

람다, 스트림 API

참고

- 참고 도서
 - 자바 8 인 액션
- 예제 코드
 - https://github.com/clghks/javacafe_lambda_stream



목차

- 람다
 - 람다란 무엇인가?
 - 함수형 인터페이스
 - 람다 활용 : 실행 어라운드 패턴
 - 메서드 레퍼런스
- 스트림 API 활용
 - 필터링과 슬라이싱
 - 매핑
 - 리듀싱
 - 스트림 만들기
- 스트림 API
 - 스트림이란 무엇인가?
 - 스트림과 컬렉션
 - 스트림 연산
- 병렬 데이터 처리와 성능
 - 순차 스트림을 병렬 스트림으로 변환하기
 - 스트림 성능 측정
 - 병렬 스트림의 올바른 사용법

람다(Lambda)

람다란 무엇인가?

- Java 7

```
public static void java7Style() {
    filterApplesByColor(inventory, "green");
    filterApplesByWeight(inventory, 150);
    filterApples(inventory, "green", 150);
}

public static List<Apple> filterApplesByColor(List<Apple> inventory, String color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getColor().equals(color)) {
            result.add(apple);
        }
    }

    return result;
}

public static List<Apple> filterApplesByWeight(List<Apple> inventory, int weight) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getWeight() > weight) {
            result.add(apple);
        }
    }

    return result;
}

public static List<Apple> filterApples(List<Apple> inventory, String color, int weight) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (apple.getWeight() > weight && apple.getColor().equals(color)) {
            result.add(apple);
        }
    }

    return result;
}
```

- 람다 사용

```
public static void java8Style() {
    filterApples(inventory, (Apple apple) -> apple.getColor().equals("green"));
    filterApples(inventory, (Apple apple) -> apple.getWeight() > 150);
    filterApples(inventory, (Apple apple) -> apple.getColor().equals("green") && apple.getWeight() > 150);
}

public static List<Apple> filterApples(List<Apple> inventory, Predicate<Apple> predicate) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
        if (predicate.test(apple)) {
            result.add(apple);
        }
    }

    return result;
}
```

람다란 무엇인가?

- 람다 표현식
 - 메서드로 전달 할 수 있는 익명 함수를 단순화한 것
 - 이름은 없지만, 파라미터 리스트, 바디, 반환 형식, 예외 리스트를 가질 수 있다.
- 람다의 특징
 - 익명
 - 보통의 메서드와 달리 이름이 없으면 익명이라 표현한다.
 - 함수
 - 람다는 메서드처럼 특정 클래스에 종속되지 않으므로 함수라고 부른다.
 - 전달
 - 람다 표현식을 메서드 인수로 전달하거나 변수로 저장할 수 있다.
 - 간결성
 - 익명 클래스처럼 많은 자질구레한 코드를 구현할 필요가 없다.

람다란 무엇인가?

- 람다가 기술적으로 **Java8** 이전의 자바로 할 수 없었던 일을 해낸다던가 하는 드라마틱한 기술은 아니다.
- 람다 표현식을 이용하면 코드가 간결하고 유연해진다.

```
public static void java8Style() {  
    filterApples(inventory, (Apple apple) -> apple.getColor().equals("green"));  
    filterApples(inventory, (Apple apple) -> apple.getWeight() > 150);  
    filterApples(inventory, (Apple apple) -> apple.getColor().equals("green") && apple.getWeight() > 150);  
}  
  
public static List<Apple> filterApples(List<Apple> inventory, Predicate<Apple> predicate) {  
    List<Apple> result = new ArrayList<>();  
    for (Apple apple : inventory) {  
        if (predicate.test(apple)) {  
            result.add(apple);  
        }  
    }  
    return result;  
}
```

람다 표현식 문법

화살표

(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

람다 파라미터

람다 바디

- 람다 표현식은 세부분으로 이루어진다.
 - 파라미터 리스트
 - Comparator의 compare 메서드의 파라미터
 - 화살표
 - 화살표(->)는 람다의 파라미터 리스트와 바디를 구분한다.
 - 람다의 바디
 - 두 사과의 무게를 비교한다.

람다 표현식 문법

- 식을 이용한 경우 (표현식)

(parameters) -> expression

- 문을 이용하는 경우 (구문)
 - 여러 문장을 포함할 수 있다.

**(parameters) -> {
statements;
}**

람다 표현식 문법

(String s) -> s.length()

(Apple a) -> a.getWeight() > 150

```
(int x, int y) -> {  
    System.out.println("Result : ");  
    System.out.println(x + y);  
}
```

() -> 42

(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

람다 표현식 문법 (사용 예제)

사용 사례	람다 예제
불린 표현식	<code>(List<String> list) -> list.isEmpty()</code>
객체 생성	<code>() -> new Apple(10)</code>
객체에서 소비	<code>(Apple a) -> { System.out.println(a.getWeight()); }</code>
두 값을 조합	<code>(int a, int b) -> a * b</code>

퀴즈 - 람다 표현식 문법

- 람다 규칙에 맞지 않는 람다 표현식을 고르시오.
 1. `() -> {}`
 2. `() -> "Raoul"`
 3. `() -> {return "Mario";}`
 4. `(Integer i) -> return "Alan" + i`
 5. `(String s) -> {"Iron Man";}`

퀴즈 정답

- 4번과 5번이 유효하지 않은 람다 표현식

4. (Integer i) -> {return "Alan" + i;} 처럼 되어야 올바른 람다 표현식이다.

5. (String s) -> "Iron Man" 또는 (String s) -> {return "Iron Man";} 처럼 되어야 올바른 람다 식이다.

함수형 인터페이스

- 람다 표현식은 함수형 인터페이스라는 문맥에서 사용할 수 있다.
- 함수형 인터페이스는 정확히 하나의 추상 메서드를 지정하는 인터페이스다.
- 자바 **API**의 함수형 인터페이스로 **Comparator**, **Runnable** 등이 있다.
- 디폴트 메서드가 있더라도 추상 메서드가 오직 하나면 함수형 인터페이스다.

함수형 인터페이스

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface Runnable is used
     * to create a thread, starting the thread causes the object's
     * run method to be called in that separately executing
     * thread.
     * 

* The general contract of the method run is that it may
     * take any action whatsoever.
     *
     * @see      java.lang.Thread#run()
     */
    public abstract void run();
}


```

함수형 인터페이스

- 람다 표현식을 사용하지 않고 함수형 인터페이스 사용 해보기

```
public static void java7Style() {  
    // 클래스 사용  
    RunnableTestClass runnable1 = new RunnableTestClass();  
  
    // 익명 클래스 사용  
    Runnable runnable2 = new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Hello World Anonymous Class");  
        }  
    };  
  
    process(runnable1);  
    process(runnable2);  
}
```

```
static class RunnableTestClass implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Hello World Class");  
    }  
}
```

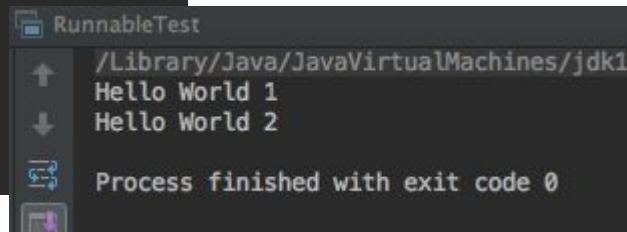
```
public static void process(Runnable r) {  
    r.run();  
}
```

```
RunnableTest  
↑ /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/...  
Hello World Class  
↓ Hello World Anonymous Class  
Process finished with exit code 0
```


함수형 인터페이스

- 람다 표현식으로 함수형 인터페이스 사용 해보기

```
public static void java8Style() {  
    // 람다 사용  
    Runnable runnable = () -> System.out.println("Hello World 1");  
  
    process(runnable);  
    // 직접 전달된 람다 사용  
    process(() -> System.out.println("Hello World 2"));  
}  
  
public static void process(Runnable r) {  
    r.run();  
}
```



RunnableTest

/Library/Java/JavaVirtualMachines/jdk1

Hello World 1

Hello World 2

Process finished with exit code 0

함수형 인터페이스 사용

- `java.util.function` 패키지로 여러가지 함수형 인터페이스를 제공한다.
 - Predicate, Consumer, Function 등



함수형 인터페이스 사용 - Predicate

- test 라는 추상 메서드를 정의
- T 객체를 파라미터로 받아 Boolean을 반환 한다.

```
public class PredicateTest {  
  
    public static void main(String args[]) {  
        List<String> listOfStrings = Arrays.asList("", "Hello", " ", "World", "!!", "");  
  
        Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();  
        List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);  
        System.out.println(nonEmpty);  
    }  
  
    public static <T> List<T> filter(List<T> list, Predicate<T> p) {  
        List<T> results = new ArrayList<>();  
        for (T s : list) {  
            if (p.test(s)) {  
                results.add(s);  
            }  
        }  
  
        return results;  
    }  
}
```

```
PredicateTest  
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jd  
[Hello,  , World, !!]  
  
Process finished with exit code 0
```

함수형 인터페이스 사용 - Consumer

- accept 라는 추상 메서드를 정의
- T 객체를 파라미터로 받아 void를 반환한다.

```
public class ConsumerTest {  
  
    public static void main(String args[]) {  
        forEach(Arrays.asList(1, 2, 3, 4, 5), (Integer i) -> System.out.println(i));  
    }  
  
    public static <T> void forEach(List<T> list, Consumer<T> c) {  
        for (T i : list) {  
            c.accept(i);  
        }  
    }  
}
```

ConsumerTest

/Library/Java/JavaVirtualMachines/jdk1.8.0_65.j

1
2
3
4
5

Process finished with exit code 0

함수형 인터페이스 사용 - Function

- apply 라는 추상 메서드를 정의
- T를 파라미터로 받아 R 객체를 반환한다.

