

*Luke's Language*로 파싱된 이펙티브 자바

스트림 선언

제작. 홍성혁

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 *API*가 아닌, **함수형 프로그래밍**에 기초한 패러다임”

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 API가 아닌, 함수형 프로그래밍에 기초한 패러다임”

함수형 프로그래밍

함수형 프로그래밍

한글 53개 언어

기사 말하다

읽다 편집하다 기록 보기 도구

위키피디아, 무료 백과사전에서

서브루틴 지향 프로그래밍에 대해서는 절차적 프로그래밍을 참조하세요.

컴퓨터 과학에서 함수형 프로그래밍은 함수를 적용하고 구성하여 프로그램을 구성하는 프로그래밍 패러다임입니다. 함수 정의가 프로그램의 실행 상태를 업데이트 하는 명령형 명령문 시퀀스가 아니라 값을 다른 값에 매핑하는 표현식 트리인 선언적 프로그래밍 패러다임입니다.

함수형 프로그래밍에서 함수는 일급 시민으로 취급됩니다. 즉, 함수는 이름(로컬 식별자 포함)에 바인딩되고, 인수로 전달되고, 다른 데이터 유형과 마찬가지로 다른 함수에서 반환될 수 있습니다. 이를 통해 프로그램을 선언적이고 구성 가능한 스타일로 작성할 수 있으며, 작은 함수가 모듈 방식으로 결합됩니다.

함수형 프로그래밍은 때때로 순수 함수형 프로그래밍과 동의어로 취급되는데, 순수 함수형 프로그래밍은 모든 함수를 결정적 수학 함수 또는 순수 함수로 취급하는 함수형 프로그래밍의 하위 집합입니다. 순수 함수가 주어진 인수와 함께 호출되면 항상 동일한 결과를 반환하며 변경 가능한 상태나 기타 부작용의 영향을 받을 수 없습니다. 이는 명령형 프로그래밍에서 일반적인 불순수 프로시저와 대조되는데, 불순수 프로시저는 부작용(예: 프로그램 상태 수정 또는 사용자 입력 수신)을 가질 수 있습니다. 순수 함수형 프로그래밍의 지지자들은 부작용을 제한함으로써 프로그램의 버그가 줄어들고 디버깅과 테스트가 더 쉬워지며 형식적 검증에 더 적합할 수 있다고 주장합니다. [1][2]

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 API가 아닌, 함수형 프로그래밍에 기초한 패러다임”

함수형 프로그래밍

함수형 프로그래밍

한글 53개 언어

기사 말하다

읽다 편집하다 기록 보기 도구

위키피디아, 무료 백과사전에서

서브루틴 지향 프로그래밍에 대해서는 절차적 프로그래밍을 참조하세요.

컴퓨터 과학에서 함수형 프로그래밍은 함수를 적용하고 구성하여 프로그램을 구성하는 프로그래밍 패러다임입니다. 함수 정의가 프로그램의 실행 상태를 업데이트 하는 명령형 명령문 시퀀스가 아니라 값을 다른 값에 매핑하는 표현식 트리 인 선언적 프로그래밍 패러다임입니다.

함수형 프로그래밍에서 함수는 일급 시민으로 취급됩니다. 즉, 함수는 이름(로컬 식별자 포함)에 바인딩되고, 인수로 전달되고, 다른 데이터 유형과 마찬가지로 다른 함수에서 반환될 수 있습니다. 이를 통해 프로그램을 선언적이고 구성 가능한 스타일로 작성할 수 있으며, 작은 함수가 모듈 방식으로 결합됩니다.

함수형 프로그래밍은 때때로 순수 함수형 프로그래밍과 동의어로 취급되는데, 순수 함수형 프로그래밍은 모든 함수를 결정적 수학 함수 또는 순수 함수로 취급하는 함수형 프로그래밍의 하위 집합입니다. 순수 함수가 주어진 인수와 함께 호출되면 항상 동일한 결과를 반환하며 변경 가능한 상태나 기타 부작용의 영향을 받을 수 없습니다. 이는 명령형 프로그래밍에서 일반적인 불순수 프로시저와 대조되는데, 불순수 프로시저는 부작용(예: 프로그램 상태 수정 또는 사용자 입력 수신)을 가질 수 있습니다. 순수 함수형 프로그래밍의 지지자들은 부작용을 제한함으로써 프로그램의 버그가 줄어들고 디버깅과 테스트가 더 쉬워지며 형식적 검증에 더 적합할 수 있다고 주장합니다. [1][2]

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 API가 아닌, 함수형 프로그래밍에 기초한 패러다임”

함수형 프로그래밍

함수형 프로그래밍

한글 53개 언어

기사 말하다

읽다 편집하다 기록 보기 도구

위키피디아, 무료 백과사전에서

서브루틴 지향 프로그래밍에 대해서는 절차적 프로그래밍을 참조하세요.

컴퓨터 과학에서 함수형 프로그래밍은 함수를 적용하고 구성하여 프로그램을 구성하는 프로그래밍 패러다임입니다. 함수 정의가 프로그램의 실행 상태를 업데이트 하는 명령형 명령문 시퀀스가 아니라 값을 다른 값에 매핑하는 표현식 트리 인 선언적 프로그래밍 패러다임입니다.

