Korea IT School 2022

JAVA

Unit 9: Stream API

multicampus

Mục lục

multicampus

Functional Interfaces

multicampus

- An **Interface** that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but **can contain only one abstract method**.

```
@FunctionalInterface
interface Sayable {  void say (String msg);
}
public class FunctionalInterfaceExample implements Sayable{
    public void say (String msg) {   System.out.println(msg);
    }
public static void main(String[] args) {
        FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
        fie.say("Hello there");
    }
```

Lambda Expressions

multicampus

- The Lambda expression is used to provide the **implementation of an interface which has functional interface.**

Java Lambda Expression Syntax

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

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

Lambda Expressions

multicampus

- Ví dụ Without Lambda Expression

```
interface Drawable {  public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;
        // without lambda, Drawable implementation using anonymous class
        Drawable d= new Drawable() {
            public void draw  () {System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Lambda Expressions

multicampus

- Ví dụ With Lambda Expression

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;
        //with lambda
        Drawable d2 = ()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Lambda Expressions

multicampus

Ví dụ With Lambda Expression

```
 interface Sayable {   public String say (String name);  }
public class LambdaExpressionExample4{
    public static void main(String[] args) {
            Sayable s1= ( hoten )->{     // Lambda expression with single parameter.
                    return "Hello, "+ hoten;
            };
            System.out.println(s1.say ( "Sonoo“   ));
             Sayable s2= name ->{     // You can omit function parentheses
            return "Hello, "+name;
        };
        System.out.println(s2.say( "Sonoo“  ));
    }
}
```

Lambda Expressions

multicampus

Ví dụ With Lambda Expression

```
Interface Addable {   int  add (int a,int b);
}
public class LambdaExpressionExample6 {
    public static void main(String[] args) {
        Addable ad1=  ( a, b ) -> (a+b) ;   // Lambda expression without return keyword. // { return (a+b); }
        System.out.println(ad1.add(10,20));
       Addable ad2=( int a,int b )-> {     // Lambda expression with return keyword.
                            return (a+b);
                        };
        System.out.println(ad2.add(100,200));
    }
}
```

Method References

multicampus

- Method reference is used to refer method of functional interface.
- There are following types of method references in java:
  - Reference to a static method. // a class method
  - Reference to an instance method.//
  - Reference to a constructor.

Method References

multicampus

- Reference to a static method.

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

Method References

multicampus

- Reference to an instance method.

```
interface Sayable{  void say();  }
public class InstanceMethodReference {
    public void saySomething(){     System.out.println("Hello, this is non-static method.");       }
    public static void main(String[] args) {
        InstanceMethodReference methodReference  = new InstanceMethodReference(); // Creating object
        Sayable sayable = methodReference :: saySomething;   // Referring non-static method using reference
        sayable.say();   // Calling interface method
        // Referring non-static method using anonymous object
        Sayable sayable2 = new InstanceMethodReference() :: saySomething;
        sayable2.say();     // Calling interface method
    }
```

Method References

multicampus

- Reference to a constructor.

```
interface Messageable {  Message getMessage (String msg);  }
class Message {
    Message () {}
    Message (String msg) {   System.out.print(msg);      }
}
public class ConstructorReference {
    public static void main(String[] args) {

        Messageable hello = Message::new;

        hello.getMessage("Hello");
    }
}
```

Constructor Reference with One Argument

multicampus

```java
// Student class
class Student {
    private String name;
    public Student(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```java
@FunctionalInterface
interface MyFunctionalInterface {
    Student getStudent(String name);
}
public class ConstructorReferenceEx1 {
    public static void main(String[] args) {
        MyFunctionalInterface mf = Student::new;

        Function<String, Student> f1 = Student::new;     // Constructor Reference
        Function<String, Student> f2 = (name) -> new Student(name);
        System.out.println(mf.getStudent("Adithya").getName());
        System.out.println(f1.apply("Jai").getName());
        System.out.println(f2.apply("Jai").getName());
    }
}
```

Constructor Reference with Two Arguments

multicampus

```java
@FunctionalInterface
interface MyFunctionalInterface {
    Student getStudent(int id, String name);
}
public class ConstructorReferenceEx2 {
    public static void main(String[] args) {
        MyFunctionalInterface mf = Student::new;     //  Constructor Reference

        BiFunction<Integer, String, Student> f1 = Student::new;
        BiFunction<Integer, String, Student> f2 = (id, name) -> new Student(id,name);

        System.out.println(mf.getStudent(101, "Adithya").getId());
        System.out.println(f1.apply(111, "Jai").getId());
        System.out.println(f2.apply(121, "Jai").getId());
    }
}
```

```java
class Student {
    private int id;
    private String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Một số Functional Interface

multicampus

- Function<T,R>; BiFunction<T,U,R>
- Predicate<T>; BiPredicate<T,U>
- Consumer<T>; BiConsumer<T,U>
- Supplier<T>

Function<T,R>

multicampus

- **Function<T, R>** có thể sử dụng cho **lambda expression** hoặc **method reference** cho một mục đích cụ thể nào đó. **Function<T,R>** chỉ có một method trừu tượng duy nhất chấp nhận **một tham số đầu vào**, và method trả về **một đối tượng khác.**
- Mục đích chính của Function là giúp chúng ta dễ dàng **chuyển một đối tượng** từ kiểu dữ liệu **này** sang kiểu dữ liệu **khác**.