```
public class FunctionTest {  
    public static void main(String args[]) {  
        List<Integer> list = map(  
            Arrays.asList("lambdas", "in", "action"),  
            (String s) -> s.length()  
        );  
  
        System.out.println(list);  
    }  
  
    public static <T, R> List<R> map(List<T> list, Function<T, R> f) {  
        List<R> result = new ArrayList<R>();  
        for (T s : list) {  
            result.add(f.apply(s));  
        }  
  
        return result;  
    }  
}
```

```
FunctionTest  
/Library/Java/JavaVirtualMachines/jdk1.8.  
[7, 2, 6]  
Process finished with exit code 0
```

기본형 특화 함수 인터페이스

- 자바의 모든 형식은 참조형(byte, Integer, Object, List 등) 아니면 기본형 (int, double, byte, char) 이다.
- 기본형을 참조형으로 변환 동작을 박싱(boxing)이라고 한다.
- 참조형을 기본형으로 변환 동작을 언박싱(unboxing)이라고 한다.
- 박싱과 언박싱이 자동으로 이루어지는 오토박싱(autoboxing)이라고 한다.
- 박싱, 언박싱, 오토박싱 과정은 비용이 소모 된다.
- 오토박싱에 의한 비용을 줄이기 위해 기본형 특화 함수형 인터페이스를 제공한다.

기본형 특화 함수 인터페이스

- 기본형 특화 함수 인터페이스 사용

```
public static void main(String args[]) {  
    // 박싱  
    Predicate<Integer> oddNumber = (Integer i) -> i % 2 == 1;  
    System.out.println(oddNumber.test(1000));  
  
    // 박싱 없음  
    IntPredicate evenNumber = (int i) -> i % 2 == 0;  
    System.out.println(evenNumber.test(1000));  
}
```

기본형 특화 함수 인터페이스

- 자바8 대표적인 함수 인터페이스

함수형 인터페이스	기본형 특화 함수형 인터페이스
Predicate<T>	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>

람다 캡처링

- 람다 표현식에서 파라미터로 넘겨진 변수가 아닌 외부에 정의된 변수를 활용할 수 있다.
 - 지역 변수는 `final` 로 선언되거나 `final` 처럼 취급 되어야한다.

```
public class LambdaCapturingTest {  
    public static void main(String[] args) {  
        int portNumber = 1337;  
        Runnable runnable = () -> System.out.println(portNumber);  
        runnable.run();  
    }  
}
```

람다 활용 : 실행 어라운드 패턴

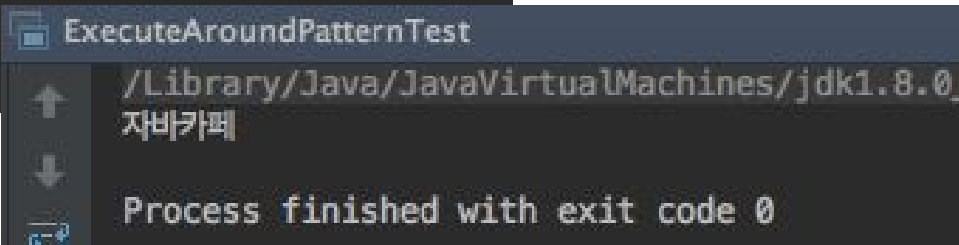
- 자원 처리에 사용하는 순환패턴은 자원을 열고, 처리한 다음에, 자원을 닫는 순서로 이루어진다.
- 설정(**setup**)과 정리(**cleanup**) 과정은 대부분 비슷하다.
- 실제 자원을 처리하는 코드를 설정과 정리 두 과정에 둘러싸는 형태를 갖는다.



람다 활용 : 실행 어라운드 패턴

- 파일에서 한 행을 읽는 코드

```
public class ExecuteAroundPatternTest {  
    public static void main(String args[]) throws IOException {  
        System.out.println(processFile());  
    }  
  
    public static String processFile() throws IOException {  
        try (BufferedReader bufferedReader = new BufferedReader(new FileReader("data.txt"))) {  
            return bufferedReader.readLine();  
        }  
    }  
}
```



```
ExecuteAroundPatternTest  
↑ /Library/Java/JavaVirtualMachines/jdk1.8.0_...  
자바카페  
↓  
Process finished with exit code 0
```

- 한번에 두줄을 읽는 요구 사항 추가!!!

람다 활용 : 실행 어라운드 패턴

- 함수형 인터페이스를 이용하여 동작 전달
 - 함수형 인터페이스 추가

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

- 동작을 파라미터로 전달

```
public class ExecuteAroundPatternTest {

    public static void main(String args[]) throws IOException {
        System.out.println(processFile(b -> b.readLine()));
    }

    public static String processFile(BufferedReaderProcessor processor) throws IOException {
        try (BufferedReader bufferedReader = new BufferedReader(new FileReader("data.txt"))) {
            return processor.process(bufferedReader);
        }
    }
}
```

람다 활용 : 실행 어라운드 패턴

- 한번에 두줄 읽는 기능 추가

```
public class ExecuteAroundPatternTest {  
    public static void main(String args[]) throws IOException {  
        System.out.println(processFile(b -> b.readLine()));  
        System.out.println(processFile(b -> b.readLine() + b.readLine()));  
    }  
  
    public static String processFile(BufferedReaderProcessor processor) throws IOException {  
        try (BufferedReader bufferedReader = new BufferedReader(new FileReader("data.txt"))) {  
            return processor.process(bufferedReader);  
        }  
    }  
}
```

ExecuteAroundPatternTest

↑ /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin
자바카페
↓ 자바카페 자바카페 페이스북 주소 : <https://www.facebook.com/groups/javacafe/>
Process finished with exit code 0

메서드 레퍼런스

```
public class MethodReferencesTest {  
    private static List<Apple> inventory;  
  
    public static void main(String args[]) {  
        inventory = new ArrayList<>();  
        inventory.add(new Apple(80, "green"));  
        inventory.add(new Apple(170, "red"));  
        inventory.add(new Apple(120, "red"));  
        inventory.add(new Apple(155, "green"));  
  
        inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));  
    }  
}
```

```
public class MethodReferencesTest {  
    private static List<Apple> inventory;  
  
    public static void main(String args[]) {  
        inventory = new ArrayList<>();  
        inventory.add(new Apple(80, "green"));  
        inventory.add(new Apple(170, "red"));  
        inventory.add(new Apple(120, "red"));  
        inventory.add(new Apple(155, "green"));  
  
        inventory.sort(Comparator.comparing(Apple::getWeight));  
    }  
}
```

메서드 레퍼런스

- 메서드 레퍼런스는 람다 표현식을 축약한 것
- 람다 표현식보다 메서드 레퍼런스를 사용하는 것이 가독성이 좋다.
- 메서드명 앞에 구분자(::)를 붙이는 방식으로 메서드 레퍼런스를 활용 할 수 있다.
- 실제 메서드를 호출하는 것이 아니므로 괄호는 필요 없음

람다 표현식	메서드 레퍼런스
(Apple a) -> a.getWeight()	Apple::getWeight
() -> Thread.currentThread().dumpstack()	Thread.currentThread()::dumpStack
(str, i) -> str.substring(i)	String::substring
(String s) -> System.out.println(s)	System.out::println

Predicate 조합

- Predicate 인터페이스는 negate, and, or 메서드를 제공한다.

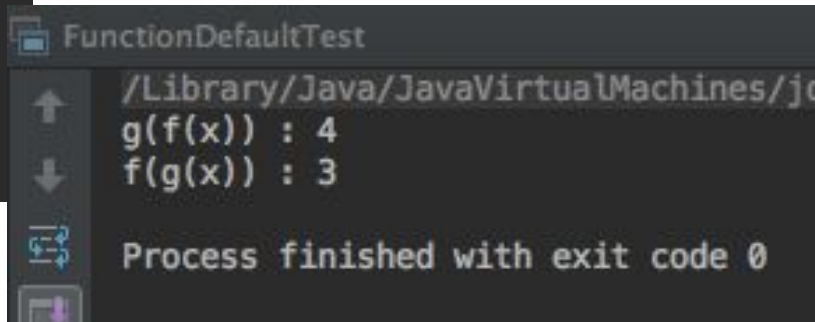
```
public static void main(String args[]) {  
    inventory = new ArrayList<>();  
    inventory.add(new Apple(80, "green"));  
    inventory.add(new Apple(155, "green"));  
    inventory.add(new Apple(120, "red"));  
    inventory.add(new Apple(170, "red"));  
  
    Predicate<Apple> redApple = (Apple apple) -> apple.getColor().equals("red");  
    System.out.println(filterApples(inventory, redApple));  
  
    Predicate<Apple> notRedApple = redApple.negate();  
    System.out.println(filterApples(inventory, notRedApple));  
  
    Predicate<Apple> redAndHeavyApple = redApple.and(a -> a.getWeight() > 150);  
    System.out.println(filterApples(inventory, redAndHeavyApple));  
  
    Predicate<Apple> redOrHeavyApple = redApple.or(a -> a.getWeight() > 150);  
    System.out.println(filterApples(inventory, redOrHeavyApple));  
  
    Predicate<Apple> redAndHeavyAppleOrGreen = redApple  
        .and(a -> a.getWeight() > 150)  
        .or(a -> a.getColor().equals("green"));  
    System.out.println(filterApples(inventory, redAndHeavyAppleOrGreen));  
}
```

```
PredicateDefaultTest  
/Library/Java/JavaVirtualMachines/jdk1  
[(red, 120), (red, 170)]  
[(green, 80), (green, 155)]  
[(red, 170)]  
[(green, 155), (red, 120), (red, 170)]  
[(green, 80), (green, 155), (red, 170)]  
Process finished with exit code 0
```


Function 조합

- Function 인터페이스는 `andThen`, `compose` 두 가지 디폴트 메서드를 제공한다.