함수형 프로그래밍에서 함수는 일급 시민으로 취급됩니다. 즉, 함수는 이름(로컬 식별자 포함)에 바인딩되고, 인수로 전달되고, 다른 데이터 유형과 마찬가지로 다른 함수에서 반환될 수 있습니다. 이를 통해 프로그램을 선언적이고 구성 가능한 스타일로 작성할 수 있으며, 작은 함수가 모듈 방식으로 결합됩니다.

함수형 프로그래밍은 때때로 순수 함수형 프로그래밍과 동의어로 취급되는데, 순수 함수형 프로그래밍은 모든 함수를 결정적 수학 함수 또는 순수 함수로 취급하는 함수형 프로그래밍의 하위 집합입니다. 순수 함수가 주어진 인수와 함께 호출되면 항상 동일한 결과를 반환하며 변경 가능한 상태나 기타 부작용의 영향을 받을 수 없습니다. 이는 명령형 프로그래밍에서 일반적인 불순수 프로시저와 대조되는데, 불순수 프로시저는 부작용(예: 프로그램 상태 수정 또는 사용자 입력 수신)을 가질 수 있습니다. 순수 함수형 프로그래밍의 지지자들은 부작용을 제한함으로써 프로그램의 버그가 줄어들고 디버깅과 테스트가 더 쉬워지며 형식적 검증에 더 적합할 수 있다고 주장합니다. [1][2]

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 *API*가 아닌, **함수형 프로그래밍**에 기초한 패러다임”

스트림 패러다임의 핵심은 계산을 일련의 변환(transformation)으로 재구성하는 것
각 변환 단계는, 이전 단계의 결과를 받아 처리하는 순수 함수

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 *API*가 아닌, **함수형 프로그래밍**에 기초한 패러다임”

스트림 패러다임의 핵심은 계산을 일련의 변환(transformation)으로 재구성하는 것
각 변환 단계는, 이전 단계의 결과를 받아 처리하는 순수 함수

=> **side effect** (함수 입력/결과 외의 다른 상태의 변경)이 없어야 한다!

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

“스트림은 그저 또 하나의 *API*가 아닌, **함수형 프로그래밍**에 기초한 패러다임”

스트림 패러다임의 핵심은 계산을 일련의 변환(transformation)으로 재구성하는 것
각 변환 단계는, 이전 단계의 결과를 받아 처리하는 순수 함수

=> side effect (함수 입력/결과 외의 다른 상태의 변경)이 없어야 한다!

코드 46-1 스트림 패러다임을 이해하지 못한 채 API만 사용했다 - 따라 하지 말 것!

```
Map<String, Long> freq = new HashMap<>();  
try (Stream<String> words = new Scanner(file).tokens()) {  
    words.forEach(word -> {  
        freq.merge(word.toLowerCase(), 1L, Long::sum);  
    });  
} => side effect
```


스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

올바른 스트림 활용 예

코드 46-2 스트림을 제대로 활용해 빈도표를 초기화한다.

```
Map<String, Long> freq;  
try (Stream<String> words = new Scanner(file).tokens()) {  
    freq = words  
        .collect(groupingBy(String::toLowerCase, counting()));  
}
```

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가

올바른 스트림 활용 예

코드 46-2 스트림을 제대로 활용해 빈도표를 초기화한다.

```
Map<String, Long> freq;  
try (Stream<String> words = new Scanner(file).tokens()) {  
    freq = words  
        .collect(groupingBy(String::toLowerCase, counting()));  
}
```

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가 - Collector

Collector? 수집기?

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가 - Collector

Collector? 수집기?

=> “축소(*reduction*) 전략을 캡슐화한 블랙박스 객체”

=> *toList()*, *toSet()*, *toCollection(collectionFactory)*

코드 46-3 빈도표에서 가장 흔한 단어 10개를 뽑아내는 파이프라인

```
List<String> topTen = freq.keySet().stream()
    .sorted(comparing(freq::get).reversed())
    .limit(10)
    .collect(toList());
```



마지막 `toList`는 `Collectors`의 메서드다. 이처럼 `Collectors`의 멤버를 정적 임포트하여 쓰면 스트림 파이프라인 가독성이 좋아져, 흔히들 이렇게 사용한다.

스트림에서는 부작용 없는 함수를 사용하라

Item 46 - 스트림이란 무엇인가 - Collector

Collector? 수집기?

=> “축소(*reduction*) 전략을 캡슐화한 블랙박스 객체”

=> *toList()*, *toSet()*, *toCollection(collectionFactory)*

=> `java.util.stream.Collectors` API 문서 읽어보세요 ~

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되지 않는다

Stream VS Collection - why?

