và không trả về kết quả nào.
Function<T,R>	Đại diện cho một hàm chấp nhận một đối số và trả về một kết quả.
Predicate<T>	Đại diện cho một hàm chấp nhận một đối số và trả về kết quả đúng hoặc sai.
BiFunction<T,U,R>	Đại diện cho một hàm chấp nhận hai đối số và trả về một kết quả.
BinaryOperator<T>	Đại diện cho một phép toán dựa trên hai toán hạng của cùng một kiểu dữ liệu. Nó trả về một kết quả cùng kiểu với các toán hạng.

Giao diện hàm	Mô tả
BiPredicate<T,U>	Đại diện cho một hàm chấp nhận hai đối số và trả về kết quả đúng hoặc sai.
BooleanSupplier	Đại diện cho một hàm cung cấp các kết quả có giá trị boolean.
DoubleBinaryOperator	Đại diện cho một phép toán dựa trên hai toán hạng kiểu Double và trả về một giá trị kiểu Double.
DoubleConsumer	Đại diện cho một phép toán chấp nhận một đối số kiểu Double duy nhất và không trả về kết quả nào.
DoubleFunction<R>	Đại diện cho một hàm chấp nhận một đối số kiểu Double và tạo ra một kết quả.
DoublePredicate	Đại diện cho một vị từ (hàm có giá trị boolean) của một đối số kiểu Double.
DoubleSupplier	Đại diện cho một nhà cung cấp kết quả loại Double.
DoubleToIntFunction	Đại diện cho một hàm chấp nhận đối số kiểu Double và tạo ra kết quả kiểu int.
DoubleToLongFunction	Đại diện cho một hàm chấp nhận đối số kiểu Double và tạo ra kết quả kiểu Long.
DoubleUnaryOperator	Đại diện cho một phép toán trên một toán hạng kiểu Double trả về kết quả kiểu Double.
IntBinaryOperator	Đại diện cho một phép toán dựa trên hai toán hạng kiểu Int và trả về một kết quả kiểu Int.
IntConsumer	Đại diện cho một phép toán chấp nhận một đối số số nguyên duy nhất và không trả về kết quả nào.
IntFunction<R>	Đại diện cho một hàm chấp nhận một đối số là số nguyên và trả về một kết quả.
IntPredicate	Đại diện cho một vị từ (hàm có giá trị boolean) của một đối số nguyên.
IntSupplier	Đại diện cho một nhà cung cấp kiểu số nguyên.
IntToDoubleFunction	Đại diện cho một hàm chấp nhận một đối số là số nguyên và trả về một giá trị Double.
IntToLongFunction	Đại diện cho một hàm chấp nhận một đối số là số nguyên và trả về một giá trị dài.
IntUnaryOperator	Đại diện cho một hoạt động trên một toán hạng số nguyên duy nhất tạo

Giao diện hàm	Mô tả
	ra một kết quả số nguyên.
LongBinaryOperator	Đại diện cho một phép toán dựa trên hai toán hạng kiểu Long và trả về một kết quả kiểu Long.
LongConsumer	Đại diện cho một phép toán chấp nhận một đối số kiểu Long duy nhất và không trả về kết quả nào.
LongFunction<R>	Đại diện cho một hàm chấp nhận một đối số kiểu Long và trả về một kết quả.
LongPredicate	Đại diện cho một vị từ (hàm có giá trị boolean) của một đối số kiểu Long.
LongSupplier	Đại diện cho một nhà cung cấp các kết quả loại Long.
LongToDoubleFunction	Đại diện cho một hàm chấp nhận đối số kiểu Long và trả về kết quả kiểu Double.
LongToIntFunction	Đại diện cho một hàm chấp nhận một đối số kiểu Long và trả về một kết quả là số nguyên.
LongUnaryOperator	Đại diện cho một phép toán trên một toán hạng kiểu Long duy nhất trả về kết quả kiểu Long.
ObjDoubleConsumer<T>	Đại diện cho một hoạt động chấp nhận một đối tượng và một đối số Double, và không trả về kết quả nào.
ObjIntConsumer<T>	Đại diện cho một hoạt động chấp nhận một đối tượng và một đối số số nguyên. Nó không trả về kết quả.
ObjLongConsumer<T>	Đại diện cho một hoạt động chấp nhận một đối tượng và một đối số Long, nó không trả về kết quả nào.
Supplier<T>	Đại diện cho một nhà cung cấp kết quả.
ToDoubleBiFunction<T,U>	Đại diện cho một hàm chấp nhận hai đối số và tạo ra một kết quả kiểu Double.
ToDoubleFunction<T>	Đại diện cho một hàm trả về kết quả kiểu Double.
ToIntBiFunction<T,U>	Đại diện cho một hàm chấp nhận hai đối số và trả về một số nguyên.
ToIntFunction<T>	Đại diện cho một hàm trả về một số nguyên.
ToLongBiFunction<T,U>	Đại diện cho một hàm chấp nhận hai đối số và trả về một kết quả kiểu Long.
ToLongFunction<T>	Đại diện cho một hàm trả về kết quả kiểu Long.

Giao diện hàm	Mô tả
UnaryOperator<T>	Đại diện cho một phép toán trên một toán hạng duy nhất trả về kết quả cùng kiểu với toán hạng của nó.

Loại giao diện Function

Function nhận một giá trị làm tham số và trả về một giá trị duy nhất. Ví dụ, Function<T, R> là một giao diện chung đại diện cho một hàm chấp nhận một giá trị kiểu T và tạo ra một kết quả thuộc loại R. Ví dụ:

```
Function<String, Integer> converter = Integer::parseInt;
converter.apply("1000"); // the result is 1000 (Integer)
ToIntFunction<String> anotherConverter = Integer::parseInt;
anotherConverter.applyAsInt("2000"); // the result is 2000 (int)
BiFunction<Integer, Integer, Integer> sumFunction = (a, b) -> a + b;
sumFunction.apply(2, 3); //5 (Integer)
```

Loại giao diện Operator

Operator nhận và trả về các giá trị cùng kiểu. Ví dụ, UnaryOperator<T> đại diện cho một Operators chấp nhận một giá trị kiểu T và tạo ra một kết quả cùng loại T. . Ví dụ:

```
UnaryOperator<Long> longMultiplier = val -> 100_000 * val;
longMultiplier.apply(2L); // the result is 200_000L (Long)
IntUnaryOperator intMultiplier = val -> 100 * val;
intMultiplier.applyAsInt(10); // 1000 (int)
BinaryOperator<String> appender = (str1, str2) -> str1 + str2;
appender.apply("str1", "str2"); // "str1str2"
```