```
public class FunctionDefaultTest {  
  
    public static void main(String[] args) {  
        Function<Integer, Integer> f = x -> x + 1;  
        Function<Integer, Integer> g = x -> x * 2;  
  
        // g(f(x))  
        Function<Integer, Integer> h1 = f.andThen(g);  
        System.out.println("g(f(x)) : " + h1.apply(1));  
  
        // f(g(x))  
        Function<Integer, Integer> h2 = f.compose(g);  
        System.out.println("f(g(x)) : " + h2.apply(1));  
    }  
}
```



```
FunctionDefaultTest  
/Library/Java/JavaVirtualMachines/jc  
g(f(x)) : 4  
f(g(x)) : 3  
  
Process finished with exit code 0
```

요약

- 람다 표현식은 익명 함수의 일종이다.
- 람다 표현식으로 간결한 코드를 구현할 수 있다.
- 함수형 인터페이스는 하나의 추상 메서드만을 정의하는 인터페이스다.
- 함수형 인터페이스를 기대하는 곳에서만 람다 표현식을 사용할 수 있다.
- `java.util.function` 패키지는 `Predicate<T>`, `Function<T, R>`, `Consumer<T>` 등을 포함해서 자주 사용하는 다양한 함수형 인터페이스를 제공한다.
- 박싱 동작을 피할 수 있도록 `IntPredicate`, `IntToLongFunction` 등과 같은 기본형 특화 인터페이스를 제공한다.
- 실행 어라운드 패턴을 람다와 활용하면 유연성과 재사용성을 추가로 얻을 수 있다.
- 메서드 레퍼런스를 이용하면 기존의 메서드 구현을 재사용하고 직접 전달할 수 있다.

스트림 API

스트림이란 무엇인가?

- Java 7

```
public static void java7Style() {
    List<Dish> lowCaloricDishes = new ArrayList<>();
    for (Dish d : DishList.getDishList()) {
        if (d.getCalories() < 400) {
            lowCaloricDishes.add(d);
        }
    }

    Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
        @Override
        public int compare(Dish o1, Dish o2) {
            return Integer.compare(o1.getCalories(), o2.getCalories());
        }
    });

    List<String> lowCaloricDishesName = new ArrayList<>();
    for (Dish d : lowCaloricDishes) {
        lowCaloricDishesName.add(d.getName());
    }
}
```

- 스트림 사용

```
public static void java8Style() {
    List<String> lowCaloricDishesName =
        DishList.getDishList().stream()
            .filter(d -> d.getCalories() < 400)
            .sorted(Comparing(Dish::getCalories))
            .map(Dish::getName)
            .collect(Collectors.toList());
}
```

스트림이란 무엇인가?

- 데이터 컬렉션 반복을 멋지게 처리할 수 있다.
- 컬렉션을 **SQL Query** 처럼 처리 할 수 있다.
 - ex) 400칼로리 이상의 요리를 찾기
 - `SELECT name FROM dishes WHERE calorie > 400`
 - `dishes.stream().filter(d -> d.getCalories() < 400).map(Dish::getName).collect(toList())`
- 멀티 스레드 코드를 구현하지 않아도 데이터를 투명하게 병렬로 처리할 수 있다.

스트림이란 무엇인가?

- 선언형
 - 루프와 if 조건문 등의 제어 블록을 사용하여 어떻게 동작 해야하는지 지정할 필요가 없다.
 - 간결하고 가독성이 좋아진다.
- 조립할 수 있음
 - `filter`, `sorted`, `map`, `collect` 같은 연산을 연결하여 처리를 할 수 있다.
 - 유연성이 좋아진다.
- 병렬화
 - 성능이 좋아진다.

스트림 시작하기

- **filter**
 - 람다를 인수로 받아 스트림에서 특정 요소를 제외 시킨다.
- **map**
 - 람다를 이용해서 한 요소를 다른 요소로 변환하거나 정보를 추출한다.
- **limit**
 - 정해진 개수 이상의 요소가 스트림에 저장되지 못하게 스트림 크기를 축소한다.
- **sorted**
 - 람다를 이용하여 스트림 요소를 정렬한다.
- **distinct**
 - 스트림에 저장되어 있는 중복 요소를 제거 한다.

스트림 시작하기

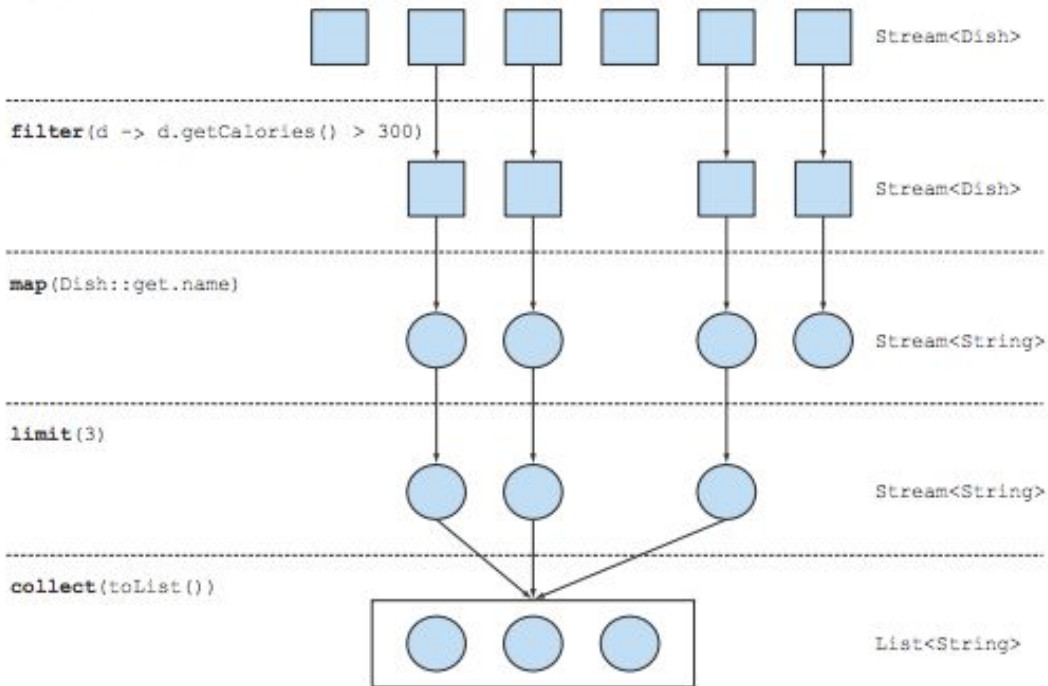
- **collect**
 - 스트림을 다른 형식으로 변환한다.
- **forEach**
 - 스트림의 각 요소를 소비하면서 람다를 적용한다.
- **count**
 - 스트림의 요소의 개수를 반환한다.

스트림 시작하기

- 스트림에서 메뉴를 필터링해서 세 개의 고칼로리 요리명 찾기

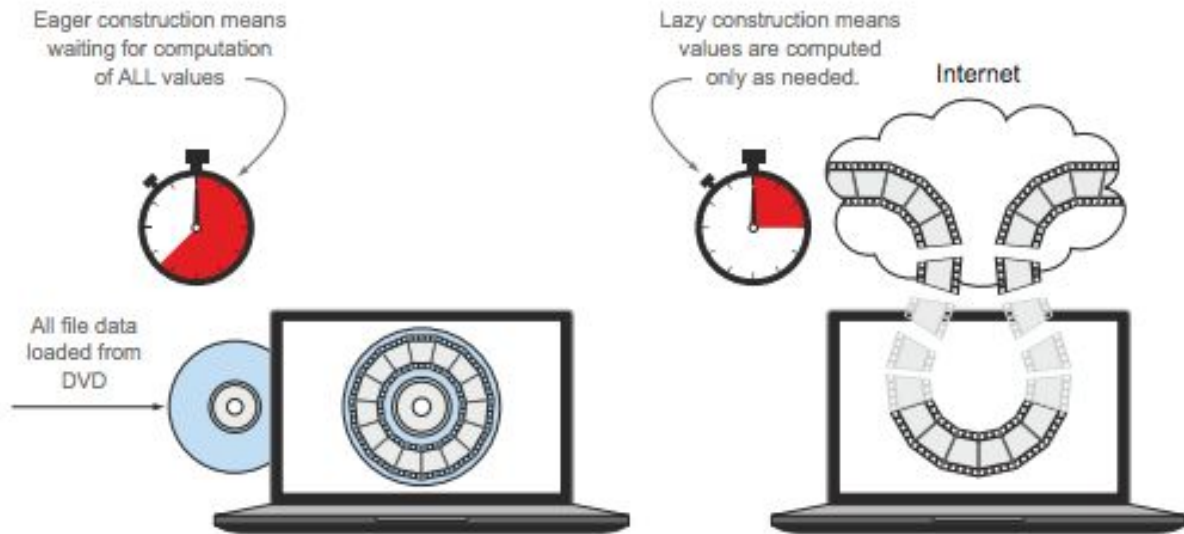
```
List<String> threeHighCaloricDishNames =  
    DishList.getDishList().stream()  
        .filter(d -> d.getCalories() > 300)  
        .map(Dish::getName)  
        .limit(3)  
        .collect(Collectors.toList());
```

Menu stream



스트림과 컬렉션

- 컬렉션
 - 현재 자료구조가 포함하는 모든 값을 메모리에 저장하는 자료구조
- 스트림
 - 요청할 때만 요소를 계산하는 고정된 자료구조



스트림과 컬렉션 - 딱 한번만 탐색할 수 있다

- 한번 탐색한 요소를 다시 탐색하려면 초기 데이터 소스에서 새로운 스트림을 만들어야 한다.

```
public class StreamCollectionTest {  
  
    public static void main(String[] args){  
        List<String> title = Arrays.asList("Java8", "In", "Action");  
        Stream<String> stream = title.stream();  
        stream.forEach(System.out::println);  
        stream.forEach(System.out::println);  
    }  
  
}
```



```
StreamCollectionTest  
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java ...  
Java8  
In  
Action  
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed  
    at java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279) <1 internal calls>  
    at kr.javacafe.stream.StreamCollectionTest.main(StreamCollectionTest.java:16) <5 internal calls>  
  
Process finished with exit code 1
```

스트림과 컬렉션 - 외부 반복과 내부 반복

- 외부 반복

- 컬렉션 인터페이스를 사용하려면 사용자가 직접 요소를 반복 해야한다.