Modifier and Type	Method and Description
`default <V> Function<T,V>`	**`andThen(Function<? super R,? extends V> after)`** Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
`R`	**`apply(T t)`** Applies this function to the given argument.
`default <V> Function<V,R>`	**`compose(Function<? super V,? extends T> before)`** Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.
`static <T> Function<T,T>`	**`identity()`** Returns a function that always returns its input argument.

Function<T,R>

multicampus

- **R apply(T t) :** là một phương thức trừu tượng có thể được sử dụng với lambda expression hoặc method reference cho một mục đích cụ thể nào đó.
- Function<T, T> **identity**() : phương thức này trả về một function luôn trả về đối số đầu vào của nó.

Function<V, R> compose, andThen

multicampus

- Function<V, R> **compose** (Function<? super V, ? extends T> **before**) : phương thức này trả về một function gộp: áp dụng với **before Function** trước và hàm này sau;
- Function<T, V> **andThen** (Function<? super R, ? extends V> **after**) : phương thức này trả về một Function gộp: áp dụng với hàm này trước và after function sau;

```
public class composeAndThen {
    public static void main(String[] args) {
        Function<Integer, Integer> times2 = n -> n * 2;
        Function<Integer, Integer> squared = n -> n * n;

        Function<Integer, Integer> andThen = times2.andThen(squared);//squared: after=> 5*2, 10*10
        System.out.println("Using andThen: " + andThen.apply(5)); // 100

        Function<Integer, Integer> compose = times2.compose(squared);//squared:before=> 5*5, 25 *2
        System.out.println("Using compose: " + compose.apply(5)); // 50
    }
}
```

Function<V, R> với primitive type

multicampus

```
@FunctionalInterface
public interface IntFunction<R> {
    R apply(int value);
}
@FunctionalInterface
public interface LongFunction<R> {
    R apply(long value);
}
@FunctionalInterface
public interface DoubleFunction<R>
{
    R apply(double value);
}
```

```
public static void main (String[] args) {
    IntFunction<String> ifunc
            = (x) -> Integer.toString(x * x);
    LongFunction<String> lfunc
            = (x) -> Long.toString(x * x);
    DoubleFunction<String> dfunc
            = (x) -> Double.toString(x * x);

    System.out.println(ifunc.apply(3)); // 9
    System.out.println(lfunc.apply(5)); // 25
    System.out.println(dfunc.apply(10)); // 100.0
}
```

BiFunction<T,U,R>

multicampus

- Function chỉ chấp nhận **1 đối số đầu vào**, để có thể sử dụng Function với **2 đối số đầu vào** chúng ta sử dụng Interface **BiFunction**.

```
public static void main(String[] args) {
    BiFunction<String, String, String> function1 = (s1, s2) -> s1 + s2;
    System.out.println(function1.apply(“gmail", ".com")); // gmail.com
    BiFunction<Integer, Integer, Integer> function2 = (a, b) -> a + b;
    System.out.println(function2.apply(1, 4)); // 5
    BiFunction<Integer, Integer, Integer> times2 = (a, b) -> a + b;
    Function<Integer, Integer> squared = (n) -> n * n;
    BiFunction<Integer, Integer, Integer> andThen = times2.andThen(squared);
    System.out.println("Using andThen: " + andThen.apply(5, 2)); // 49
}
```

Predicate<T>

multicampus

- **Predicate<T>** là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference.
- Predicate<T> sẽ trả về giá trị true/false tương ứng với một tham số kiểu **T** đưa vào, cụ thể là điều kiện được viết trong phương thức **test()**.

Predicate<T> -- Một số phương thức default

multicampus

- and() : Nó thực hiện logic AND của predicate mà nó được gọi với một biến predicate khác. Ví dụ: predicate1.and (predicate2).
- or() : Nó thực hiện logic OR của predicate mà nó được gọi với một biến predicate khác. Ví dụ: predicate1.or(predicate2).
- negate() : Nó thực hiện phủ định kết quả của biến predicate được gọi. Ví dụ: predicate1.negate().