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Stream

```
package java.util.stream;

import ...

/** A sequence of elements supporting sequential and parallel aggregate ...*/
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    /** Returns a stream consisting of the elements of this stream that match ...*/
    @Contract(pure = true)
    Stream<T> filter(Predicate<? super T> predicate);

    /** Returns a stream consisting of the results of applying the given ...*/
    @Contract(pure = true)
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);

    /** Returns an {@code IntStream} consisting of the results of applying the ...*/
    @Contract(pure = true)
    IntStream mapToInt(ToIntFunction<? super T> mapper);

    /** Returns a {@code LongStream} consisting of the results of applying the ...*/
    @Contract(pure = true)
    LongStream mapToLong(ToLongFunction<? super T> mapper);

    /** Returns a {@code DoubleStream} consisting of the results of applying the ...*/
    @Contract(pure = true)
    DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);

    /** Returns a stream consisting of the results of replacing each element of ...*/
    @Contract(pure = true)
    <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Collection

```
package java.util;

import ...

/** The root interface in the <i>collection hierarchy</i>. A collection ...*/

public interface Collection<E> extends Iterable<E> {
    // Query operations

    /** Returns the number of elements in this collection. If this collection ...*/
    @Contract(pure = true)
    int size();

    /** Returns {@code true} if this collection contains no elements. ...*/
    boolean isEmpty();

    /** Returns {@code true} if this collection contains the specified element. ...*/
    @Contract(pure = true)
    boolean contains(Object o);

    /** Returns an iterator over the elements in this collection. There are no ...*/
    @NotNull
    Iterator<E> iterator();

    /** Returns an array containing all of the elements in this collection. ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(pure = true)
    Object[] toArray();

    /** Returns an array containing all of the elements in this collection; ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(mutates = "param1")
    <T> T[] toArray( @NotNull T[] a);
}
```


반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Collection
Iterable?

```
package java.util;

import ...

/** The root interface in the <i>collection hierarchy</i>. A collection ...*/

public interface Collection<E> extends Iterable<E> {
    // Query Operations

    /** Returns the number of elements in this collection. If this collection ...*/
    @Contract(pure = true)
    int size();

    /** Returns {@code true} if this collection contains no elements. ...*/
    boolean isEmpty();

    /** Returns {@code true} if this collection contains the specified element. ...*/
    @Contract(pure = true)
    boolean contains(Object o);

    /** Returns an iterator over the elements in this collection. There are no ...*/
    @NotNull
    Iterator<E> iterator();

    /** Returns an array containing all of the elements in this collection. ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(pure = true)
    Object[] toArray();

    /** Returns an array containing all of the elements in this collection; ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(mutates = "param1")
    <T> T[] toArray( @NotNull T[] a);
}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Collection

Iterable? 반복 가능한?

```
package java.lang;
```

```
import ...
```

```
/** Implementing this interface allows an object to be the target of the enhanced ...*/
```

```
public interface Iterable<T> {
```

```
    Returns an iterator over elements of type T.
```

```
    반환: an Iterator.
```

```
    @NotNull
```

```
    Iterator<T> iterator();
```

```
/** Performs the given action for each element of the {@code Iterable} ...*/
```

```
default void forEach(Consumer<? super T> action) {
```

```
    Objects.requireNonNull(action);
```

```
    for (T t : this) {
```

```
        action.accept(t);
```

```
    }
```

```
}
```

```
/** Creates a {@link Spliterator} over the elements described by this ...*/
```

```
default Spliterator<T> spliterator() {
```

```
    return Spliterators.spliteratorUnknownSize(iterator(), characteristics: 0);
```

```
}
```

```
}!
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Collection

Iterable? 반복 가능한?

Iterator?

```
package java.lang;
```

```
import ...
```

```
/** Implementing this interface allows an object to be the target of the enhanced ...*/
```

```
public interface Iterable<T> {
```

```
    Returns an iterator over elements of type T.
```

```
    반환: an Iterator.
```

```
    @Nonnull
```

```
    Iterator<T> iterator();
```

```
/** Performs the given action for each element of the {@code Iterable} ...*/
```

```
default void forEach(Consumer<? super T> action) {
```

```
    Objects.requireNonNull(action);
```

```
    for (T t : this) {
```

```
        action.accept(t);
```

```
    }
```

```
}
```

```
/** Creates a {@link Spliterator} over the elements described by this ...*/
```

```
default Spliterator<T> spliterator() {
```

```
    return Spliterators.spliteratorUnknownSize(iterator(), characteristics: 0);
```

```
}
```

```
}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되지 않는다

Stream VS Collection - why?

Collection

Iterable? 반복 가능한?

Iterator? 반복자?

```
package java.util;
```

```
import java.util.function.Consumer;
```

An iterator over a collection. **Iterator** takes the place of **Enumeration** in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

API 참고 사항: An **Enumeration** can be converted into an **Iterator** by using the **Enumeration.asIterator** method.