```
public static List<String> foreach() {  
    List<String> names = new ArrayList<>();  
    for (Dish d : DishList.getDishList()) {  
        names.add(d.getName());  
    }  
    return names;  
}
```

- 내부 반복

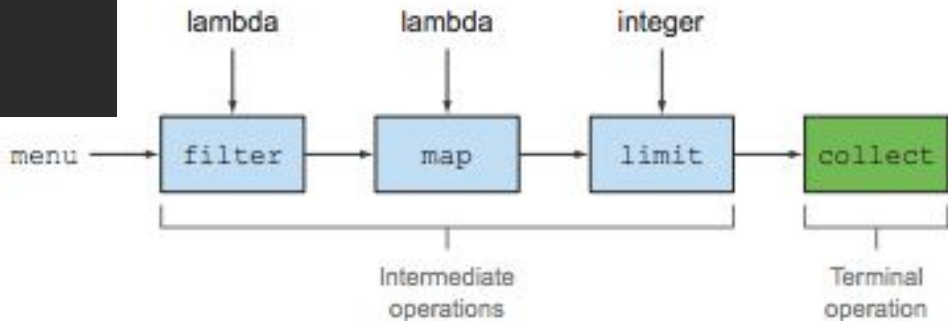
- 함수에 어떤 작업을 수행할지만 지정하면 모든 것이 알아서 처리된다.

```
public static List<String> streamLoop() {  
    List<String> names = DishList.getDishList()  
        .stream()  
        .map(Dish::getName)  
        .collect(toList());  
    return names;  
}
```

스트림 연산

- 스트림 인터페이스의 연산을 크게 중간 연산과 최종 연산으로 구분 할 수 있다.

```
public class StreamOperationsTest {  
    public static void main(String[] args) {  
        streamOperation();  
    }  
  
    public static void streamOperation() {  
        List<String> names = DishList.getDishList().stream()  
            .filter(d -> d.getCalories() > 300)  
            .map(Dish::getName)  
            .limit(3)  
            .collect(Collectors.toList());  
  
        System.out.println(names);  
    }  
}
```



스트림 연산

- 중간 연산을 합친 다음에 합쳐진 중간 연산을 최종 연산으로 한번에 처리 한다.

```
public class StreamOperationsTest {

    public static void main(String[] args) {
        streamLog();
    }

    public static void streamLog() {
        List<String> names = DishList.getDishList().stream()
            .filter(d -> {
                System.out.println("filter : " + d.getName());
                return d.getCalories() > 300;
            })
            .map(d -> {
                System.out.println("map : " + d.getName());
                return d.getName();
            })
            .limit(3)
            .collect(Collectors.toList());

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

```
StreamOperationsTest  
/Library/Java/JavaVirtualMachines/jdk-8.0.60-jre/  
filter : pork  
map : pork  
filter : beef  
map : beef  
filter : chicken  
map : chicken  
[pork, beef, chicken]  
  
Process finished with exit code 0
```

중간 연산

- 최종 연산을 수행하기 전까지는 아무 연산도 수행하지 않는다.
- 중간 연산을 합친 다음에 합쳐진 중간 연산을 최종 연산으로 한 번에 처리 한다.

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

최종 연산

- 스트림 파이프라인에서 결과를 도출 한다.
- 최종 연산에 의해 **List**, **Integer** **void** 등 스트림 이외의 결과가 반환된다.

Operation	Type	Purpose
<code>forEach</code>	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns <code>void</code> .
<code>count</code>	Terminal	Returns the number of elements in a stream. The operation returns a <code>long</code> .
<code>collect</code>	Terminal	Reduces the stream to create a collection such as a <code>List</code> , a <code>Map</code> , or even an <code>Integer</code> . See chapter 6 for more detail.

퀴즈 - 중간 연산과 최종 연산

- 다음 스트림 파이프라인에서 중간 연산과 최종 연산을 구별하시오.

```
long count = menu.stream()  
    .filter( d -> d.getCalories() > 300)  
    .distinct()  
    .limit(3)  
    .count();
```

퀴즈 정답

- 중간 연산
 - Filter, distinct, limit
- 최종 연산
 - count

요약

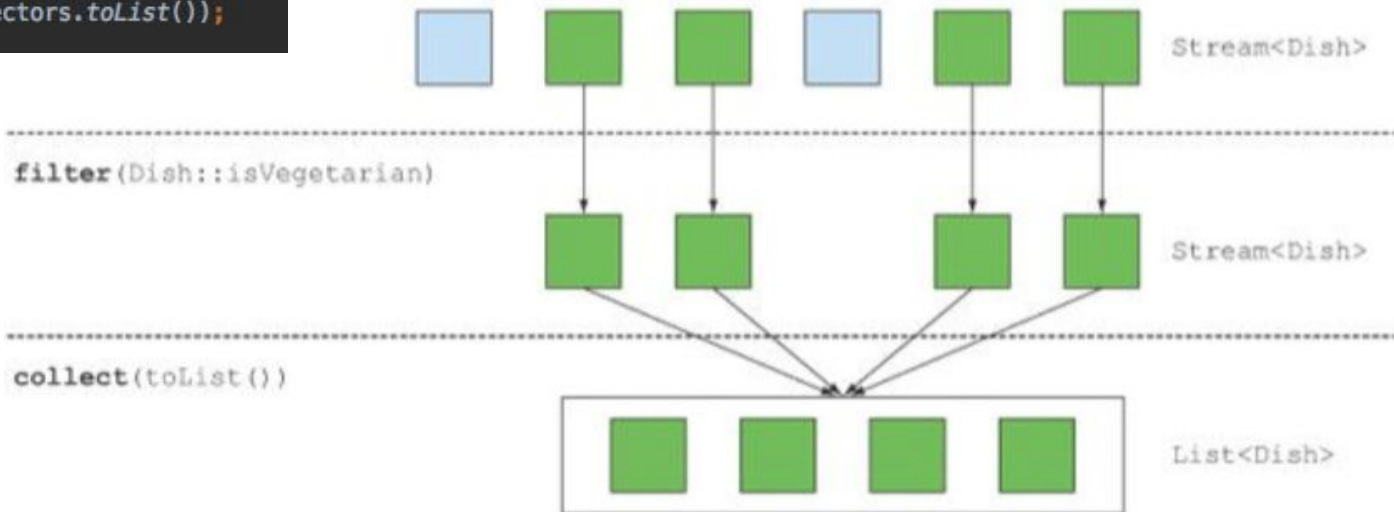
- 스트림은 소스에서 추출된 연속된 요소로, 데이터 처리 연산을 지원한다.
- 스트림은 내부 반복을 지원한다.
- 스트림에는 중간 연산과 최종 연산이 있다.
- 중간 연산을 이용해서 파이프라인을 구성할 수 있지만 중간 연산으로는 어떤 결과도 생성할 수 없다.
- 스트림의 요소는 요청할 때만 계산된다.

스트림 API 활용

필터링과 슬라이싱

- 프레디케이트로 필터링
 - 프레디케이트와 일치하는 모든 요소를 포함하는 스트림을 반환한다.

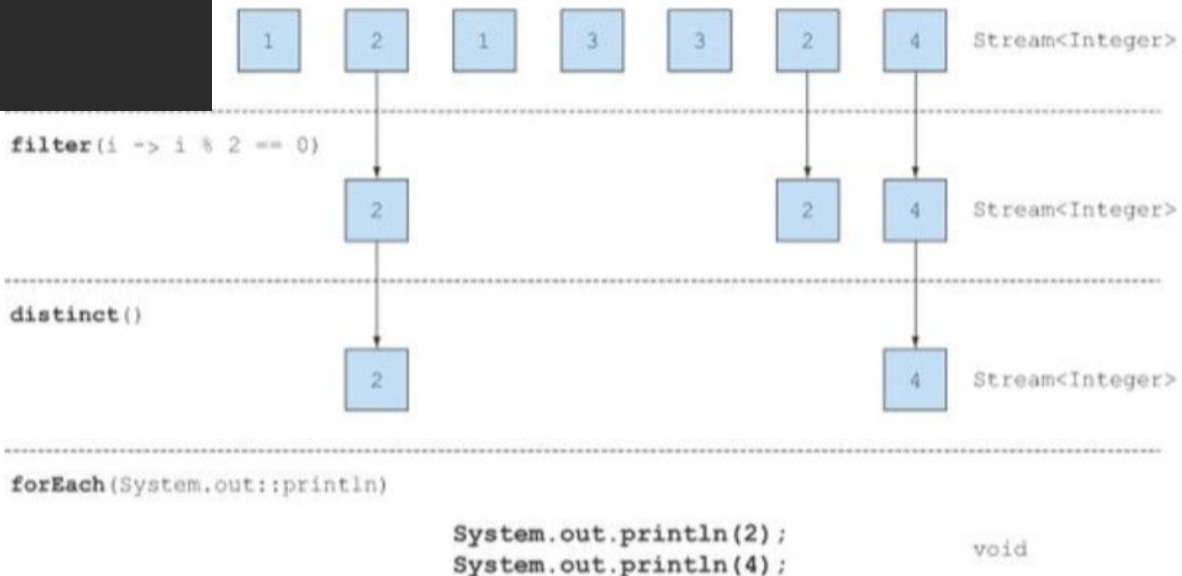
```
List<Dish> vegetarianMenu =  
    DishList.getDishList().stream()  
        .filter(Dish::isVegetarian)  
        .collect(Collectors.toList());
```



필터링과 슬라이싱

- 고유 요소 필터링
 - 고유 요소로 이루어진 스트림을 반환하는 `distinct`라는 메서드를 지원한다. (중복제거)