Predicate<T>

multicampus

```
public static void main(String[] args) {
    Predicate<String> predicateString = s -> {  return s.equals("mta.edu.vn"); };
    System.out.println(predicateString.test(" mta.edu.vn ")); // true

    Predicate<Integer> predicateInt = x -> {return x > 0;};
    System.out.println(predicateInt.test(1)); // true
    System.out.println(predicateInt.test(-1)); // false
}
```

Predicate<T>: and, or,..

multicampus

```
public static void main(String[] args) {
    Predicate<String> predicate = s -> {return s.equals("mta.edu.vn");   };
    Predicate<String> predicateAnd = predicate.and (s -> s.length() == 10);
    System.out.println(predicateAnd.test("mta.edu.vn")); // true
    Predicate<String> predicateOr = predicate.or(s -> s.length() == 10);
    System.out.println(predicateOr.test(“mta.edu.vn")); // true
    Predicate<String> predicateNegate = predicate.negate();
    System.out.println(predicateNegate.test(“mta.edu.vn ")); // false
}
```

Kết hợp nhiều biến Predicate<T>

multicampus

```
public static void main(String[] args) {
    Predicate<Integer> greaterThanTen = (i) -> i > 10;
    Predicate<Integer> lessThanTwenty = (i) -> i < 20;
    // Calling Predicate Chaining
    boolean result = greaterThanTen.and(lessThanTwenty).test(15);
    System.out.println(result); // true
    // Calling Predicate method
    boolean result2 = greaterThanTen.and(lessThanTwenty).negate().test(15);
    System.out.println(result2); // false
}
```

Predicate với primitive type

multicampus

```
public static void main(String[] args) {
    int[] intNumbers = { 3, 5, 6, 2, 1 };
    IntPredicate intPredicate = i -> i > 5;
    Arrays.stream(intNumbers).filter(intPredicate).forEach(System.out::println);
    long[] longNumbers = { 3, 5, 6, 2, 1 };
    LongPredicate longPredicate = l -> l > 5;
    Arrays.stream(longNumbers).filter(longPredicate).forEach(System.out::println);
    double[] dbNumbers = { 3.2, 5.0, 6.3, 2.5, 1.0 };
    DoublePredicate dbPredicate = d -> d > 5;
    Arrays.stream(dbNumbers).filter(dbPredicate).forEach(System.out::println);
}
```

BiPredicate<T,U>

multicampus

- Về cơ bản, interface **BiPredicate** không khác biệt so với **Predicate**, ngoại trừ nó chấp nhận **2** đối số đầu vào.

```
public static void main(String[] args) {
    BiPredicate<Integer, String> condition = (i, s) -> i > 2 && s.startsWith("Java");
    System.out.println(condition.test(5, "Java")); // true
    System.out.println(condition.test(2, "Javascript")); // false
    System.out.println(condition.test(5, "C#")); // false
}
```

Consumer<T>

multicampus

- **Consumer<T>** là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference. **Consumer<T>** chấp nhận một tham số đầu vào, và method này không trả về gì cả.

```
interface Consumer<T> {
   void accept(T t);
  default Consumer<T> andThen (Consumer<? super T> after) ;
}
```

Consumer<T>

multicampus

```
public class Consumer1 {
    static void printValue (int val) { System.out.println(val);    }
   public static void main(String[] args) {
        // Create Consumer interface with lambda expression
        Consumer<String> consumer1 = (name) -> System.out.println("Hello, " + name);
        consumer1.accept("mta.edu.vn"); // Hello, mta.edu.vn
        // Create Consumer interface with method reference
        Consumer<Integer> consumer2 = Consumer1::printValue;
        consumer2.accept(1966); // 1966
    }
}
```

Consumer<T> andThen()

multicampus

```
public static void main(String[] args) {
    Consumer<Integer> times2 = (e) -> System.out.println(e * 2);
    Consumer<Integer> squared = (e) -> System.out.println(e * e);
    Consumer<Integer> isOdd = (e) -> System.out.println(e % 2 == 1);
    times2.accept(5); // 5*2=10
    squared.accept(5); // 5*5=25
    isOdd.accept(5); // 5%2=1=> true
     Consumer<Integer> combineConsumer1 = times2.andThen(squared);
     combineConsumer1.accept(5); // 10 25
    Consumer<Integer> combineConsumer2 = times2.andThen(squared).andThen(isOdd);
    combineConsumer2.accept(5); // 10 25 true
}
```

BiConsumer

multicampus

- Về cơ bản, interface **BiConsumer** không khác biệt so với **Consumer**, ngoại trừ nó chấp nhận **2** đối số đầu vào.

```
public static void main(String[] args) {
    Map<String, Integer> map = new HashMap<>();
    map.put("Java", 5);       map.put("PHP", 2);       map.put("C#", 1);
    BiConsumer<String, Integer> biConsumer
        = (key, value) -> System.out.println("Key: " + key + " - Value: " + value);
    biConsumer.accept("ABC", 100);
    map.forEach(biConsumer);
}
```

Supplier<T>

multicampus

- Supplier<T> là một functional interface và do đó nó có thể được sử dụng với lambda expression hoặc method reference.

```
interface Supplier<R>{
    R get();//=> R is a return type
}
 public static void main(String[] args) {
        Supplier<String> supplier = () -> "Welcome to Java Core";
        String hello = supplier.get();
        System.out.println(hello); //-> "Welcome to Java Core"
    }
```