시작 시간: 1.2

관련 주제: **Collection**,
ListIterator,
Iterable

작성자: Josh Bloch

타입 매개변수: **<E>** - the type of elements returned by this iterator

```
public interface Iterator<E> {  
    /** Returns {@code true} if the iteration has more elements. ...*/  
    @Contract(pure = true)  
    boolean hasNext();  
  
    /** Returns the next element in the iteration. ...*/  
    @Contract(mutates = "this")  
    E next();  
  
    /** Removes from the underlying collection the last element returned ...*/  
    default void remove() { throw new UnsupportedOperationException("remove"); }  
  
    /** Performs the given action for each remaining element until all elements ...*/  
    default void forEachRemaining(Consumer<? super E> action) {...}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되지 않는다

Stream VS Collection - why?

Collection

Iterable? 반복 가능한?

Iterator? 반복자?

컬렉션에 대한 반복자입니다. Iterator는 Java 컬렉션 프레임워크에서 Enumeration을 대신합니다.

반복자는 다음 두 가지 면에서 열거형과 다릅니다:

- 반복자를 사용하면 호출자가 잘 정의된 의미 체계를 사용하여 반복하는 동안 기본 컬렉션에서 요소를 제거할 수 있습니다.
- 메소드 이름이 개선되었습니다.

이 인터페이스는 Java 컬렉션 프레임워크의 멤버입니다.

```
package java.util;  
  
import java.util.function.Consumer;
```

An iterator over a collection. **Iterator** takes the place of **Enumeration** in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

API 참고 사항: An **Enumeration** can be converted into an **Iterator** by using the **Enumeration.asIterator** method.

시작 시간: 1.2

관련 주제: **Collection**,
ListIterator,
Iterable

작성자: Josh Bloch

```
/** Removes from the underlying collection the last element returned ...*/  
default void remove() { throw new UnsupportedOperationException("remove"); }
```

```
/** Performs the given action for each remaining element until all elements ...*/  
default void forEachRemaining(Consumer<? super E> action) { ... }
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되지 않는다

Iterator

📄 21 languages ▾

Article [Talk](#)

[Read](#) [Edit](#) [View history](#) [Tools](#) ▾

[컴퓨터 프로그래밍](#)에서 **반복자**는 [컬렉션](#)의 각 항목에 대한 액세스를 순서대로 제공하는 [객체](#)입니다. ^{[1][2][3]}

컬렉션은 [인터페이스](#)를 통해 여러 개의 반복자를 제공할 수 있으며, 반복자는 항목을 서로 다른 순서(예: 앞뒤)로 제공합니다.

반복자는 종종 컬렉션 구현의 기본 구조에 따라 구현되며 반복자의 작동적 의미론을 구현하기 위해 컬렉션과 긴밀하게 [결합되는 경우가 많습니다](#).

반복자는 [데이터베이스 커서](#)와 동작이 비슷합니다.

반복자는 1974년 [CLU 프로그래밍 언어](#)에서 처음 등장했습니다.

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되지 않는다

Iterator

📄 21 languages

Article Talk

Read Edit View history Tools

컴퓨터 프로그래밍에서 반복자는 **컬렉션**의 각 항목에 대한 액세스를 순서대로 제공하는 객체입니다. ^{[1][2][3]}

컬렉션은 인터페이스를 통해 여러 개의 반복자를 제공할 수 있으며, 반복자는 항목을 서로 다른 순서(예: 앞뒤)로 제공합니다.

반복자는 종종 컬렉션 구현의 기본 구조에 따라 구현되며 반복자의 작동적 의미를 구현하기 위해 컬렉션과 긴밀하게 결합되는 경우가 많습니다.

반복자는 데이터베이스 커서와 동작이 비슷합니다.

반복자는 1974년 CLU 프로그래밍 언어에서 처음 등장했습니다.

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Stream

=> Iterable을 extend 하지 않아, for-each 불가

```
package java.util.stream;

import ...

/** A sequence of elements supporting sequential and parallel aggregate ...*/
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    /** Returns a stream consisting of the elements of this stream that match ...*/
    @Contract(pure = true)
    Stream<T> filter(Predicate<? super T> predicate);

    /** Returns a stream consisting of the results of applying the given ...*/
    Stream<R> map(Function<? super T, ? extends R> mapper);

    /** Returns an {@code IntStream} consisting of the results of applying the ...*/
    @Contract(pure = true)
    IntStream mapToInt(ToIntFunction<? super T> mapper);

    /** Returns a {@code LongStream} consisting of the results of applying the ...*/
    @Contract(pure = true)
    LongStream mapToLong(ToLongFunction<? super T> mapper);

    /** Returns a {@code DoubleStream} consisting of the results of applying the ...*/
    @Contract(pure = true)
    DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);

    /** Returns a stream consisting of the results of replacing each element of ...*/
    @Contract(pure = true)
    <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```


반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Stream

=> Iterable을 extend 하지 않아, for-each 불가

Collection

=> Iterable을 extend, stream으로 쉽게 변환 가능

```
package java.util;

import ...