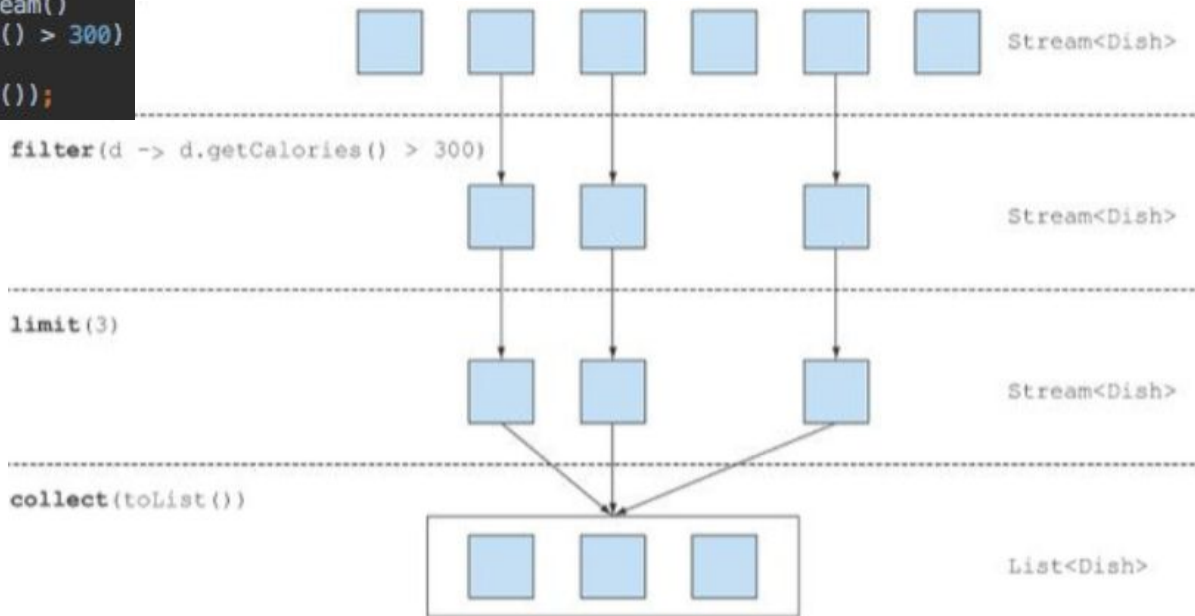
```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
    .filter(i -> i % 2 == 0)
    .distinct()
    .forEach(System.out::println);
```



필터링과 슬라이싱

- 스트림 축소
 - 주어진 사이즈 이하의 크기를 갖는 새로운 스트림을 반환한다.
 - `limit(n)` 을 호출하면 처음 `n`개 요소를 선택한 다음 즉시 결과를 반환한다.

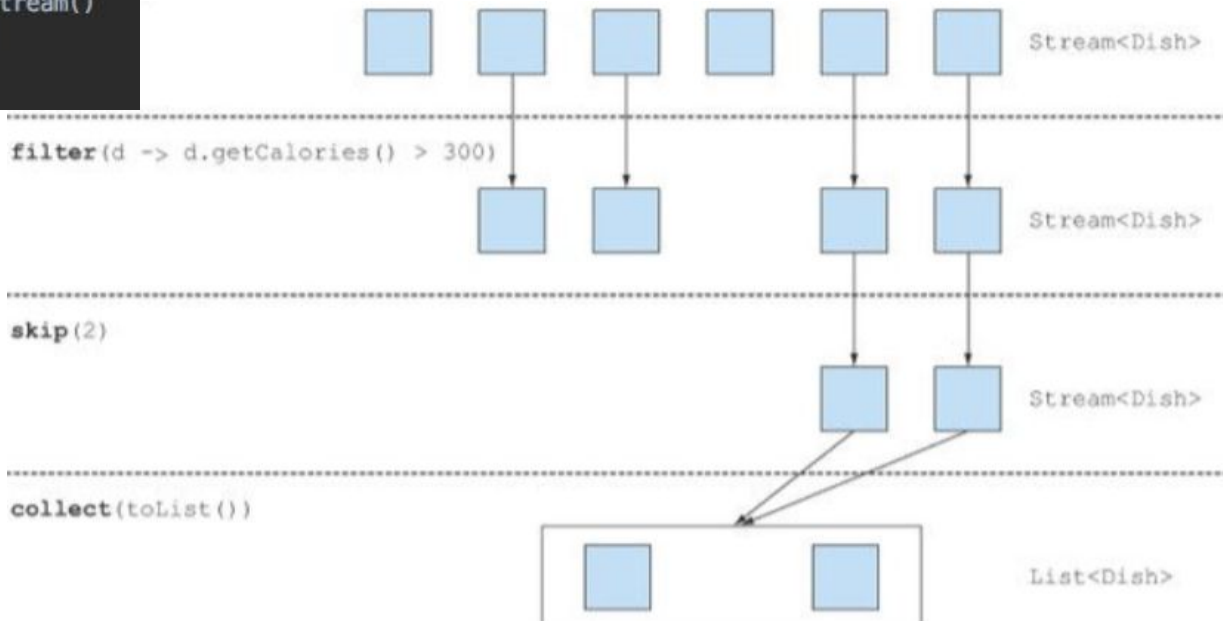
```
List<Dish> dishes = DishList.getDishList().stream()
    .filter(d -> d.getCalories() > 300)
    .limit(3)
    .collect(Collectors.toList());
```



필터링과 슬라이싱

- 요소 건너뛰기
 - 처음 n 개 요소를 제외한 스트림을 반환하는 `skip(n)` 메서드를 지원한다.
 - n 개 이하의 요소를 포함하는 스트림에 `skip(n)`을 호출하면 빈 스트림이 반환된다.

```
List<Dish> dishes = DishList.getDishList().stream()  
    .filter(d -> d.getCalories() > 300)  
    .skip(2)  
    .collect(Collectors.toList());
```



퀴즈

- 스트림을 이용해서 처음 등장하는 두 고기 요리를 필터링하시오.

퀴즈

- 스트림을 이용해서 처음 등장하는 두 고기 요리를 필터링하시오.

- 정답 :

```
List<Dish> dishes = menu.stream()  
    .filter(d -> d.getType() == Dish.Type.MEAT)  
    .limit(2)  
    .collect(toList());
```

매핑

- 스트림의 각 요소에 함수 적용하기
 - 스트림은 함수를 인수로 받는 `map` 메서드를 지원한다.
 - 인수로 받은 함수를 적용한 결과가 새로운 요소로 매핑된다.
- ex) `Dish::getName`을 `map` 메서드로 전달해서 요리명을 추출하는 코드

```
List<String> dishNames = DishList.getDishList().stream()
    .map(Dish::getName)
    .collect(toList());
```

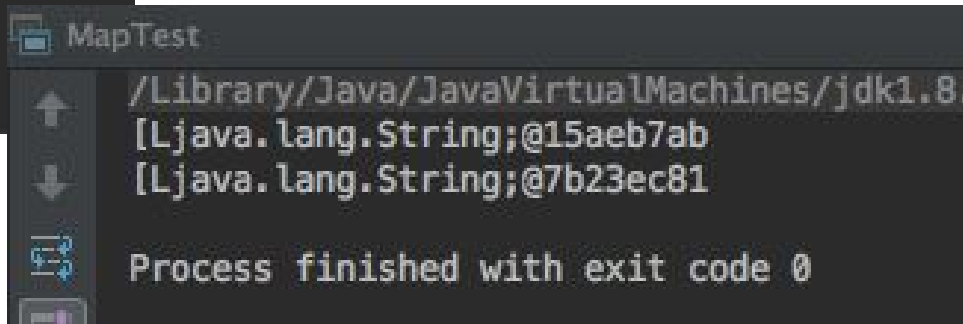
- ex) 다른 `map` 메서드를 연결할 수 있다.

```
List<Integer> dishNameLength = DishList.getDishList().stream()
    .map(Dish::getName)
    .map(String::length)
    .collect(toList());
```

매핑

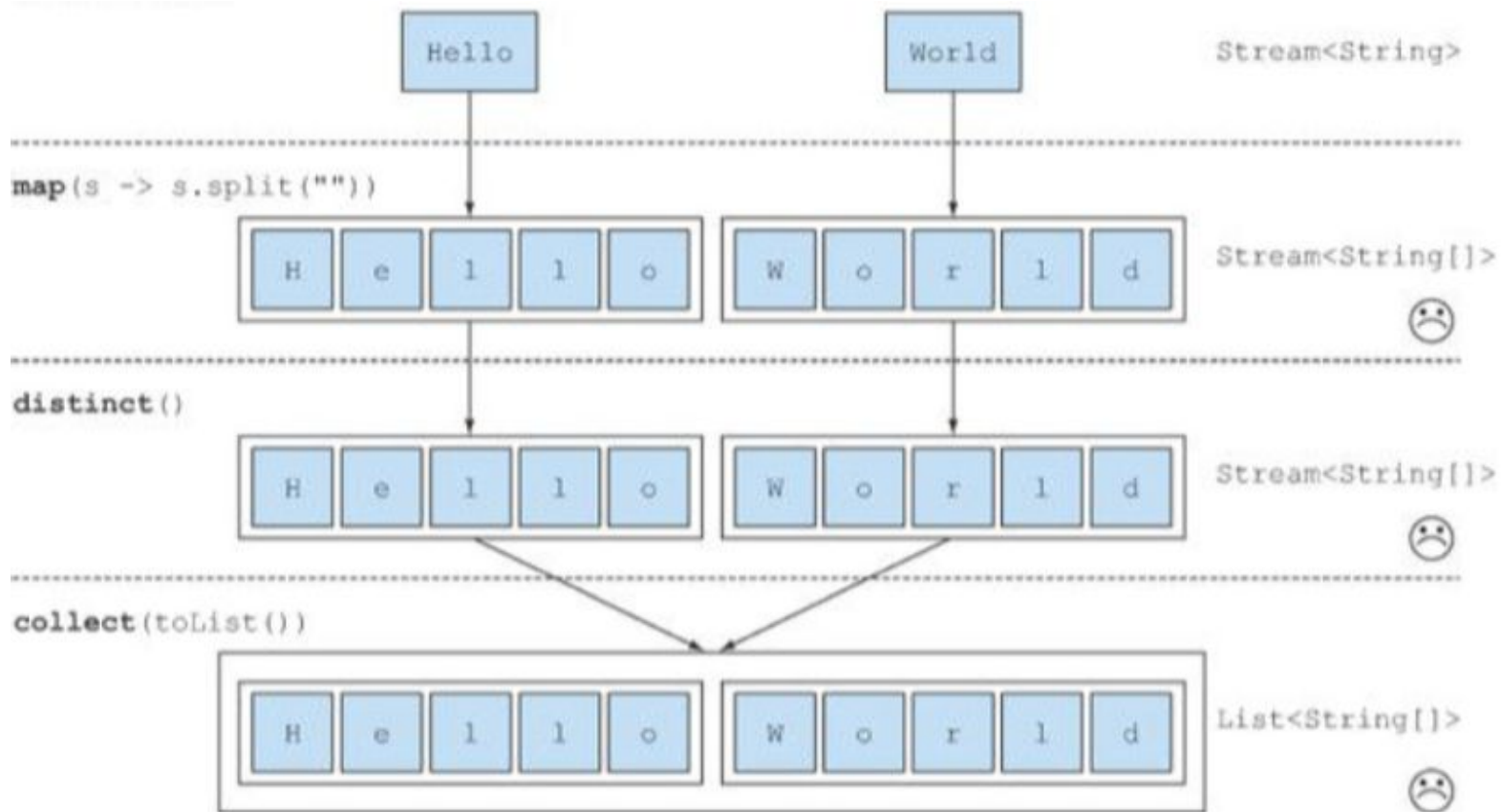
- 스트림 평면화
 - flatMap은 각 배열을 스트림이 아니라 스트림의 콘텐츠로 매핑한다.
- ex) ["Hello", "World"] 리스트를 ["H", "e", "l", "O", "W", "r", "d"] 로 변환해보자

```
private static void uniqueCharacters() {  
    List<String> words = Arrays.asList("Hello", "World");  
    words.stream()  
        .map(w -> w.split(""))  
        .distinct()  
        .forEach(System.out::println);  
}
```



```
MapTest  
/Library/Java/JavaVirtualMachines/jdk1.8.  
[Ljava.lang.String;@15aeb7ab  
[Ljava.lang.String;@7b23ec81  
Process finished with exit code 0
```