BinaryOperator

multicampus

- BinaryOperator represents an operation upon two operands of the same type, producing a result of the same type.

```
public static void main(String[] args) {
    BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;
    System.out.println(adder.apply(3, 4));
}
```

Stream

multicampus

- A stream represents a sequence of elements supporting sequential and parallel aggregate operations. Since Java 8, we can generate a stream from a collection, an array or an I/O channel.

```
List<Student> listStudents = new ArrayList<>();
 listStudents.add(new Student("Alice", 82));
Stream<Student> stream = listStudents.stream();
int[] arrayIntegers = {1, 8, 2, 3, 98, 11, 35, 91};
IntStream streamIntegers = Arrays.stream(arrayIntegers);
BufferedReader bufferReader = new BufferedReader(new FileReader("students.txt"));
Stream<String> streamLines = bufferReader.lines();
```

Java Stream API

multicampus

- Java Stream API which is added to JDK since Java 8.
- Java Stream API brings to us totally new ways for working with collections.

```java
public class Student implements Comparable<Student> {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void setScore(int score) {
        this.score = score;
    }

    public int getScore() {
        return this.score;
    }

    public String toString() {
        return this.name + " - " + this.score;
    }

    public int compareTo(Student another) {
        return another.getScore() - this.score;
    }
}
```

```java
List<Student> listStudents = new ArrayList<>();

listStudents.add(new Student("Alice", 82));
listStudents.add(new Student("Bob", 90));
listStudents.add(new Student("Carol", 67));
listStudents.add(new Student("David", 80));
listStudents.add(new Student("Eric", 55));
listStudents.add(new Student("Frank", 49));
listStudents.add(new Student("Gary", 88));
listStudents.add(new Student("Henry", 98));
listStudents.add(new Student("Ivan", 66));
listStudents.add(new Student("John", 52));
```

Java Stream API

multicampus

- First, find the students whose scores are greater than or equal to 70.

A non-stream solution would look like this:

```
1  // find students whose score >= 70
2
3  List<Student> listGoodStudents = new ArrayList<>();
4
5  for (Student student : listStudents) {
6      if (student.getScore() >= 70) {
7          listBadStudents.add(student);
8      }
9  }
10
11 for (Student student : listGoodStudents) {
12     System.out.println(student);
13 }
```

With the Stream API, we can replace the above code with the following:

```
1  // find students whose score >= 70
2  List<Student> listGoodStudents = listStudents.stream()
3                      .filter(s -> s.getScore() >= 70)
4                      .collect(Collectors.toList());
5
6  listGoodStudents.stream().forEach(System.out::println);
```

Java Stream API

multicampus

```java
// calculate average score of all students

double sum = 0.0;

for (Student student : listStudents) {
    sum += student.getScore();
}

double average = sum / listStudents.size();

System.out.println("Average score: " + average);
```

And here's a stream-based version:

```java
// calculate average score of all students
double average = listStudents.stream()
            .mapToInt(s -> s.getScore())
            .average().getAsDouble();

System.out.println("Average score: " + average);
```

Stream Pipeline

multicampus

- A stream is a pipeline of aggregate operations. A pipeline consists of the following parts:
  - A source: can be a collection, an array or an I/O channel.
  - Zero or more **intermediate operations** which produce new streams, such as filter, map, sorted, etc.
  - **A terminal operation** that produces a non-stream result such as a **primitive value**, a collection, or void (such as the forEach operation).
-

Intermediate Operations

multicampus

- An intermediate operation processes over a stream and return a new stream as a response. Then we can execute another intermediate operation on the new stream, and so on, and finally execute the terminal operation.
- One interesting point about intermediate operations is that they are lazily executed. That means they are not run until a terminal operation is executed.
- The Stream API provides the following common intermediate operations:
    - **map()**
    - **filter()**
    - **sorted()**
    - **limit()**
    - **distinct()**

Terminal Operations

multicampus

- A stream pipeline always ends with a terminal operation, which returns a concrete type or produces a side effect. For instances, the collect operation produces a collection; the forEach operation does not return a concrete type, but allows us to add side effect such as print out each element.
- The common terminal operations provided by the Stream API include:
  - **collect()**
  - **reduce()**
  - **forEach()**

Stream Aggregate Functions Examples (Intermediate Operations)

multicampus

- public class Person implements Comparable<Person> {

  private String firstName;

  private String lastName;

  private String email;

  private Gender gender;

  private int age;

… }

```
List <Person> listPersons = new ArrayList<>();

listPersons.add(new Person("Bob", "Young", "bob@gmail.com", Gender.MALE, 32));

listPersons.add(new Person("Carol", "Hill", "carol@gmail.com", Gender.FEMALE, 23));
```

Stream filter operation- Aggregate Functions

multicampus

```
1  listPersons.stream()
2      .filter(p -> p.getGender().equals(Gender.MALE))
3      .forEach(System.out::println);
```

```
1  listPersons.stream()
2      .filter(p -> p.getGender().equals(Gender.FEMALE) && p.getAge() <= 25)
3      .forEach(System.out::println);
```

Stream map operation- Aggregate Functions

multicampus

- The map operation returns a new stream.
- The Stream API provides **4 methods** for the map operation:

✓ **map()**: transforms a stream of **objects of type T** to a stream of objects of type **R**.