/** The root interface in the <i>collection hierarchy</i>. A collection ...*/

public interface Collection<E> extends Iterable<E> {
    // Query operations

    /** Returns the number of elements in this collection. If this collection ...*/
    @Contract(pure = true)
    int size();

    /** Returns true if this collection contains no elements. ...*/
    boolean isEmpty();

    /** Returns true if this collection contains the specified element. ...*/
    boolean contains(Object o);

    /** Returns an iterator over the elements in this collection. There are no ...*/
    @NotNull
    Iterator<E> iterator();

    /** Returns an array containing all of the elements in this collection. ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(pure = true)
    Object[] toArray();

    /** Returns an array containing all of the elements in this collection; ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(mutates = "param1")
    <T> T[] toArray(@NotNull T[] a);
}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Stream

=> Iterable을 extend 하지 않아, for-each 불가

Collection

=> Iterable을 extend, stream으로 쉽게 변환 가능

=> 때문에 원소 시퀀스를 반환할때, Collection 또는 그 하위 타입을 반환하는게 일반적

```
package java.util;

import ...

/** The root interface in the <i>collection hierarchy</i>. A collection ...*/

public interface Collection<E> extends Iterable<E> {
    // query operations

    /** Returns the number of elements in this collection. If this collection ...*/
    @Contract(pure = true)
    int size();

    boolean isEmpty();

    /** Returns true if this collection contains no elements. ...*/

    /** Returns true if this collection contains the specified element. ...*/

    @NotNull
    Iterator<E> iterator();

    /** Returns an array containing all of the elements in this collection. ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(pure = true)
    Object[] toArray();

    /** Returns an array containing all of the elements in this collection; ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(mutates = "param1")
    <T> T[] toArray( @NotNull T[] a);
}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

Stream VS Collection - why?

Stream

=> Iterable을 extend 하지 않아, for-each 불가

Collection

=> Iterable을 extend, stream으로 쉽게 변환 가능

=> 때문에 원소 시퀀스를 반환할때, Collection 또는 그 하위 타입을 반환하는게 일반적

=> 단, 컬렉션을 반환한다는 이유로 덩치 큰 시퀀스를 메모리에 올려서는 안된다!

```
package java.util;

import ...

/** The root interface in the <i>collection hierarchy</i>. A collection ...*/

public interface Collection<E> extends Iterable<E> {
    // query operations

    /** Returns the number of elements in this collection. If this collection ...*/
    @Contract(pure = true)
    int size();

    boolean isEmpty();

    /** Returns an array containing all of the elements in this collection. ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(pure = true)
    Object[] toArray();

    /** Returns an array containing all of the elements in this collection; ...*/
    @NotNull @Flow(sourcesContainer = true, targetsContainer = true) @Contract(mutates = "param1")
    <T> T[] toArray( @NotNull T[] a);
}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되지 않는다

그래서 어떻게 하지?

=> `AbstractList` 구현으로 간단하게 끝낼 수 있음

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

그래서 어떻게 하지?

=> AbstractList 구현으로 간단하게

코드 47-5 입력 집합의 역집합을 전용 컬렉션에 담아 반환한다.

```
public class PowerSet {
    public static final <E> Collection<Set<E>> of(Set<E> s) {
        List<E> src = new ArrayList<>(s);
        if (src.size() > 30)
            throw new IllegalArgumentException(
                "집합에 원소가 너무 많습니다(최대 30개).: " + s);

        return new AbstractList<Set<E>>() {
            @Override public int size() {
                // 역집합의 크기는 2를 원래 집합의 원소 수만큼 거듭제곱 것과 같다.
                return 1 << src.size();
            }

            @Override public boolean contains(Object o) {
                return o instanceof Set && src.containsAll((Set)o);
            }

            @Override public Set<E> get(int index) {
                Set<E> result = new HashSet<>();
                for (int i = 0; index != 0; i++, index >>= 1)
                    if ((index & 1) == 1)
                        result.add(src.get(i));
                return result;
            }
        };
    }
}
```

반환 타입으로는 스트림보다 컬렉션이 낫다

Item 47 - 흐름은 반복되

그래서 어떻게 하지?

=> AbstractList 구현으로 간단하게

코드 47-5 입력 집합의 역집합을 전용 컬렉션에 담아 반환한다.

```
public class PowerSet {
    public static final <E> Collection<Set<E>> of(Set<E> s) {
        List<E> src = new ArrayList<>(s);
        if (src.size() > 30)
            throw new IllegalArgumentException(
                "집합에 원소가 너무 많습니다(최대 30개).: " + s);

        return new AbstractList<Set<E>>() {
            @Override public int size() {
                // 역집합의 크기는 2를 원래 집합의 원소 수만큼 거듭제곱 것과 같다.
                return 1 << src.size();
            }

            @Override public boolean contains(Object o) {
                return o instanceof Set && src.containsAll((Set)o);
            }

            @Override public Set<E> get(int index) {
                Set<E> result = new HashSet<>();
                for (int i = 0; index != 0; i++, index >>= 1)
                    if ((index & 1) == 1)
                        result.add(src.get(i));
                return result;
            }
        };
    }
}
```

=> size(), contains() 정도만 구현해주면 된다.

스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

Stream reduce의 인자가 되는, 누적(accumulator)과 결합(combiner) 는 반드시 다음 조건을 지켜야 한다.