Loại giao diện Predicate

Predicate nhận một giá trị làm tham số và trả về true hoặc false. Ví dụ, Predicate<T> là một giao diện chung đại diện cho một Predicate chấp nhận một giá trị kiểu T và tạo ra một kết quả có giá trị boolean. Ví dụ:

```
Predicate<Character> isDigit = Character::isDigit;
isDigit.test('h'); // false (boolean)
IntPredicate isEven = val -> val % 2 == 0;
isEven.test(10); // true (boolean)
```

Loại giao diện Supplier

Mỗi **Supplier** không chấp nhận tham số và trả về một giá trị duy nhất. Ví dụ, Supplier<T> đại diện cho một nhà cung cấp không chấp nhận đối số và trả về một giá trị kiểu T. Ví dụ:

```
Supplier<String> stringSupplier = () -> "Hello";
stringSupplier.get(); // the result is "Hello" (String)
BooleanSupplier booleanSupplier = () -> true;
```

```
booleanSupplier.getAsBoolean(); // Kết quả là true (boolean)
IntSupplier intSupplier = () -> 33;
intSupplier.getAsInt(); // 33 (int)
```

Loại giao diện Consumer

Consumer chấp nhận hai đối số đầu vào và không trả về bất kỳ kết quả nào. Đây hai đối tượng đại diện. Nó cung cấp một phương thức chức năng accept(Object, Object) để thực hiện các thao tác tùy chỉnh. Xem xét đoạn mã nguồn sau:

```
import java.util.function.BiConsumer;
public class Program {
    static void ShowDetails(String name, Integer age){
        System.out.println(name+" "+age);
    }
    public static void main(String[] args) {
        // Referring method
        BiConsumer<String, Integer> biCon = Program::ShowDetails;
        biCon.accept("Rama", 20);
        biCon.accept("Shyam", 25);
        // Using lambda expression
        BiConsumer<String, Integer> biCon2 = (name, age)->System.out.println(name+"
"+age);
        biCon2.accept("Peter", 28);
    }
}
```

Tuỳ chọn (Optional)

Giống như nhiều ngôn ngữ lập trình, Java sử dụng **null** đại diện cho sự vắng mặt của một giá trị. Đôi khi cách tiếp cận này dẫn đến các trường hợp ngoại lệ **NullPointerException** trong khi các kiểm tra khác null làm cho mã trở lên rất khó đọc. Nhà khoa học máy tính người Anh Tony Hoare — người phát minh ra khái niệm null— thậm chí mô tả việc giới thiệu null như một "sai lầm hàng tỷ đô la" vì nó đã dẫn đến vô số lỗi, lỗ hổng bảo mật và sự cố hệ thống. Để tránh các vấn đề liên quan đến null, Java cung cấp lớp Optional, đó là một giải pháp thay thế an toàn hơn cho tiêu chuẩn null.

Optional đại diện cho sự hiện diện hoặc vắng mặt của một giá trị của kiểu được chỉ định T.

Xét ví dụ sau:

```
public static void main(String[] args) {
    String str = "Hello";
    // Tạo một Optional từ một giá trị có thể null
    Optional<String> optionalStr = Optional.ofNullable(str);
    // Kiểm tra xem Optional có chứa giá trị hay không và truy cập nếu có
    if (optionalStr.isPresent()) {
        System.out.println("Giá trị của chuỗi là: " + optionalStr.get());
    } else {
        System.out.println("Chuỗi là null");
    }
    // Sử dụng phương thức orElse để cung cấp một giá trị mặc định nếu không có giá
trị
```

```
12      String defaultValue = optionalStr.orElse("Giá trị mặc định");
13      System.out.println("Giá trị chuỗi (sử dụng orElse): " + defaultValue);
14      // Sử dụng phương thức ifPresent để xử lý nếu giá trị tồn tại
15      optionalStr.ifPresent(s -> System.out.println("Giá trị chuỗi (nếu tồn tại): " +
s));
16  }
```

Kết quả khi chạy:

```
1  Giá trị của chuỗi là: Hello
2  Giá trị chuỗi (sử dụng orElse): Hello
3  Giá trị chuỗi (nếu tồn tại): Hello
```

Xét thêm một số phương thức quan trọng kh sử dụng Optional:

get() trả về giá trị nếu nó hiện diện, nếu không sẽ ném một ngoại lệ;

orElse() trả về giá trị nếu nó hiện diện, nếu không thì trả về other;

orElseGet() trả về giá trị nếu nó hiện diện, nếu không sẽ gọi other và trả về kết quả của nó.

Hãy xem chúng hoạt động như thế nào. Đầu tiên, sử dụng phương thức **get()** để có được giá trị hiện tại:

```
1  Optional<String> optName = Optional.of("John");
2  String name = optName.get(); // "John"
```

Phương thức **orElse()** được sử dụng để trích xuất giá trị được bao bọc bên trong một đối tượng Optional hoặc trả về một số giá trị mặc định khi Optional trống. Giá trị mặc định được chuyển cho phương thức dưới dạng đối số của nó:

```
1  String nullableName = null;
2  String name = Optional.ofNullable(nullableName).orElse("unknown");
3  System.out.println(name); // unknown
```

Phương thức **orElseGet** tương tự, nhưng nó cần một hàm Supplier để tạo ra một kết quả thay vì lấy một số giá trị để trả về:

```
1  String name = Optional
2        .ofNullable(nullableName)
3        .orElseGet(SomeClass::getDefaultResult);
```

Ngoài ra còn có các phương thức thuận tiện lấy các hàm làm đối số và thực hiện một số hành động trên các giá trị được bao bọc bên trong Optional:

- **ifPresent** thực hiện hành động đã cho với giá trị, nếu không thì không làm gì cả;
- **ifPresentOrElse** thực hiện hành động đã cho với giá trị, nếu không thì thực hiện hành động dựa trên sản phẩm trống đã cho.

ifPresent cho phép chạy một số mã trên giá trị nếu Optional không trống. Phương thức lấy một **Consumer** có thể xử lý giá trị.