✓ **mapToInt()**: transforms a stream of objects to a stream of int primitives.

✓ **mapToLong()**: transforms a stream of objects to a stream of long primitives.

✓ **mapToDouble()**: transforms a stream of objects to a stream of double primitives.

The following code maps each person to his/her respective email address:

```
1  listPersons.stream()
2      .map(p -> p.getEmail())
3      .forEach(System.out::println);
```

Stream map operation- Aggregate Functions

multicampus

- The following example maps each person to his/her age:

```
1  listPersons.stream()
2      .mapToInt(p -> p.getAge())
3      .forEach(age -> System.out.print(age + " - "));
```

The following example maps each person to his/her first name in uppercase:

```
1  listPersons.stream()
2      .map(p -> p.getFirstName().toUpperCase())
3      .forEach(System.out::println);
```

Stream sorted operation- Aggregate Functions

multicampus

- The Stream API provides two overloads of the sorted operation that returns a new stream consisting elements sorted according to natural order or by a specified comparator:
  - ✓ **sorted()**: sorts by natural order
  - ✓ **sorted(comparator)**: sorts by a comparator
- Săp xếp theo:

```
public int compareTo(Person another) {
    return this.age - another.getAge();
}
```

```
1  listPersons.stream()
2      .sorted()
3      .forEach(p -> System.out.println(p + " - " + p.getAge()));
```

Stream sorted operation- Aggregate Functions

multicampus

- Sắp xếp sử dụng comparator

```
1  listPersons.stream()
2      .sorted((p1, p2) -> p1.getLastName().compareTo(p2.getLastName()))
3      .forEach(System.out::println);
```

Stream distinct operation- Aggregate Functions

- The distinct() operation returns a stream consisting of the distinct elements (no duplicates) by comparing objects via their hashCode, equals() method.

The following example returns a stream of distinct numbers from an array source:

```
1  int[] numbers = {23, 58, 12, 23, 17, 29, 99, 98, 29, 12};
2  Arrays.stream(numbers).distinct().forEach(i -> System.out.print(i + " "));
```

Output:

```
1  23 58 12 17 29 99 98
```

Stream limit operation- Aggregate Functions

multicampus

- The limit() operation returns a stream containing only a specified number of elements.
- Ex:

```
1  listPersons.stream()
2          .sorted()
3          .limit(5)
4          .forEach(System.out::println);
```

Stream skip operation- Aggregate Functions

multicampus

- The **skip()** operation returns a stream containing the remaining elements after discarding the first **n** elements of the stream.
- Ex: Combining with the sorted and map operations, the following example finds the oldest age of the persons above:

```
1  System.out.print("The oldest age: ");
2  listPersons.stream()
3      .sorted()
4      .map(p -> p.getAge())
5      .skip(listPersons.size() - 1)
6      .forEach(System.out::println);
```

allMatch operation -Terminal Operations

multicampus

- The **allMatch()** operation answers the question: ***Do all elements in the stream meet this condition?*** It returns true if and only if all elements match a provided predicate, otherwise return false.

The following example checks if all persons are male:

```
1  boolean areAllMale = listPersons.stream()
2                    .allMatch(p -> p.getGender().equals(Gender.MALE));
3
4  System.out.println("Are all persons male? " + areAllMale);
```

anyMatch operation -Terminal Operations

multicampus

- The anyMatch() operation returns true if any element in the stream matches a provided predicate. In other words, it answers the following question: Is there any element that meets this condition?
- The following example checks whether the list has any female:

```
boolean anyFemale = listPersons.stream()
                                .anyMatch(p -> p.getGender().equals(Gender.FEMALE));
System.out.println("Is there any female? " + anyFemale);
```

noneMatch operation -Terminal Operations

multicampus

- In contrast to the allMatch() operation, the noneMatch() operation returns true if no elements in the stream match a provided predicate. In other words, it answers the question: Does no element meet this condition?
- The following example checks if no one uses Yahoo email:

```
boolean noYahooMail = listPersons.stream()
                .noneMatch(p -> p.getEmail().endsWith("yahoo.com"));

System.out.println("No one uses Yahoo mail? " + noYahooMail);
```

collect operation -Terminal Operations

multicampus

- The collect() operation accumulates elements in a stream into a container such as a collection.
- The following example accumulates emails of the persons into a list collection:

```
List<String> listEmails = listPersons.stream()
                .map(p -> p.getEmail())
                .collect( Collectors.toList()    );

System.out.println("List of Emails: " + listEmails);
```

count operation -Terminal Operations

multicampus

- The count() operation simply returns total number of elements in the stream. The following example finds how many people are male:

```
long totalMale = listPersons.stream()
        .filter(p -> p.getGender().equals(Gender.MALE))
        .count();
System.out.println("Total male: " + totalMale);
```

forEach operation -Terminal Operations

multicampus

- The forEach() operation performs an action for each element in the stream, thus creating a side effect, such as print out information of each female person as shown in the following example:

```
System.out.println("People are female:");
 listPersons.stream()
    .filter(p -> p.getGender().equals(Gender.FEMALE))
    .forEach(System.out::println);
```