1. 결합 법칙을 만족한다. (associative)
2. 간섭받지 않는다. (non-interfering)
3. 상태를 갖지 않는다. (stateless)

=> 지키지 않으면, **병렬화시 난리남...**

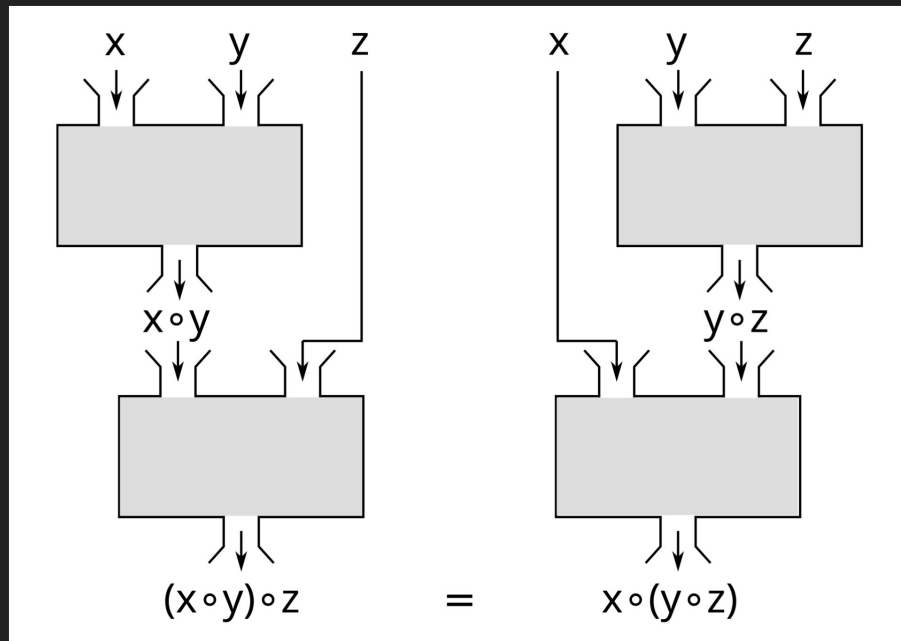
스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

1. 결합 법칙 (associative)

수학에서 **결합법칙** (結合 法則, associative property)은 이항연산이 가질 수 있는 성질이다. 한 식에서 연산이 두 번 이상 연속될 때, 앞쪽의 연산을 먼저 계산한 값과 뒤쪽의 연산을 먼저 계산한 결과가 항상 같을 경우 그 연산은 **결합법칙**을 만족한다고 한다.



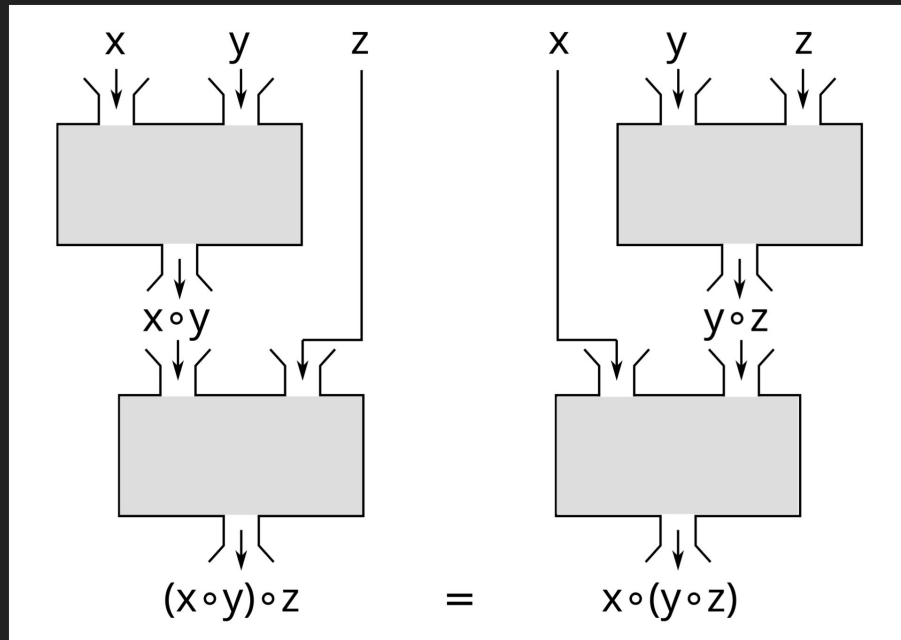
스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

1. 결합 법칙 (associative)

수학에서 **결합법칙** (結合 法則, associative property)은 이항연산이 가질 수 있는 성질이다. 한 식에서 **연산이** 두 번 이상 **연속될 때**, 앞쪽의 연산을 먼저 계산한 값과 뒤쪽의 연산을 먼저 계산한 결과가 항상 같을 경우 그 연산은 **결합법칙을** 만족한다고 한다.