Ví dụ sau in độ dài của tên công ty bằng cách sử dụng **ifPresent**()

```
Optional<String> companyName = Optional.of("Google");
companyName.ifPresent((name) -> System.out.println(name.length())); // 6
```

Tuy nhiên, đoạn mã sau không in được gì vì đối tượng Optional trống.

```
Optional<String> noName = Optional.empty();
noName.ifPresent((name) -> System.out.println(name.length()));
```

Phương thức **ifPresentOrElse** là một sự thay thế an toàn hơn cho toàn bộ tuyên bố if-else. Nó thực thi một trong hai chức năng tùy thuộc vào việc giá trị có trong Optional.

```
Optional<String> optName = Optional.ofNullable();
optName.ifPresentOrElse(
            (name) -> System.out.println(name.length()),
            () -> System.out.println(0)
);
```

Xét ví dụ dưới đây, đây là trường hợp không sử dụng Optional, khiến cho chương trình phát sinh một ngoại lệ.

```
public static void main(String[] args) throws IOException {
   String[] str = new String[10];
   String lowercaseString = str[5].toLowerCase();
   System.out.print(lowercaseString);
}
```

Kết quả khi chạy

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.toLowerCase()" because "str[5]" is null
      at Program.main(Program.java:11)
```

Vấn đề trên có thể được giải quyết bằng việc sử dụng biểu thức điều kiện kết hợp với Optional để kiểm tra:

```
public static void main(String[] args) throws IOException {
   String[] str = new String[10];
   Optional<String> checkNull = Optional.ofNullable(str[5]);
   if(checkNull.isPresent()){  // check for value is present or not
       String lowercaseString = str[5].toLowerCase();
       System.out.print(lowercaseString);
   }else
       System.out.println("Giá trị không tồn tại");
}
```

Kết quả:

```
Giá trị không tồn tại
```

Sửa lại hàm main như sau:

```
public static void main(String[] args) throws IOException {
   String[] str = new String[10];
   str[5] = "Ví dụ về Optional";
   Optional<String> checkNull = Optional.ofNullable(str[5]);
   if(checkNull.isPresent()){
       String lowercaseString = str[5].toLowerCase();
       System.out.print(lowercaseString);
   }else
       System.out.println("Giá trị không tồn tại");
}
```

Kết quả khi chạy:

```
ví dụ về optional
```

Một ví dụ minh họa khác:

```
String[] str = new String[10];
str[5] = "Ví dụ về Optional";
Optional<String> empty = Optional.empty();
System.out.println(empty);
Optional<String> value = Optional.of(str[5]);
System.out.println("Kết quả: "+value.filter((s)->s.equals("Abc")));
System.out.println("Kết quả: "+value.filter((s)->s.equals("VÍ DỤ VỀ OPTIONAL")));
System.out.println("Kết quả: "+value.get());
System.out.println("Giá trị: "+value.hashCode());
System.out.println("Kết quả: "+value.isPresent());
System.out.println("Kết quả: "+Optional.ofNullable(str[5]));
System.out.println("Kết quả: "+value.orElse("Value is not present"));
System.out.println("Kết quả: "+empty.orElse("Value is not present"));
value.ifPresent(System.out::println);
```

Ví dụ sau sẽ minh họa cho việc sử dụng Optional thông qua một số các đoạn mã để minh họa cho việc xử lý dữ liệu văn bản. Bắt đầu bằng việc tạo ra luồng bằng cách đọc dữ liệu từ file văn bản:

```
public class OptionalTest {
   public static void main(String[] args) throws IOException {
       var contents = Files.readString(Paths.get("alice30.txt"));
       List<String> wordList = List.of(contents.split("\\PL+"));
   }
}
```

Tìm kiếm các từ dài (các từ có độ dài lớn hơn 12 ký tự):

```
Optional<String> optionalValue = wordList.stream().filter(s -> s.length()>12).findFirst();
System.out.println(optionalValue.orElse("No word"));
```

Sử dụng phương thức **orElseGet()** trong việc rẽ nhánh:

```
Optional<String> optionalValue = wordList.stream().filter(s -> s.length()>22).findFirst();
System.out.println(optionalValue.orElseGet(() -> {
    String s = "Very long word";
    return s;
}));
```

Sử dụng **ifPresent()** để kiểm tra sự tồn tại thay cho việc dùng cú pháp if thông thường:

```
Optional<String> optionalValue = wordList.stream().filter(s -> s.length()>12).findFirst();
LinkedList<String> results = new LinkedList<String>();
optionalValue.ifPresent(v->results.add(v));
System.out.println(results.getFirst());
```

Sử dụng phương thức tham chiếu:

```
var results = new HashSet<String>();
optionalValue.ifPresent(results::add);
```

Chuyển sang map:

```
Optional<String> optionalValue = wordList.stream().filter(s -> s.length()>12).findFirst();
Optional<String> transformed = optionalValue.map(String::toUpperCase);
System.out.println(optionalValue.orElse("No word"));
```

Stream API

Java 8 Stream

Java cung cấp một gói bổ sung mới trong **Java 8** được gọi là **java.util.stream**. Gói này bao gồm các lớp, giao diện và enum để cho phép các thao tác trên các phần tử dữ liệu. Stream cung cấp các đặc tính:

- Stream không lưu trữ các phần tử. Nó chỉ truyền tải các phần tử từ một nguồn chẳng hạn như cấu trúc dữ liệu, mảng...
- Các hoạt động được thực hiện trên một dòng không sửa đổi nguồn của nó.
- Stream lười biếng và chỉ đánh giá mã khi được yêu cầu.
- Các phần tử của Stream chỉ được truy cập một lần trong suốt vòng đời.

Stream có thể sử dụng để lọc, thu thập, in và chuyển đổi từ cấu trúc dữ liệu này sang cấu trúc dữ liệu khác, v.v. Bằng cách sử dụng Stream API, một lập trình viên không cần phải viết các vòng lặp rõ ràng, vì mỗi Stream có một vòng lặp được tối ưu hóa bên trong.

Nội dung ở trên có để cập tới việc xử lý tuần tự đối tượng bằng cách sử dụng vòng lặp và collections. Tuy nhiên, cách làm này khá dễ xảy ra lỗi do các biến có thể thay đổi và logic lặp phức tạp. Nó cũng có thể làm cho mã khó đọc hơn, dẫn đến phức tạp khi phát triển thêm. Một động lực chính đối với cách tiếp

cận mới là làm cho trình biên dịch Java có thể "song song hóa" một phép tính, nghĩa là chia nó thành nhiều phần có thể chạy đồng thời trên một số bộ vi xử lý.
Theo một nghĩa nào đó, một Stream (dòng) gợi nhớ một collection. Nhưng nó không thực sự lưu trữ các phần tử. Thay vào đó, nó truyền tải các phần tử từ một nguồn như bộ sưu tập, hàm trình tạo (generator function), tệp, kênh I/O, stream khác, sau đó xử lý các phần tử bằng cách sử dụng một chuỗi các hoạt động được xác định trước (predefined operations), được kết hợp thành một đường ống (pipeline) duy nhất. Điều quan trọng là một stream luôn chỉ có một thao tác đầu cuối duy nhất và số lượng thao tác trung gian tùy ý.

Có ba giai đoạn làm việc với một Stream:

- Lấy Stream từ một nguồn.
- Thực hiện các thao tác trung gian với Stream để xử lý dữ liệu.
- Thực hiện thao tác cuối để tạo ra kết quả.

So sánh vòng lặp với Stream

Hãy xem xét một ví dụ đơn giản. Giả sử chúng ta có một danh sách các số và chúng ta muốn đếm các số lớn hơn 5:

```
List<Integer> numbers = List.of(1, 4, 7, 6, 2, 9, 7, 8);
```

Một cách "truyền thống" để làm điều đó là viết một vòng lặp như sau:

```
long count = 0;
for (int number : numbers) {
    if (number > 5) {
            count++;
    }
}
System.out.println(count); // 5
```

Mã này in ra "5" vì danh sách ban đầu chỉ chứa năm số lớn hơn 5 (7, 6, 9, 7, 8). Một vòng lặp với một điều kiện lọc là một cấu trúc thường được sử dụng trong lập trình. Có thể đơn giản hóa mã này bằng cách viết lại nó bằng một stream:

```
long count = numbers.stream()
            .filter(number -> number > 5)
            .count(); // 5
```

Ở đây, chúng ta nhận được một Stream từ danh sách numbers, sau đó lọc các phần tử của nó bằng cách sử dụng biểu thức lambda Predicate và sau đó đếm các số thỏa mãn điều kiện. Mặc dù mã này tạo ra cùng một kết quả, nhưng nó dễ đọc và sửa đổi hơn. Ví dụ, chúng ta có thể dễ dàng thay đổi nó để bỏ qua bốn số đầu tiên khỏi danh sách.

```
long count = numbers.stream()
            .skip(4)  // skip 1, 4, 7, 6
```

```
3               .filter(number -> number > 5)
4               .count();  // 3
```

Quá trình xử lý được thực hiện như một chuỗi các lệnh gọi phương thức được phân tách bằng dấu chấm với một thao tác đầu cuối duy nhất. Để cải thiện khả năng đọc, nên đặt mỗi cuộc gọi vào một dòng mới nếu dòng chứa nhiều hơn một thao tác.

Tạo Stream

Có rất nhiều cách để tạo Stream bao gồm sử dụng list, set, string, array v.v. làm nguồn.

Cách phổ biến nhất để tạo Stream từ một Collection. Bất kỳ bộ sưu tập nào cũng có stream() cho mục đích này.

```
1  List<Integer> famousNumbers = List.of(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55);
2  Stream<Integer> numbersStream = famousNumbers.stream();
3  Set<String> usefulConcepts = Set.of("functions", "lazy", "immutability");
4  Stream<String> conceptsStream = usefulConcepts.stream();
```

Cũng có thể tạo Stream từ một mảng:

```
Stream<Double> doubleStream = Arrays.stream(new Double[]{ 1.01, 1d, 0.99, 1.02, 1d, 0.9
9 });
```

Tạo Stream trực tiếp từ một số giá trị:

```
Stream<String> persons = Stream.of("John", "Demetra", "Cleopatra");
```

Tạo Stream bằng cách nối các Stream khác với nhau:

```
1  Stream<String> stream1 = Stream.of(/* some values */);
2  Stream<String> stream2 = Stream.of(/* some values */);
3  Stream<String> resultStream = Stream.concat(stream1, stream2);
```

Có một số khả năng tạo các Stream trống (có thể được sử dụng làm giá trị trả về từ các phương thức):

```
1  Stream<Integer> empty1 = Stream.of();
2  Stream<Integer> empty2 = Stream.empty();
```

Ngoài ra còn có các phương pháp khác để tạo Stream từ các nguồn khác nhau: từ tệp, từ luồng I/O, v.v.

Các thao tác trên Stream

Tất cả các hoạt động Stream được chia thành hai nhóm: hoạt động trung gian(intermediate) và hoạt động đầu cuối(terminal).

- **Các hoạt động trung gian** không được đánh giá ngay lập tức khi gọi. Chúng chỉ đơn giản là trả về các Stream mới để gọi các thao tác tiếp theo trên chúng. Các hoạt động như vậy được gọi là lười biếng bởi vì chúng không thực sự làm bất cứ điều gì hữu ích.

- **Các hoạt động đầu cuối** bắt đầu tất cả các đánh giá với Stream để tạo ra kết quả hoặc tạo ra hiệu ứng phụ. Như đã đề cập trước đây, một Stream luôn chỉ có một thao tác đầu cuối.

Khi thao tác đầu cuối đã được đánh giá, không thể sử dụng lại Stream một lần nữa. Nếu bạn cố gắng làm điều đó, chương trình sẽ ném IllegalStateException. Stream cung cấp một số lượng lớn các hoạt động được thực hiện trên các phần tử. Cách duy nhất để học tất cả chúng là thực hành. Đó là lý do tại sao chúng tôi chỉ cung cấp cho bạn một danh sách nhỏ các hoạt động cho mỗi nhóm ở đây.

Các phương thức với hoạt động trung gian:

- **filter** trả về một Stream mới bao gồm các phần tử khớp với một predicate;
- **limit** trả về một Stream mới bao gồm n phần tử đầu tiên Stream;
- **skip** trả về một Stream mới mà không có n phần tử đầu tiên của Stream;
- **distinct** trả về một Stream mới chỉ bao gồm các phần tử duy nhất theo kết quả của equals;
- **sorted** trả về một Stream mới bao gồm các phần tử được sắp xếp theo thứ tự tự nhiên hoặc comparator được truyền;
- **map** trả về một Stream mới bao gồm các phần tử thu được bằng cách áp dụng một hàm(tức là chuyển đổi từng phần tử).

Ví dụ minh hoạ các phương thức trên:

```
 1  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
 2  // Filter: Lọc các số chẵn từ danh sách
 3  List<Integer> evenNumbers = numbers.stream()
 4          .filter(n -> n % 2 == 0)
 5          .toList();
 6
 7  System.out.println("Các số chẵn: " + evenNumbers);
 8  // Limit: Giới hạn số lượng phần tử trả về
 9  List<Integer> firstThreeNumbers = numbers.stream()
10          .limit(3)
11          .toList();
12  System.out.println("Ba số đầu tiên: " + firstThreeNumbers);
13
14  // Skip: Bỏ qua ba số đầu tiên và trả về các số còn lại
15  List<Integer> numbersAfterSkip = numbers.stream()
16          .skip(3)
17          .toList();
18  System.out.println("Các số sau khi bỏ qua ba số đầu tiên: " + numbersAfterSkip);
19
20  // Distinct: Lọc các số duy nhất từ danh sách
21  List<Integer> distinctNumbers = Arrays.asList(1, 2, 2, 3, 3, 4, 5, 5, 6, 7, 8, 9,
9, 10);
22  List<Integer> uniqueNumbers = distinctNumbers.stream()
23          .distinct()
24          .toList();
25  System.out.println("Các số duy nhất: " + uniqueNumbers);
26
```

```java
// Sorted: Sắp xếp các số theo thứ tự tăng dần
List<Integer> sortedNumbers = numbers.stream()
        .sorted()
        .toList();
System.out.println("Các số được sắp xếp: " + sortedNumbers);

// Peek: Quan sát và in ra các số chẵn từ danh sách
System.out.print("Các số chẵn:");
numbers.stream()
        .filter(n -> n % 2 == 0)
        .peek(n -> System.out.print(" "+n))
        .toList();
System.out.println();

// Map: Chuyển đổi các số thành bình phương của chúng
List<Integer> squares = numbers.stream()
        .map(n -> n * n)
        .toList();
System.out.println("Bình phương của các số: " + squares);
```

Kết quả khi thực thi chương trình trên:

```
Các số chẵn: [2, 4, 6, 8, 10]
Ba số đầu tiên: [1, 2, 3]
Các số sau khi bỏ qua ba số đầu tiên: [4, 5, 6, 7, 8, 9, 10]
Các số duy nhất: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Các số được sắp xếp: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Các số chẵn: 2 4 6 8 10
Bình phương của các số: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Các phương thức **hoạt động đầu cuối**:

- **count** trả về số phần tử trong Stream dưới dạng giá trị long;
- **max/min** trả lại Optional phần tử lớn nhất / nhỏ nhất của Stream theo bộ so sánh đã cho;
- **reduce** kết hợp các giá trị từ Stream thành một giá trị duy nhất (giá trị tổng hợp);
- **findFirst/findAny** trả về phần tử đầu tiên/bất kỳ của Stream dưới dạng Optional;
- **anyMatch** trả lại true nếu ít nhất một phần tử khớp với một Predicate (xem thêm: allMatch, noneMatch);
- **forEach** lấy một consumer và áp dụng nó cho từng phần tử của Stream (ví dụ: in nó);
- **collect** trả về một tập hợp các giá trị trong Stream;
- **toArray** trả về một mảng các giá trị trong một Stream.

Ví dụ minh hoạ các phương thức trên:

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

// count - trả về số phần tử trong Stream
```

```java
long count = numbers.stream().count();
System.out.println("Số phần tử trong Stream: " + count);

// max/min - trả về Optional phần tử lớn nhất / nhỏ nhất của Stream
Optional<Integer> maxNumber = numbers.stream().max(Integer::compareTo);
maxNumber.ifPresent(max -> System.out.println("Giá trị lớn nhất: " + max));

Optional<Integer> minNumber = numbers.stream().min(Integer::compareTo);
minNumber.ifPresent(min -> System.out.println("Giá trị nhỏ nhất: " + min));

// reduce - kết hợp các giá trị từ Stream thành một giá trị duy nhất
Optional<Integer> sum = numbers.stream().reduce(Integer::sum);
sum.ifPresent(total -> System.out.println("Tổng các số: " + total));

// findFirst/findAny - trả về phần tử đầu tiên/bất kỳ của Stream dưới dạng Optional
Optional<Integer> firstNumber = numbers.stream().findFirst();
firstNumber.ifPresent(first -> System.out.println("Phần tử đầu tiên: " + first));

Optional<Integer> anyNumber = numbers.stream().findAny();
anyNumber.ifPresent(any -> System.out.println("Phần tử bất kỳ: " + any));

// anyMatch - trả lại true nếu ít nhất một phần tử khớp với một Predicate
boolean hasEvenNumber = numbers.stream().anyMatch(n -> n % 2 == 0);
System.out.println("Có số chẵn trong Stream không? " + hasEvenNumber);

// forEach - lấy một consumer và áp dụng nó cho từng phần tử của Stream
System.out.print("Các số trong Stream: ");
numbers.stream().forEach(number -> System.out.print(number + " "));
System.out.println();

// collect - trả về một tập hợp các giá trị trong Stream
List<Integer> squaredNumbers = numbers.stream().map(n -> n * n).collect(Collectors.toList());
System.out.println("Các số bình phương: " + squaredNumbers);

// toArray - trả về một mảng các giá trị trong một Stream
Integer[] numberArray = numbers.stream().toArray(Integer[]::new);
System.out.println("Mảng các số: " + Arrays.toString(numberArray));
```

Kết quả khi chạy mã lệnh trên:

```
Số phần tử trong Stream: 5
Giá trị lớn nhất: 5
Giá trị nhỏ nhất: 1
Tổng các số: 15
Phần tử đầu tiên: 1
Phần tử bất kỳ: 1
Có số chẵn trong Stream không? true
Các số trong Stream: 1 2 3 4 5
Các số bình phương: [1, 4, 9, 16, 25]
Mảng các số: [1, 2, 3, 4, 5]
```

Chú ý, các hoạt động (hàm) như **filter**, **map**, **reduce**, **forEach**, **anyMatch** và một số hàm khác được gọi là hàm bậc cao (higher-order functions) vì chúng chấp nhận các hàm khác làm đối số. Một số hoạt động đầu cuối trả lại Optional bởi vì Stream có thể trống và bạn cần chỉ định giá trị mặc định hoặc một hành động nếu Stream đó trống.

Stream song song

Một trong những tính năng lớn nhất của Stream API và lập trình chức năng nói chung là khả năng dễ dàng viết mã rõ ràng để xử lý dữ liệu song song. Không cần tạo threads theo cách thủ công, kiểm tra xem mã có được đồng bộ hóa tốt hay không và gọi các phương thức wait/notify. Tất cả những điều này được thực hiện bên trong các Stream song song một cách tự động.

Có một số cách đơn giản để tạo Stream song song:

- Gọi phương thức **parallelStream()** của một bộ sưu tập thay vì stream():

```
List<String> languages = List.of("java", "scala", "kotlin", "C#");
List<String> jvmLanguages = languages.parallelStream()
            .filter(lang -> !Objects.equals(lang, "C#"))
            .collect(Collectors.toList());
System.out.print(jvmLanguages); // [java, scala, kotlin]
```

- Chuyển một Stream hiện có thành một Stream song song bằng cách sử dụng phương thức **parallel()**:

```
long sum = LongStream
            .rangeClosed(1, 1_000_000)
            .parallel()
            .sum();
System.out.println(sum); // 500000500000
```

Thực sự dễ dàng tạo một Stream song song, nhưng chúng ta có nên luôn làm điều này không? Không hẳn vậy, một dòng song song không phải lúc nào cũng nhanh hơn so với dòng tuần tự tương đương. Có một số yếu tố ảnh hưởng đáng kể đến hiệu suất của Stream song song. Số lượng lõi (CPU Core) có sẵn trong thời gian chạy. Càng nhiều lõi khả dụng → tốc độ càng lớn.

Chi phí cho mỗi phần tử xử lý. Mỗi phần tử được xử lý càng lâu → khả năng song song hóa càng hiệu quả, nhưng không nên sử dụng dòng song song để thực hiện các hoạt động quá dài (ví dụ: kết nối mạng).

Duyệt và sắp xếp các phần tử

Đưa ra một danh sách các số đã được sắp xếp 1, 2, ..., 10. Xử lý và in từng số từ danh sách bằng các Stream.

Đây là một Stream tuần tự:

```
sortedNumbers.stream()
            .map(Function.identity()) // some processing
            .forEach(n -> System.out.print(n + " "));
```

Đầu ra là:

```
1 2 3 4 5 6 7 8 9 10
```

Đây là một Stream song song:

```
1  sortedNumbers.parallelStream()
2        .map(x -> x) // some processing
3        .forEach(n -> System.out.print(n + " "));
```

Đầu ra:

```
6 7 9 8 10 3 4 1 5 2
```

Lưu ý về thứ tự của Stream song song

Phương thức **forEach()** phá vỡ thứ tự khi được sử dụng với các Stream song song. Nếu viết lại điều này bằng cách sử dụng **forEachOrdered()**, mã sẽ hoạt động như mong đợi:

```
1  sortedNumbers.parallelStream()
2              .map(Function.identity()) // some processing
3              .forEachOrdered(n -> System.out.print(n + " "));
```

Khi sử dụng một Stream song song, hãy sử dụng **forEachOrdered** thay vì **forEach** nếu thứ tự của các phần tử quan trọng đối với bạn. Tuy nhiên, điều này sẽ làm giảm tốc độ từ quá trình song song hóa.

Xét một ví dụ khác, giả sử, chúng ta muốn lấy ba số chẵn đầu tiên từ một Stream song song từ hai Stream được ghép nối.

```
1  // create a filled list of integers
2  List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
3  // create an empty list
4  List<Integer> emptyList = List.of();
```

Đây là cách ghép và xử lý hai danh sách.

```
1  Stream.concat(numbers.stream(), emptyList.stream())
2              .parallel()
3              .filter(x -> x % 2 == 0)
4              .limit(3)
5              .forEachOrdered((n) -> System.out.print(n + " "));
```

Đầu ra:

```
2 4 6
```

Nhưng nếu tạo một danh sách trống bằng cách sử dụng **Collections.emptyList()**, thì chúng ta sẽ có kết quả:

```
2 4 10
```

Vì vậy cần cẩn thận khi sử dụng các Stream song song. Rõ ràng, Stream giúp sử dụng các Stream song song rất dễ dàng. Nhưng không phải lúc nào cũng nhanh hơn các Stream tuần tự tương đương vì hiệu suất của chúng phụ thuộc vào nhiều yếu tố bao gồm khối lượng dữ liệu, phần cứng và các hoạt động được sử dụng. Đồng thời, khá khó để dự đoán tốc độ tăng tốc nếu không thực hiện các phép đo trong thực tế. Ngoài ra, có một số lưu ý có thể xảy ra khi sử dụng một Stream song song đặc biệt liên quan đến thứ tự của các phần tử. Vì vậy, cần phải chắc chắn tại sao cần các Stream song song trong các tình huống khác nhau. Nếu có đủ tài nguyên hoặc các hoạt động được thực hiện đơn giản, không yêu cầu tốc độ cao, có thể tốt hơn nếu sử dụng các Stream tuần tự. Nhưng nếu bạn đã đạt được tốc độ ổn định và nhận thấy tác dụng rõ ràng, có thể thử sử dụng các Stream song song để cải thiện tốc độ. Lưu ý, Stream song song không thể nhanh hơn khi sử dụng cho các máy tính.

Xét một ví dụ khác, giả sử muốn tính tổng các số và thêm 100 vào kết quả. Khi sử dụng Stream tuần tự, bạn chỉ cần đặt 100 làm giá trị ban đầu của phương thức **reduce():**

```
int result = numbers.stream().reduce(100, Integer::sum);
```

Đoạn mã này tạo ra kết quả tương tự như sau:

```
int result = numbers.stream().reduce(0, Integer::sum) + 100;
```

Nhưng khi sử dụng một Stream song song, đoạn mã đầu tiên sẽ tạo ra một kết quả lạ. Lý do là tập dữ liệu của bạn sẽ được chia thành một số phần và giá trị 100 sẽ được thêm vào mỗi phần. Khi sử dụng Stream song song, chỉ sử dụng một phần tử trung tính (0 cho tổng, 1 cho phép nhân, v.v.) làm giá trị ban đầu trong phương thức **reduce()**. Tốt hơn là làm tương tự với các Stream tuần tự.

Câu hỏi ôn tập lý thuyết

1. Lambda expression có thể được sử dụng với loại nào của Java?

- A. Interface
- B. Class
- C. Abstract class
- D. Tất cả đều đúng

2. Trong biểu thức lambda, input có thể có bao nhiêu tham số?

- A. Một
- B. Hai
- C. Ba

D. Tùy ý

3. Biểu thức lambda trong Java được giới thiệu trong phiên bản nào của Java?

A. Java 7

B. Java 8

C. Java 9

D. Java 10

4. Lambda expression trong Java được sử dụng cho mục đích gì?

A. Viết code ngắn gọn hơn

B. Tăng tốc độ thực thi chương trình

C. Giảm kích thước của file code

D. Cả A và B

5. Lambda expression trong Java bao gồm những phần nào?

A. Tham số, operator và giá trị trả về

B. Tham số, biểu thức lambda và giá trị trả về

C. Biểu thức lambda và giá trị trả về

D. Chỉ có giá trị trả về

6. Functional Interface trong Java được sử dụng để làm gì?

A. Để đánh dấu một interface là abstract

B. Để định nghĩa các phương thức trừu tượng

C. Để định nghĩa các phương thức có thể sử dụng được Lambda

D. Để định nghĩa các phương thức có thể sử dụng được anonymous class

7. Cú pháp để sử dụng Lambda với Functional Interface như thế nào?

A. (parameters) -> expression

B. (parameters) -> { statement }

C. (parameters) -> { return expression; }

D. Tất cả đều đúng

8.Trong Method Reference, cú pháp nào sử dụng để tham chiếu tới một phương thức của một đối tượng cụ thể?

A. object::method

B. ClassName::method

C. ClassName::new

D. object::new

9. Optional trong Java là gì?

A. Một kiểu dữ liệu

B. Một interface

C. Một lớp

D. Một hàm

10. Lợi ích chính của Optional là gì?

A. Tránh được NullPointerException

B. Tăng tốc độ xử lý

C. Tạo ra mã nguồn dễ đọc hơn

D. Đảm bảo tính toàn vẹn của dữ liệu

11. Phương thức nào được sử dụng để lọc các phần tử của Stream dựa trên một điều kiện được cung cấp bởi một đối tượng Predicate?

A. filter()

B. map()

C. reduce()

D. forEach()

12. Phương thức nào được sử dụng để thực hiện một hoạt động trên từng phần tử của Stream và trả về một Stream mới chứa kết quả?

A. filter()

B. map()

C. reduce()

D. forEach()

13. Phương thức nào được sử dụng để giảm tập hợp phần tử của Stream thành một phần tử duy nhất bằng cách áp dụng một hoạt động nhất định trên các phần tử?

A. filter()

B. map()

C. reduce()

D. forEach()

Thực hành

Tạo và khởi động tiểu trình

Đây là bài thực hành sử dụng **Lambda** để rút gọn mã khi tạo tiến trình:

```
public static void main(String[] args) {
    Runnable r1 = new Runnable() {
        public void run() {
            System.out.println("Thread1 is running...");
        }
    };
    Thread t1 = new Thread(r1);
    t1.start();
    //Thread Example with lambda
    Runnable r2 = () -> {
        System.out.println("Thread2 is running...");
    };
    Thread t2 = new Thread(r2);
    t2.start();
}
```

Sắp xếp sản phẩm

Bài thực hành dưới đây minh họa việc sử dụng **Lambda** để rút gọn mã khi so sánh và sắp xếp:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Product {
    int id;
    String name;
    int price;
    public Product(int id, String name, int price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class Program {
    public static void main(String[] args) {
        List<Product> list = new ArrayList<Product>();
        //Adding Products
        list.add(new Product(1, "Iphone 14", 850));
        list.add(new Product(3, "Samsung Note21", 300));
        list.add(new Product(2, "Vivo 7 Plus", 150));
        System.out.println("Sorting on the basis of price...");
        Collections.sort(list, (o1, o2) -> (int) (o1.price - o2.price));
        for (Product p : list) {
            System.out.println(p.id + " " + p.name + " " + p.price);
        }
    }
}
```

Xử lý dữ liệu văn bản

Tải file dữ liệu từ địa chỉ dưới đây và đưa vào trong thư mục dự án:

https://raw.githubusercontent.com/mthli/Java/master/CoreJava/gutenberg/alice30.txt

Viết mã để đọc file nội dung nói trên:

```
public static void main(String[] args) {
   try {
       File myObj = new File("alice30.txt");
       Scanner myReader = new Scanner(myObj);
       while (myReader.hasNextLine()) {
           String data = myReader.nextLine();
           System.out.println(data);
       }
       myReader.close();
   } catch (FileNotFoundException e) {
       System.out.println("Có lỗi xảy ra.");
       e.printStackTrace();
   }
}
```

Sử dụng stream để đếm các từ dài (Các từ có nhiều hơn 12 ký tự)

```
public static void main(String[] args) {
   String contents;
   try {
       contents = Files.readString(Paths.get("alice30.txt"));
       List<String> words = List.of(contents.split(" "));
       long count = 0;
       for (String w : words) {
           if (w.length() > 12) count++;
       }
       System.out.println(count);
   } catch (IOException e) {
       e.printStackTrace();
   }
}
```

Kết quả hiển thị số ký tự dài trong văn bản là 947:

```
947
```

Nếu sử dụng Stream, cách viết

```
public static void main(String[] args) {
   String contents;
   try {
       contents = Files.readString(Paths.get("alice30.txt"));
       List<String> words = List.of(contents.split(" "));
```

```
        long count = words.stream().filter(w -> w.length() > 12).count();
        System.out.println(count);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Kết quả tương tự như cách đếm truyền thống:

```
947
```

Đếm các ký tự dài sử dụng kỹ thuật song song (ParallelStra

```
public static void main(String[] args) {
    String contents;
    try {
        contents = Files.readString(Paths.get("alice30.txt"));
        List<String> words = List.of(contents.split(" "));
       long  count = words.parallelStream().filter(w -> w.length() > 12).count();
        System.out.println(count);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Output

```
947
```