Min(max) operation -Terminal Operations

multicampus

- The min (max) returns the minimum (max) element in the stream according to the provided comparator
- Ex: finds the youngest female person in the list:

```
Optional<Person> min = listPersons.stream()
        .filter(p -> p.getGender().equals(Gender.FEMALE))
        .min ( (p1, p2) -> p1.getAge() - p2.getAge() );

if (min.isPresent()) {
    Person youngestGirl = min.get();
    System.out.println("The youngest girl is: "
            + youngestGirl + " (" + youngestGirl.getAge() + ")");
}
```

reduce operation -Terminal Operations

multicampus

- The Stream API provides three versions of **reduce()** methods which are general reduction operations. Let's look at each version.
- **Version #1**: **Optional<T> reduce(BinaryOperator<T> accumulator)**

This method performs a reduction on the elements of the stream, using an associative accumulation function, and returns an Optional object describing the reduced value. For example, the following code accumulates first names of all persons into a String:

```
Optional<String> reducedValue = listPersons.stream()
                .map(p -> p.getFirstName())
                .reduce( (name1, name2) -> name1 + ", " + name2);
 if (reducedValue.isPresent()) {
    String names = reducedValue.get();
    System.out.println(names);
}
```

reduce operation -Terminal Operations

multicampus

- **Version #2: T reduce (T identity, BinaryOperator<T> accumulator)**
- This method is similar to the version #1, but it returns the reduced value of the specified type T. The identity value là giá trị khởi tạo để bắt đầu tích luỹ.
- The following example calculates sum of numbers in a stream:

```
int[] numbers = {123, 456, 789, 246, 135, 802, 791};
int sum = Arrays.stream(numbers).reduce( 0, (x, y) -> (x + y));
System.out.println("sum = " + sum);
```

(Accumulator: bộ tích luỹ với giá trị khởi tạo ban đầu)

reduce operation -Terminal Operations

multicampus

- **Version #3**: U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator <U> combiner)
- This is the most general reduction method that performs a reduction on elements of the stream, using the provided identity, accumulator and combiner.
- The following example shows how this general reduction operation is used to accumulate numbers to calculate sum of them:

```
Integer prod= Stream.of (2, 3, 4).parallel()
        .reduce (0, (x, y) -> x + y, (p, q) ->  p * q);
System.out.println (prod);//24
```

- Accumulator (Tích luỹ): từng giá trị x được tính bằng (x+identity)
- Sau đó được kết hợp bằng combiner

Parallel Streams

multicampus

- The powerful feature of streams is that stream pipelines may execute either sequentially or in parallel. All collections support the **parallelStream()** method that returns a possibly parallel stream.
- When a stream executes in parallel, the Java runtime divides the stream into multiple sub streams. The aggregate operations iterate over and process these sub streams in parallel and then combine the results.