매핑



매핑

- flatMap 사용하기

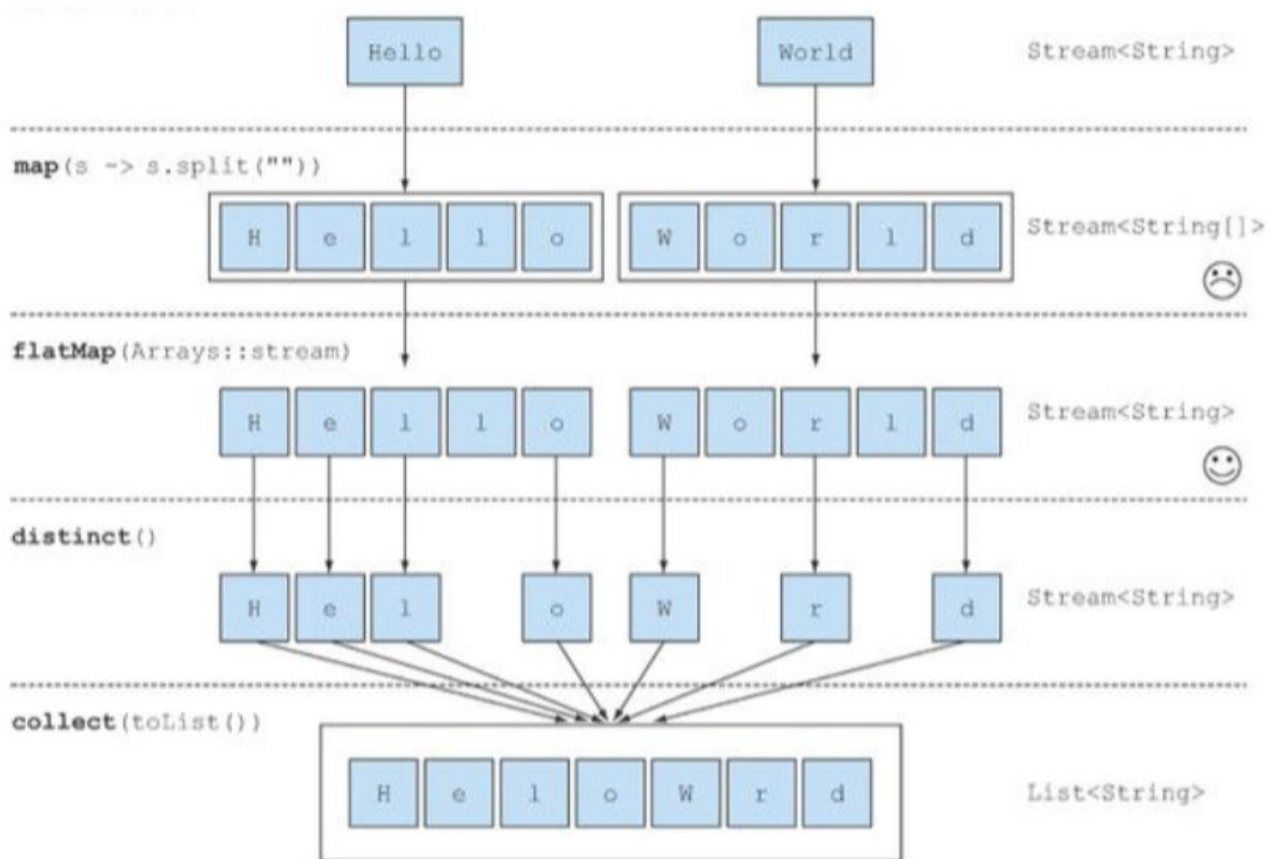
```
private static void uniqueCharactersFlatMap() {  
    List<String> words = Arrays.asList("Hello", "World");  
    List<String> uniqueCharacters = words.stream()  
        .map(w -> w.split(""))  
        .flatMap(Arrays::stream)  
        .distinct()  
        .collect(toList());  
  
    System.out.println(uniqueCharacters);  
}
```

MapTest

/Library/Java/JavaVirtualMachines/jdk1.8.0_101-b14/jre/bin/java -Djava.class.path=. MapTest
[H, e, l, o, W, r, d]

Process finished with exit code 0

매핑



map과 Arrays.stream 활용

- 배열 스트림 대신 문자열 스트림이 필요할 경우
Arrays.stream을 사용할 수 있다.

```
String[] arrayOfWords = {"Goodbye", "World"};  
Stream<String> streamOfWords = Arrays.stream(arrayOfWords);
```

```
List<Stream<String>> streamOfhelloWorld = helloWorldWords.stream()  
    .map(word -> word.split(""))  
    .map(Arrays::stream)  
    .distinct()  
    .collect(Collectors.toList());
```


검색과 매칭

- **anyMatch**
 - 적어도 한 요소와 일치하는지 확일 할때 사용
- **allMatch**
 - 모든 요소가 주어진 프레디케이드와 일치하는지 검사
- **noneMatch**
 - 주어진 프레디케이와 일치하는 요소가 없는지 확인한다.

anyMach, allMatch, noneMatch는 스트림의 쇼트서킷 기법

검색과 매칭

```
public static void anyMatchTest() {
    if (DishList.getDishList().stream().anyMatch(Dish::isVegetarian)) {
        System.out.println("채식 메뉴가 존재 합니다.");
    } else {
        System.out.println("채식 메뉴가 없습니다.");
    }
}

private static void allMatchTest() {
    if (DishList.getDishList().stream().allMatch(d -> d.getCalories() < 1000)) {
        System.out.println("모든 메뉴가 1000 칼로리 이하입니다.");
    } else {
        System.out.println("모든 메뉴가 1000 칼로리 이하가 아닙니다.");
    }
}

private static void noneMatchTest() {
    if (DishList.getDishList().stream().noneMatch(d -> d.getCalories() >= 10000)) {
        System.out.println("모든 메뉴가 1000 칼로리 이상이 아닙니다.");
    } else {
        System.out.println("모든 메뉴가 1000 칼로리 이상입니다.");
    }
}
```

MatchingTest

/Library/Java/JavaVirtualMachines/jdk-11.0.2/bin/java -Djava.class.path=. MatchingTest
채식 메뉴가 존재 합니다.
모든 메뉴가 1000 칼로리 이하입니다.
모든 메뉴가 1000 칼로리 이상이 아닙니다.

쇼트서킷

- 모든 스트림을 처리하지 않고 결과를 반환 할 수 있다.
- 무한한 요소를 가진 스트림을 유한한 크기로 줄일 수 있는 유용한 연산
 - ex) `anyMatch` 에서 하나라도 일치하는 결과가 나오면 나머지 결과에 상관없이 **True**를 반환한다.

검색과 매칭

- **findAny**
 - 현재 스트림에서 임의 요소를 반환한다.
- **findFirst**
 - 현재 스트림에서 첫 번째 요소를 찾아 반환한다. (병렬 실행시 성능 이슈)
- **findFirst**와 **findAny**는 병렬성 때문에 사용 된다.
 - 병렬 실행에서는 첫 번째 요소를 찾기 어렵다.
반환 순서가 상관 없다면 병렬 스트림에서는 **findAny**를 사용하자.

검색과 매칭

```
public class FindTest {  
  
    public static void main(String[] args) {  
        findAnyTest();  
        findFirstTest();  
    }  
  
    private static void findFirstTest() {  
        List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);  
        someNumbers.stream()  
            .filter(x -> x % 3 == 0)  
            .findFirst()  
            .ifPresent(System.out::println);  
    }  
  
    private static void findAnyTest() {  
        Optional<Dish> dish = DishList.getDishList().stream()  
            .filter(Dish::isVegitarian)  
            .findAny();  
  
        if (dish.isPresent()) {  
            System.out.println("채식 메뉴 있음.");  
        } else {  
            System.out.println("채식 메뉴 없음.");  
        }  
    }  
}
```

FindTest

/Library/Java/JavaVirtualMachines/j
채식 메뉴 있음.