Lọc dữ liệu bằng Stream

Triển khai một danh sách và lọc dữ liệu cần từ danh sách đó bằng phương thức thông thường (Duyệt, lặp)

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class Program {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
```

```
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> productPriceList = new ArrayList<Float>();
        for(Product product: productsList){
            // filtering data of list
            if(product.price<30000){
                productPriceList.add(product.price);    // adding price to a produc
tPriceList
            }
        }
        System.out.println(productPriceList);   // displaying data
    }
}
```

Triển khai lại công việc trên bằng cách sử dụng Stream:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
class Product {
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class Program {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        productsList.add(new Product(1, "HP", 25000f));
        productsList.add(new Product(2, "Dell", 30000f));
        productsList.add(new Product(3, "Lenevo", 28000f));
        productsList.add(new Product(4, "Sony", 28000f));
        productsList.add(new Product(5, "Apple", 90000f));
        List<Float> productPriceList2 = productsList.stream()
                .filter(new Predicate<Product>() {
                    @Override
                    public boolean test(Product p) {
                        return p.price > 30000;
                    }
                })// filtering data
                .map(new Function<Product, Float>() {
                    @Override
                    public Float apply(Product p) {
                        return p.price;
                    }
                })        // fetching price
                .collect(Collectors.toList()); // collecting as list
```

```
        System.out.println(productPriceList2);
    }
}
```

Sử dụng Lambda để rút gọn câu lệnh:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
class Product {
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class Program {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1, "HP", 25000f));
        productsList.add(new Product(2, "Dell", 30000f));
        productsList.add(new Product(3, "Lenevo", 28000f));
        productsList.add(new Product(4, "Sony", 28000f));
        productsList.add(new Product(5, "Apple", 90000f));
        List<Float> productPriceList2 = productsList.stream()
                .filter(p -> p.price > 30000)// filtering data
                .map(p -> p.price)        // fetching price
                .collect(Collectors.toList()); // collecting as list
        System.out.println(productPriceList2);
    }
}
```

Thực hành tạo Streams

Tạo stream từ dữ liệu trong tập tin

```
public static void main(String[] args) throws IOException {
    Path path = Paths.get("alice30.txt");
    var contents = Files.readString(path);
    Stream<String> words = Stream.of(contents.split("\\PL+"));
}
```

Tạo phương thức show để in dữ liệu stream đang chứa

```
public static void show(String title, Stream<String> stream) {
    System.out.print(title + ": ");
```

```
 3     stream.limit(10).collect(Collectors.toList()).forEach(s -> System.out.print(s +
",","));
 4     System.out.println();
 5  }
```

Gọi phương thức show từ hàm main()

```
show("words", words);
```

Kết quả hiển thị khi gọi hàm

```
1  words: ***This,is,the,Project,Gutenberg,Etext,of,Alice,in,Wonderland***
2  *This,
```

Tạo sreams từ các "từ":

```
 1  Stream<String> name = Stream.of("Đại", "học", "kinh", "tế", "quốc", "dân");
 2  show("tên", name);
```

Kết quả:

```
tên: Đại, học, kinh, tế, quốc, dân
```

Tạo một stream rỗng:

```
1  Stream<String> silence = Stream.empty();
2  show("silence", silence);
```

Kết quả:

```
silence:
```

Tạo một stream "vang" (echo). Đây là một stream cho ra những nội dung giống nhau, ví dụ

```
1  Stream<String> echos = Stream.generate(() -> "KTQD");
2  show("echos", echos);
```

Kết quả:

```
echos: KTQD, KTQD, KTQD, KTQD, KTQD, KTQD, KTQD, KTQD, KTQD, KTQD, KTQD, KTQD,
```

Tạo ra một Stream chứa dữ liệu ngẫu nhiên, trường hợp này sử dụng một tham chiếu hàm để tạo ra các số ngẫu nhiên:

```
1  Stream<Double> randoms = Stream.generate(Math::random);
2  show("Ngẫu nhiên", randoms);
```

Hàm show có thể viết lại sử dụng lập trình tổng quát:

```
 1  public static <T> void show(String title, Stream<T> stream) {
 2     System.out.print(title + ": ");
 3     stream.limit(10).collect(Collectors.toList()).forEach(s -> System.out.print(s +
",","));
 4     System.out.println();
 5  }
```

Kết quả hiển thị:

```
Ngẫu nhiên: 0.807718927722843,0.36946784644216246,0.08143227201572512,0.644657600309728
4,0.366443263082484,0.9752767239380946,0.6188587519876564,0.7061125729225125,0.82143360
91276199,0.6824309847153482,
```

Bài tập tự thực hành

Thực hành với Lambda

- Viết một chương trình Java sử dụng biểu thức Lambda để sắp xếp một danh sách các chuỗi theo thứ tự từ điển.
- Viết một chương trình Java sử dụng biểu thức Lambda để tính tổng các số trong một danh sách số nguyên.
- Viết một chương trình Java sử dụng biểu thức Lambda để tìm ra số lớn nhất trong một danh sách các số nguyên.

Thực hành với Stream API

- Viết chương trình sử dụng Stream API để tìm ra các số chẵn trong một danh sách số nguyên.
- Viết chương trình sử dụng Stream API để tìm ra các chuỗi có độ dài lớn hơn hoặc bằng 5 trong một danh sách chuỗi.
- Viết chương trình sử dụng Stream API để tìm ra các số nguyên tố trong một danh sách số nguyên.

Tài liệu tham khảo

[1] Core Java: Fundamentals (2021) , Cay Horstmann (Oracle Press Java)

[2] Head First Java: A Brain-Friendly Guide (2022), Kathy Sierra, O'Reilly Media

[3] Java OOP Done Right: Create object oriented code you can be proud of with modern Java Paperback (2019), Mr Alan Mellor, Mellor Books

[4] Murach's Java Programming (5th Edition) (2017), Joe Murach, Mike Murach & Associates

[5] Java for Absolute Beginners Learn to Program the Fundamentals the Java 9+ Way

[6] Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9 (2017), by Ken Kousen, O'Reilly Media

[7] Effective Java (2018), Joshua Bloch, Addison-Wesley Professional.