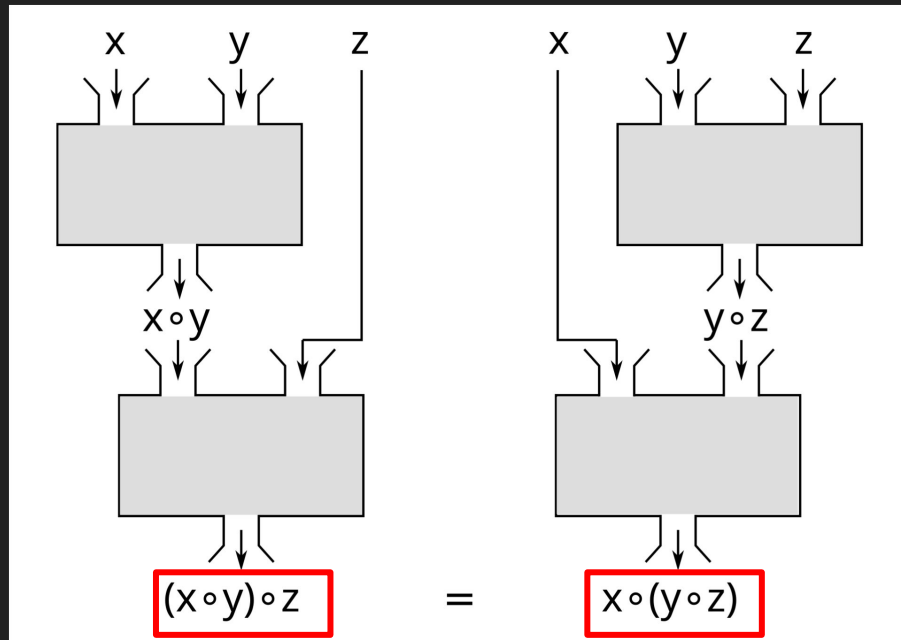
스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

1. 결합 법칙 (associative)

수학에서 **결합법칙** (結合 法則, associative property)은 이항연산이 가질 수 있는 성질이다. 한 식에서 연산이 두 번 이상 연속될 때, **앞쪽의 연산을 먼저 계산한 값과 뒤쪽의 연산을 먼저 계산한 결과**가 항상 같을 경우 그 연산은 **결합법칙을 만족한다**고 한다.



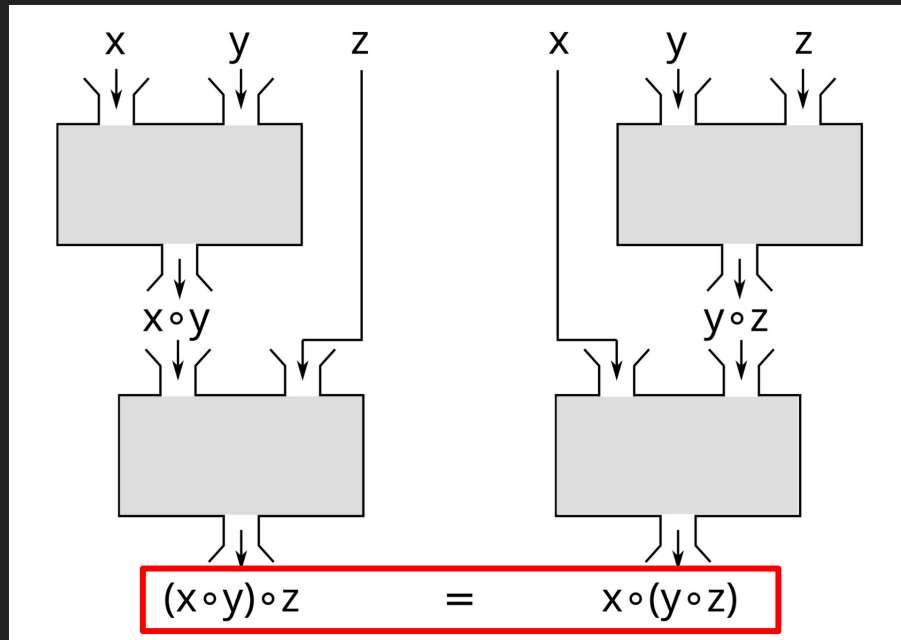
스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

1. 결합 법칙 (associative)

수학에서 **결합법칙** (結合 法則, associative property)은 이항연산이 가질 수 있는 성질이다. 한 식에서 연산이 두 번 이상 연속될 때, 앞쪽의 연산을 먼저 계산한 값과 뒤쪽의 연산을 먼저 계산한 결과가 항상 **같은 경우** 그 연산은 **결합법칙**을 만족한다고 한다.