3

Process finished with exit code 0

Optional

- 값이 존재하는지 확인하고 값이 없을 때 어떻게 처리할 것인지 강제하는 기능을 제공한다
 - `isPresent` 는 `Optional`이 값을 포함하면 `True`를 반환하고, 값을 포함하지 않으면 `false`를 반환한다.
 - `ifPresent` 는 값이 있으면 주어진 블록을 실행한다.
 - `get` 은 값이 존재하면 반환하고, 값이 없으면 `NoSuchElementException`을 일으킨다.
 - `orElse` 는 값이 있으면 값을 반환하고, 값이 없으면 기본값을 반환한다.

리듀싱

- 모든 스트림 요소를 처리해서 값으로 도출하는 연산
 - 메뉴의 모든 칼로리 합계를 구하기
 - 메뉴의 가장 작은 칼로리 찾기
 - 메뉴의 가장 높은 칼로리 찾기
- 초기 값을 지정하지 않으면 **Optional** 객체 반환한다.
- 내부 반복이 추상화되면서 병렬로 **reduce**를 실행 할 수 있다.

리듀싱

```
public class ReducingTest {  
  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);  
        System.out.println("Number Count : " + numbers.stream().reduce(0, (a, b) -> a + b));  
        System.out.println("Number Count Method Reference : " + numbers.stream().reduce(0, Integer::sum));  
  
        Optional<Integer> sum1 = numbers.stream().reduce((a, b) -> a + b);  
        System.out.println("Number Count Optional<Integer> : " + sum1.get());  
  
        Optional<Integer> sum2 = numbers.stream().reduce(Integer::sum);  
        System.out.println("Number Count Method Reference Optional<Integer> : " + sum2.get());  
  
        System.out.println("Number Max Method Reference : " + numbers.stream().reduce(0, Integer::max));  
        System.out.println("Number Min Method Reference : " + numbers.stream().reduce(0, Integer::min));  
    }  
}
```

ReducingTest

/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Co

Number Count : 21

Number Count Method Reference : 21

Number Count Optional<Integer> : 21

Number Count Method Reference Optional<Integer> : 21

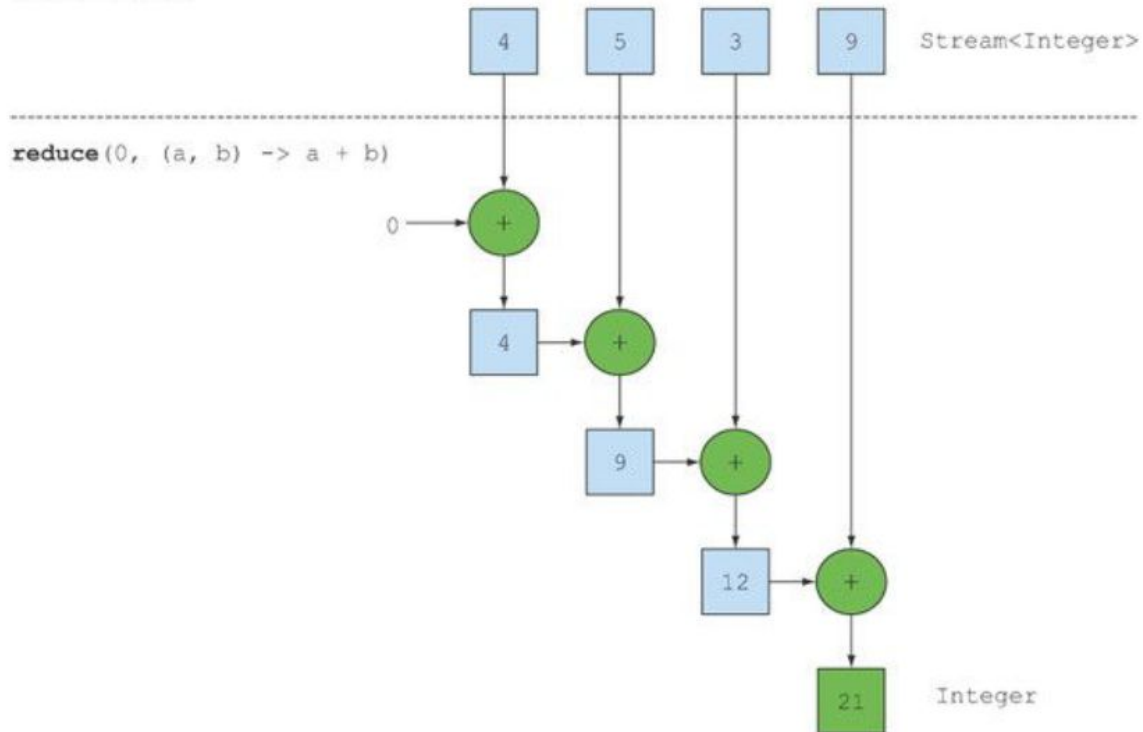
Number Max Method Reference : 6

Number Min Method Reference : 0

Process finished with exit code 0

리듀싱

Numbers stream



숫자형 스트림

- 기본형 특화 스트림
 - 스트림 API는 박싱비용을 피할 수 있도록 `IntStream`, `DoubleStream`, `LongStream`을 제공한다.
- 숫자 범위
 - `IntStream`과 `LongStream`은 `range`와 `rangeClose` 메서드를 제공한다.
 - `range` 메서드는 시작값과 종료값이 결과에 포함되지 않는다.
 - `rangeClose` 메서드는 시작값과 종료값이 결과에 포함된다.

스트림 만들기

- 값으로 스트림 만들기
 - 임의의 수를 인수로 받는 정적 메서드 **Stream.of**를 이용해서 스트림을 만들 수 있다.
 - **Stream.empty**를 이용해서 스트림을 비울 수 있다.
- 배열로 스트림 만들기
 - **Arrays.stream**을 이용해서 스트림을 만들 수 있다.
- 파일로 스트림 만들기
 - **java.nio.file.Files**를 이용해서 스트림을 만들 수 있다.
- 함수로 무한 스트림 만들기
 - **Stream.iterate**와 **Stream.generate**를 이용해서 무한 스트림을 만들 수 있다.
 - **Stream.iterate**는 기존의 결과에 의존해서 순차적으로 연산을 수행
 - **Stream.generate**는 새로운 값을 생산한다.
- **IntStream**을 이용하면 박싱 연산 문제를 해결 할 수 있다.

스트림 만들기

```
public class BuildingStream {  
    public static void main(String[] args) {  
        Stream<String> stream = Stream.of("Java 8", "Lambda", "In", "Action");  
        stream.map(String::toUpperCase).forEach(System.out::println);  
  
        int[] numbers = {2, 3, 5, 7, 11, 13};  
        int sum = Arrays.stream(numbers).sum();  
        System.out.println("Number Sum : " + sum);  
  
        Stream.iterate(0, n -> n + 2)  
            .limit(10)  
            .forEach(System.out::println);  
  
        Stream.generate(Math::random)  
            .limit(5)  
            .forEach(System.out::println);  
    }  
}
```

BuildingStream

/Library/Java/JavaVirtualMachines
JAVA 8
LAMBDA
IN
ACTION
Number Sum : 41
0
2
4
6
8
10
12
14
16
18
0.05127637399290852
0.7704987424523977
0.654988681269224
0.19715285900835344
0.8986935545491765

병렬 데이터 처리와 성능

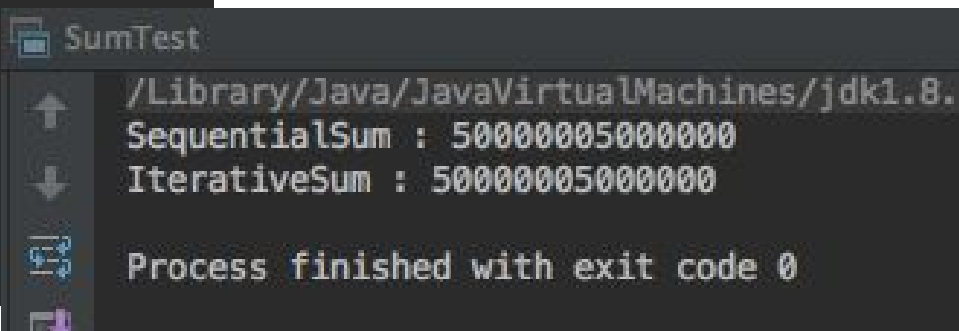
병렬 스트림

- 컬렉션에 `parallelStream`을 호출 하면 병렬 스트림이 생성된다.
- 병렬 스트림을 이용하면 모든 멀티코어 프로세서가 각가의 청크를 처리하도록 할당할 수 있다.

병렬 스트림

- 숫자 n 을 인수로 받아서 1부터 n 까지 모든 숫자의 합계를 구하는 코드

```
public class SumTest {  
  
    public static void main(String[] args) {  
        System.out.println("SequentialSum : " + sequentialSum(10_000_000));  
        System.out.println("IterativeSum : " + iterativeSum(10_000_000));  
    }  
  
    public static long sequentialSum(long n) {  
        return Stream.iterate(1L, i -> i + 1)  
            .limit(n)  
            .reduce(0L, Long::sum);  
    }  
  
    public static long iterativeSum(long n) {  
        long result = 0;  
        for (long i = 1L; i <= n; i++) {  
            result += i;  
        }  
  
        return result;  
    }  
}
```



```
SumTest  
/Library/Java/JavaVirtualMachines/jdk1.8.  
SequentialSum : 50000005000000  
IterativeSum : 50000005000000  
  
Process finished with exit code 0
```

병렬 스트림

- 숫자 n 이 커진다면 연산을 병렬로 처리하는 것이 좋을까?
- 병렬로 처리하기 위해서 어디를 수정 해야할까?
- 결과 변수는 어떻게 동기화 해야할까?
- 몇 개의 스레드를 사용해야할까?
- 숫자는 어떻게 생성할까?
- 생성된 숫자는 누가 더할까?