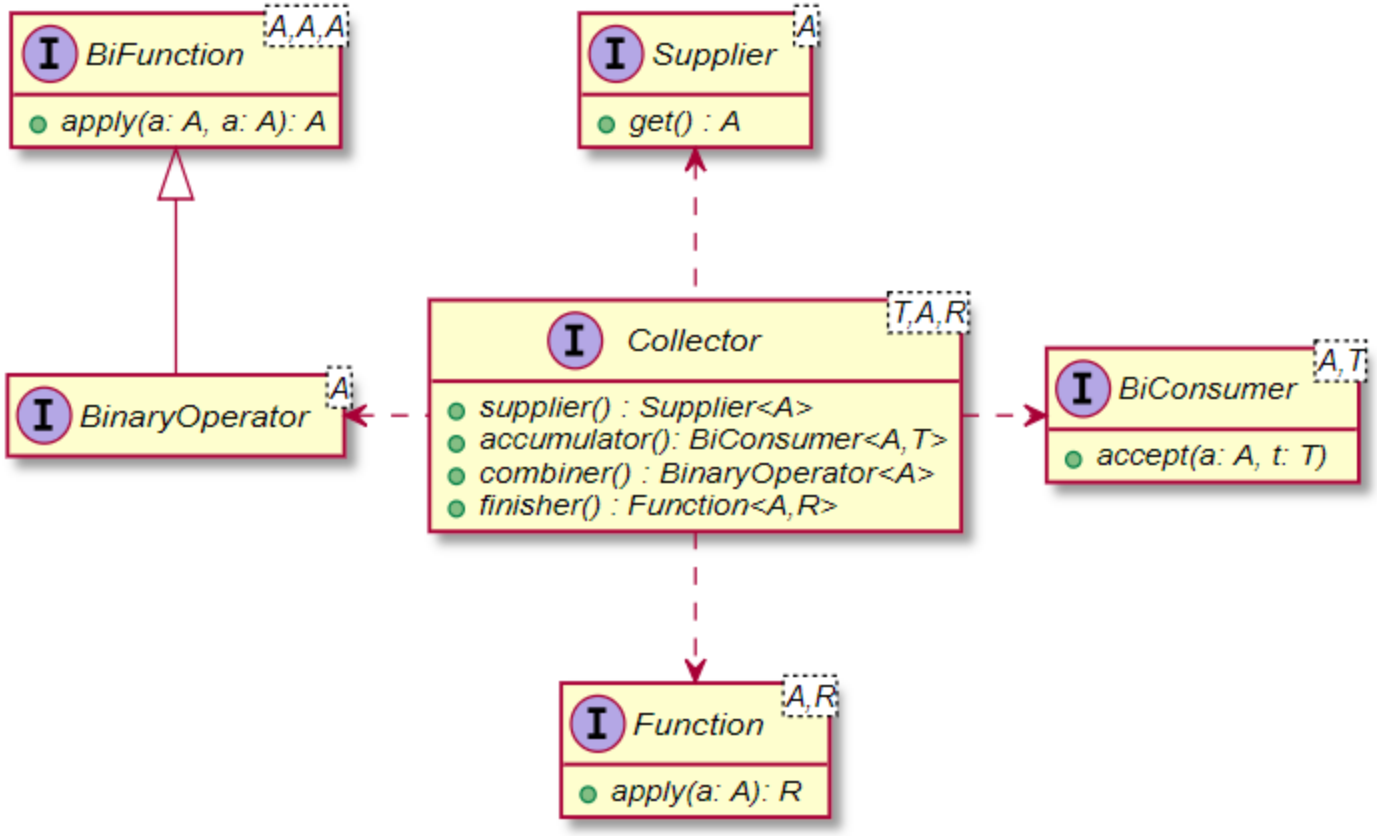
```
listStudents.parallelStream()
    .filter(s -> s.getScore() >= 70)
    .sorted().limit(3)
    .forEach(System.out::println);
```

Lớp Collectors

multicampus

- Stream.collect() là một trong các phương thức đầu cuối (terminal operation) của Stream API. Nó cho phép thực hiện các thao tác có thể thay đổi trên các phần tử được lưu giữ trong Stream. Chẳng hạn như: chuyển các phần tử sang một số cấu trúc dữ liệu khác, áp dụng một số logic bổ sung, tính toán, …
- Lớp Java Collectors cung cấp nhiều phương thức khác nhau để xử lý các phần tử của Stream API:
  - Collectors.toList()
  - Collectos.toSet()
  - Collectors.toMap()
  - …

Lớp Collectors



- Mỗi static method trong lớp **Collectors** đều trả về một object kiểu **Collector**. Collector là một **interface**.
- **Collector<T, A, R>**:

T : kiểu dữ liệu của phần tử trong Stream

A : kiểu dữ liệu được sử dụng để giữ một phần kết quả của hoạt động thu thập.

R : kiểu dữ liệu trả về.

Collectors

multicampus

- Collectors.toList(): có thể được sử dụng để thu thập tất cả các phần tử Stream vào một List (kết quả luôn là ArrayList):

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

- **Collectors.toSet()** : có thể được sử dụng để thu thập tất cả các phần tử Stream vào một **Set** (kết quả luôn là **HashSet**).

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

- **Collectors.toCollection():** Khi sử dụng **Collectors.toSet(); toList()**, chúng ta không thể xác định lớp cài đặt cụ. Nếu muốn sử dụng lớp cài đặt cụ thể, chúng ta cần phải sử dụng phương thức **Collectors.toCollection( )**

```
List<String> result = list.stream().collect(Collectors.toCollection(LinkedList::new));
```

Collectors.toMap()

multicampus

- **public static** Collector<T, ?, M> toMap (**Function**<? **super** T, ? **extends** K> *keyMapper*, **Function**<? **super** T, ? **extends** U> *valueMapper*, **BinaryOperator<U>** mergeFunction, **Supplier** <M> mapSupplier) {}

keyMapper : để trích xuất một khóa (key) từ một phần tử Stream.

valueMapper : để trích xuất một giá trị (value) được liên kết với một khóa (key).

mergeFunction : để giải quyết xung đột của khóa. Tham số này không bắt buộc. Tuy nhiên, nếu trong danh sách có key trùng, nó sẽ throw một ngoại lệ IllegalStateException.

mapSupplier: trả về một instance của **Map** mới, rỗng, trong đó kết quả sẽ được chèn vào. Tham số này không bắt buộc, mặc định là **HashMap**.