스트림 병렬화는 주의해서 적용하라

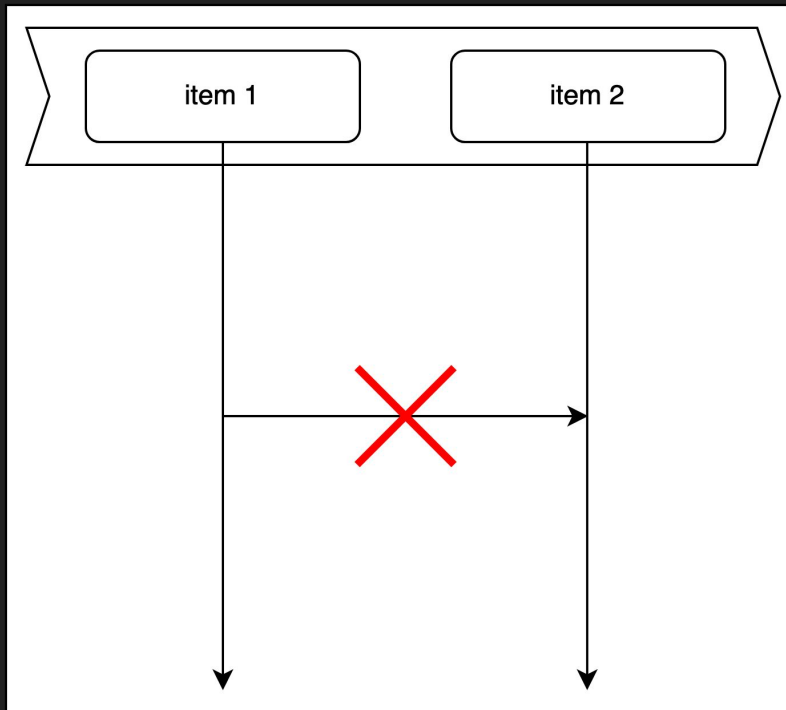
Item 48 - 패러럴 스트림

Stream에 대한 명세

2. 불간섭 (non-interfering)

=> 사이드 이펙트 이슈

=> DB 트랜잭션 Isolation을 생각해보자



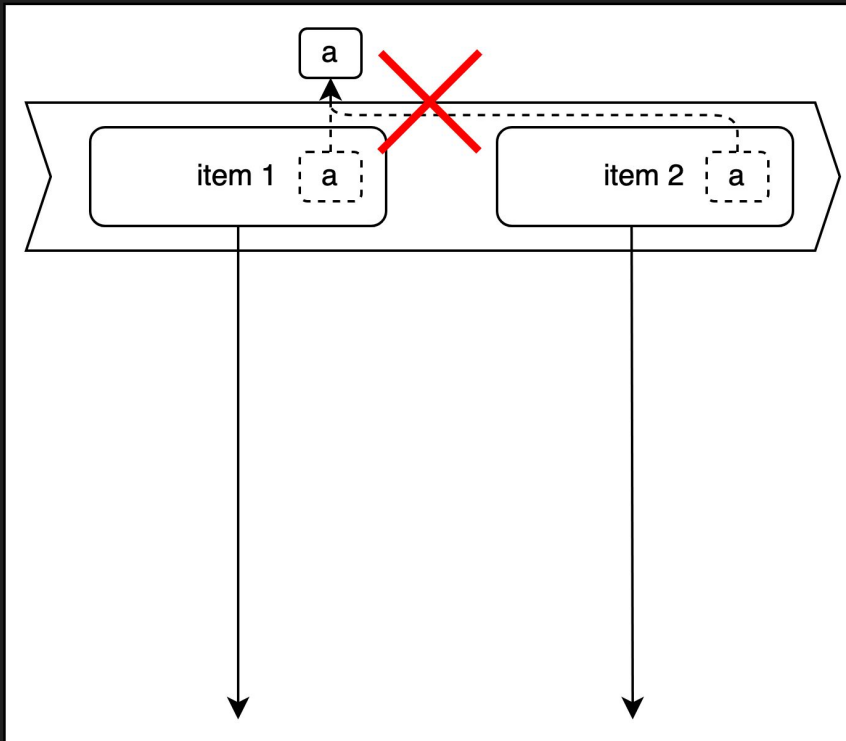
스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

3. 상태를 갖지 않는다. (stateless)

=> 사이드 이펙트 이슈



스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

Stream reduce의 인자가 되는, 누적(accumulator)과 결합(combiner) 는 반드시 다음 조건을 지켜야 한다.

1. 결합 법칙을 만족한다. (associative)
2. 간섭받지 않는다. (non-interfering)
3. 상태를 갖지 않는다. (stateless)

=> 함수형 프로그래밍, 멀티스레드 환경에서 가장 중요한 것

스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

Stream에 대한 명세

Stream reduce의 인자가 되는, 누적(accumulator)과 결합(combiner) 는 반드시 다음 조건을 지켜야 한다.

1. 결합 법칙을 만족한다. (associative)
2. 간섭받지 않는다. (non-interfering)
3. 상태를 갖지 않는다. (stateless)

=> 함수형 프로그래밍, 멀티스레드 환경에서 가장 중요한 것

=> 각 과정은 **원자성(atomicity)**이 보장되어야 한다.

스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

자바의 동시성 역사

Thread, synchronized, wait/notify

java.util.concurrent - Executor

java.util.concurrent - fork / join

java.util.stream - parallel

스트림 병렬화는 주의해서 적용하라

Item 48 - 패러럴 스트림

자바의 동시성 역사

Thread, synchronized, wait/notify

java.util.concurrent - Executor

java.util.concurrent - fork / join

=> 이거 좀 재밌음. 한번 찾아보셈

java.util.stream - parallel

E.O.D.