순차 스트림을 병렬 스트림으로 변환하기

- 순차 스트림에 `parallel` 메서드를 호출 하면 기존의 함수형 리듀싱 연산이 병렬로 처리된다.

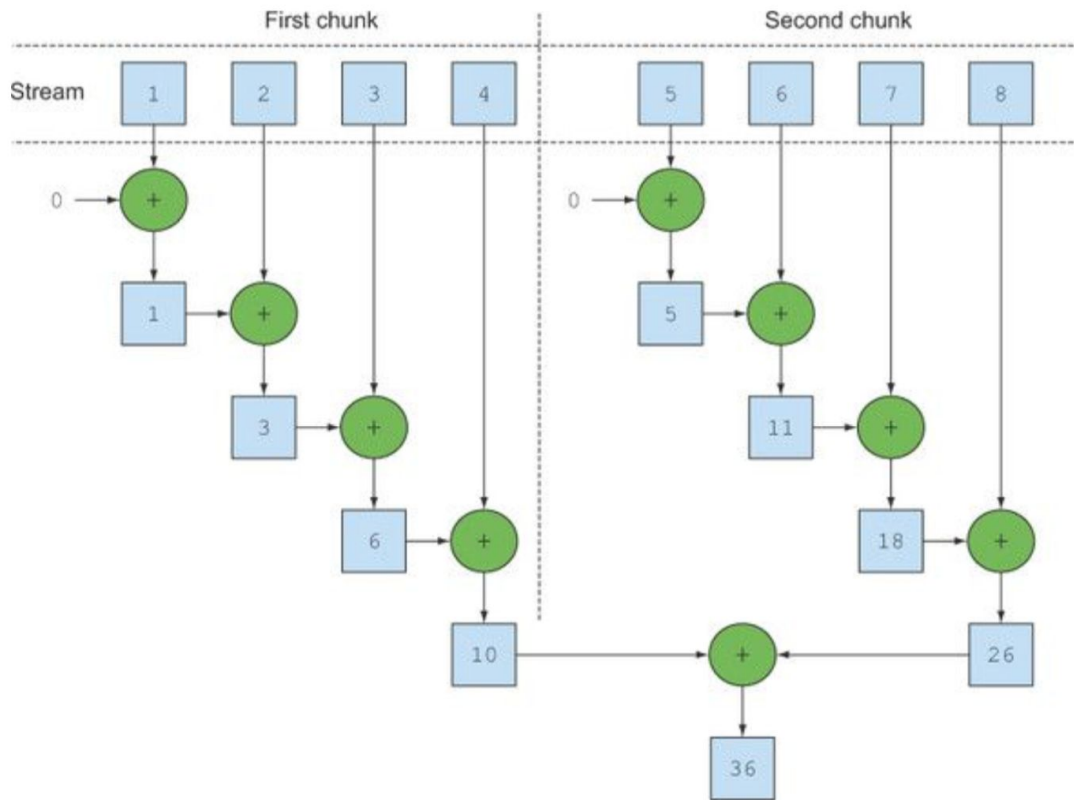
```
public class SumTest {  
  
    public static void main(String[] args) {  
        System.out.println("SequentialSum : " + sequentialSum(10_000_000));  
    }  
  
    public static long sequentialSum(long n) {  
        return Stream.iterate(1L, i -> i + 1)  
            .limit(n)  
            .reduce(0L, Long::sum);  
    }  
}
```

병렬 스트림

->

```
public class SumTest {  
  
    public static void main(String[] args) {  
        System.out.println("ParallelSum : " + parallelSum(10_000_000));  
    }  
  
    public static long parallelSum(long n) {  
        return Stream.iterate(1L, i -> i + 1)  
            .parallel()  
            .limit(n)  
            .reduce(0L, Long::sum);  
    }  
}
```

순차 스트림을 병렬 스트림으로 변환하기



스트림 성능 측정

- 병렬화를 이용하면 순차나 반복 형식에 비해 성능이 더 좋아질 것이라 추측했다.

```
public class PerformanceTest {  
  
    public static void main(String[] args) {  
        System.out.println("Sequential sum done in : " + measureSumPerformance(SumTest::sequentialSum, 10_000_000) + " msecs");  
        System.out.println("Iterative sum done in : " + measureSumPerformance(SumTest::iterativeSum, 10_000_000) + " msecs");  
        System.out.println("Parallel sum done in : " + measureSumPerformance(SumTest::parallelSum, 10_000_000) + " msecs");  
    }  
  
    public static long measureSumPerformance(Function<Long, Long> adder, long n) {  
        long fastest = Long.MAX_VALUE;  
        for (int i = 0; i < 10; i++) {  
            long start = System.nanoTime();  
            long sum = adder.apply(n);  
            long duration = (System.nanoTime() - start) / 1_000_000;  
  
            System.out.println("Result : " + sum);  
            if (duration < fastest) {  
                fastest = duration;  
            }  
        }  
  
        return fastest;  
    }  
}
```

스트림 성능 측정

- ## ● 순차 스트림

[illegible]

- For 루프를 사용

[illegible]

스트림 성능 측정

- ## ● 병렬 스트림

[illegible]

스트림 성능 측정

- 병렬 스트림이 순차 스트림보다 성능이 나쁘게 나왔다.
 - `iterate`가 박싱된 객체를 생성하므로 이를 다시 언박싱하는 과정이 필요했다.
 - `iterate`는 병렬로 실행될 수 있도록 독립적인 청크로 분할하기가 어렵다.
- 리듀싱 과정을 시작하는 시점에 전제 숫자 리스트가 준비 되지 않았음으로 스트림을 병렬로 처리할 수 있도록 청크로 분할할 수 없다.
- 스트림이 병렬로 처리되도록 지시 했지만 순차처리 방식과 크게 다른 점이 없으므로 스레드를 할당하는 오버헤드만 증가 했다.

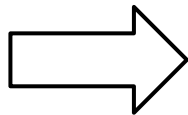
더 특화된 메서드 사용

- 멀티코어 프로세서를 활용해서 효과적으로 합계 연산을 병렬로 실행하려면 어떻게 해야할까?
- `LongStream.rangeClosed` 메서드는 `iterate`에 비해 다음과 같은 장점을 제공한다.
 - `LongStream.rangeClosed`는 기본형 `long`을 직접 사용하므로 박싱과 언박싱 오버헤드가 사라진다.
 - `LongStream.rangeClosed`는 쉽게 청크로 분할할 수 있는 숫자 범위를 생산한다.
 - 1-20 범위로 숫자를 각각 1-5, 6-10, 11-15, 16-20 범위의 숫자로 분할 할 수 있다.

더 특화된 메서드 사용

- LongStream.rangeClosed 적용

```
public static long sequentialSum(long n) {  
    return LongStream.iterate(1L, i -> i + 1)  
        .limit(n)  
        .reduce(0L, Long::sum);  
}  
  
public static long parallelSum(long n) {  
    return LongStream.iterate(1L, i -> i + 1)  
        .parallel()  
        .limit(n)  
        .reduce(0L, Long::sum);  
}
```



```
public static long rangedSum(long n) {  
    return LongStream.rangeClosed(1, n)  
        .reduce(0L, Long::sum);  
}  
  
public static long parallelRangedSum(long n) {  
    return LongStream.rangeClosed(1, n)  
        .parallel()  
        .reduce(0L, Long::sum);  
}
```


더 특화된 메서드 사용

- ## ● 순차 스트림

[illegible]

- ## ● 병력 스트림

[illegible]

더 특화된 메서드 사용

- 순차 실행보다 빠른 성능을 갖는 병렬 리듀싱을 만들었다.
- 이번에는 실질적으로 리듀싱 연상이 병렬로 실행된다.
- 올바른 자료구조를 선택해야 병렬 실행도 최적의 성능을 발휘할 수 있다.
- 병렬화는 완전 공짜가 아니다.
- 코어 간에 데이터 전송 시간보다 훨씬 오래 걸리는 작업만 병렬로 다른 코어에서 수행하는 것이 바람직하다.
- 병렬화해서 코드 실행 속도를 빠르게 하고싶으면 항상 병렬화를 올바르게 사용하고 있는지 확인해야 한다.

병렬 스트림의 올바른 사용법

- 숫자 n까지의 자연수를 더하면서 공유된 누적자를 바꾸는 코드

```
public class Accumulator {  
    public long total = 0;  
  
    public void add(long value) {  
        total += value;  
    }  
  
    public static long sideEffectParallelSum(long n) {  
        Accumulator accumulator = new Accumulator();  
        LongStream.rangeClosed(1, n).parallel().forEach(accumulator::add);  
        return accumulator.total;  
    }  
}
```

병렬 스트림의 올바른 사용법

- `total`을 접근 할 때마다 데이터 레이스 문제가 발생
- 여러 스레드에서 공유하는 객체의 상태를 바꾸는 `forEach` 블록 내부에서 `add` 메서드를 호출하면서 문제 발생

```
Result : 28129762871517
Result : 10281406556999
Result : 19595240673030
Result : 12217589508994
Result : 4101562812500
Result : 7925857044439
Result : 17859139775519
Result : 7146721643457
Result : 19949776397698
Result : 13054543779420
SideEffect parallel sum done in : 4 msecs
```

병렬 스트림 효과적으로 사용하기

- 확신이 서지 않는다면 직접 측정하라.
- 박싱을 주의하라
- 순차 스트림보다 병렬 스트림에서 성능이 떨어지는 연산이 있다.
- 스트림에서 수행하는 전체 파이프라인 연산 비용을 고려하라.
- 소량의 데이터에서는 병렬 스트림 도움 되지 않는다.
- 스트림을 구성하는 자료구조가 적절한지 확인하라.
- 스트림의 특성과 파이프라인의 중간 연산이 스트림의 특성을 어떻게 바꾸는지에 따라 분해 과정의 성능이 달라질 수 있다.
- 최종 연산의 병합 과정 비용을 살펴보라.