Collectors.toMap()

multicampus

```
public static void main(String[] args) {
    List<Student> students = Arrays.asList(    new Student("B", 70), new Student("A", 80),   new Student("C", 75), new Student("A", 100)   );
   // Resolve collisions between values associated with the same key
    Map<String, Integer> result1 = students.stream().collect(  Collectors.toMap(Student::getName, Student::getScore,   (s1, s2) -> (s1 > s2 ? s1 : s2) //khi trùng key thì so sánh 2 value, lấy giá trị lớn hơn
      ));
    System.out.println("result1 : " + result1);
    // Identify the instance of LinkedHashMap
    Map<String, Integer> result2 = students.stream().collect( Collectors.toMap(Student::getName, Student::getScore , (s1, s2) -> (s1 > s2 ? s1 : s2),   LinkedHashMap::new // Supplier
      ));
    System.out.println("result2 : " + result2);
  }
```

Collectors.collectingAndThen()

multicampus

- **Collectors.collectingAndThen()** là một bộ thu thập đặc biệt cho phép thực hiện một hành động khác ngay lập tức sau khi thu thập kết thúc.

```
public static void main(String[] args) {
    List<String> list = Arrays.asList("Java", "C++", "C#", "PHP");
    List<String> result5 = list.stream().collect(
    // Lấy 2 phần tử từ vị trí 0
    Collectors.collectingAndThen(Collectors.toList(), x -> x.subList(0, 2)));
    System.out.println(result5);
    // => [Java, C++]
}
```

Collectors.groupingBy()

multicampus

- public static <T, K> Collector<T, ?, **Map<K, List<T>>** > **groupingBy** (Function<? super T, ? extends K> classifier);

The simplest variant of groupingBy() method applies classifier Function<**T**,***R***> to each individual element of type T collected from Stream<**T**>.

Then groups elements into individual lists based on classifier, and stores them in a Map<***R***, List<T>>.

Phương thức collect của Stream:

collect (Collectors.groupingBy()) sẽ thu thập và trả về kiểu Map<***R***, List<T>> (Cụ thể mặc định là HashMap)

Collectors.groupingBy()

multicampus

- Collectors.groupingBy() : được sử dụng để nhóm các đối tượng theo một số thuộc tính và lưu trữ các kết quả trong một Map.

```
public static void main(String[] args) {
    List<Book> books = Arrays.asList(   new Book(1, "A", 1),   new Book(2, "B", 1), //
            new Book(3, "C", 2),   new Book(4, "D", 3),  new Book(5, "E", 1) //
    );
    Map<Integer, Set<Book>> result = books.stream()
            .collect( Collectors.groupingBy( Book::getCagegoryId, Collectors.toSet()) );
    result.forEach((catId, booksInCat)
            -> System.out.println("Category " + catId + " : " + booksInCat.size()));
}
```

Collectors.partitioningBy()

multicampus

- Collectors.partitioningBy() : là một trường hợp đặc biệt của Collectors. groupingBy() nhận vào một Predicate và thu thập các phần tử của Stream vào một Map với giá trị Boolean như khóa và Collection như giá trị.

Key = true, là một tập hợp các phần tử phù hợp với Predicate đã cho

Key = false, là một tập hợp các phần tử không khớp với Predicate đã cho.

```
public static void main(String[] args) {
    List<Book> books = Arrays.asList(  new Book(1, "A", 1),  new Book(2, "B", 1), //
            new Book(3, "C", 2),  new Book(4, "D", 3),  new Book(5, "E", 1) //
    );
    Map<Boolean, Set<Book>> partitioningBy = books.stream()
            .collect(Collectors.partitioningBy(b -> b.getCagegoryId() > 2, Collectors.toSet()));
    System.out.println(partitioningBy);
}
```

Collectors.reducing()

multicampus

- **Collectors.reducing()** : thực hiện giảm các phần tử đầu vào của nó trong một **BinaryOperator** được chỉ định.

```
public static <T> Collector<T,?,T>          reducing(T identity, BinaryOperator<T> op) {}
public static <T> Collector<T, ?, Optional<T>> reducing(BinaryOperator<T> op) {}
```

- **identity** : giá trị khởi tạo để thực hiện reduction (cũng là giá trị được trả về khi không có phần tử đầu vào).
- **op** : một BinaryOperator<T> được sử dụng để giảm các phần tử đầu vào.

Collectors.reducing()

multicampus

```
public static void main(String[] args) {
    List<Employee> list = Arrays.asList( new Employee("Emp1", 22, "A", 50),   new Employee("Emp2", 23, "A", 60),    new Employee("Emp3", 22, "B", 40),   new Employee("Emp4", 21, "B", 70)   );
    // Find employees with the maximum age of each company
    Comparator<Employee> ageComparator = Comparator.comparing(Employee::getAge);
    Map<String, Optional<Employee>> map = list.stream().collect(
        Collectors.groupingBy(Employee::getCompanyName,
            Collectors.reducing( BinaryOperator.maxBy(ageComparator))));
    map.forEach((k, v) -> System.out.println(  "Company: " + k  + ", Age: " + ((Optional<Employee>) v).get().getAge()  + ", Name: " + ((Optional<Employee>) v).get().getName()));
}
```

THANK YOU

multicampus