

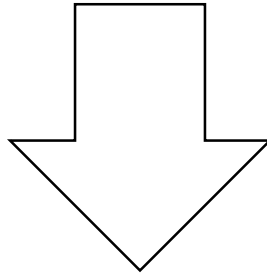
4장. 스트림 소개

거의 모든 자바 애플리케이션은 컬렉션(Collections)을 만들고 처리하는 과정을 포함한다.

ex) 요리 컬렉션이 있는데 컬렉션의 요리를 반복하면서 각 요리의 칼로리량을 더한다. 어떤 사람은 컬렉션에서 칼로리가 적은 요리만 골라 특별 건강 메뉴를 구성하고 싶을지도 모른다.

대부분의 자바 애플리케이션에서는 컬렉션을 많이 사용하지만 완벽한 컬렉션 관련 연산을 지원하려면 한참 멀었다.

- 대부분의 비즈니스 로직에는 요리를 카테고리(예를 들면 채식주의자용)로 그룹화한다든가 가장 비싼 요리를 찾는 등의 연산이 포함된다.
- 데이터베이스에서는 선언형으로 이와 같은 연산을 표현할 수 있다.
ex) `SELECT name FROM dishes WHERE calorie < 400` 컬렉션에서도 가능하지 않을까?
- 많은 요소를 포함하는 커다란 컬렉션은 어떻게 처리할까?
병렬로 컬렉션의 요소를 처리해야 한다. 하지만 단순 반복 처리 코드에 비해 복잡하고 어렵다.



이 질문의 답은 스트림(Stream)이다.

- 스트림을 이용하면 선언형(즉, 데이터를 처리하는 임시 구현 코드 대신 질의로 표현할 수 있다)으로 컬렉션 데이터를 처리할 수 있다.
- 스트림을 이용하면 멀티스레드 코드를 구현하지 않아도 데이터를 투명하게 병렬로 처리할 수 있다.

자바 7 코드

```
1 List<Dish> lowCaloricDishes = new ArrayList<>();
2 for(Dish dish: menu) {
3     if(dish.getCalories() < 400) {
4         lowCaloricDishes.add(dish);
5     }
6 }
7 Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
8     public int compare(Dish dish1, Dish dish2) {
9         return Integer.compare(dish1.getCalories(), dish2.getCalories());
10    }
11 });
12 List<String> lowCaloricDishesName = new ArrayList<>();
13 for(Dish dish: lowCaloricDishes) {
14     lowCaloricDishesName.add(dish.getName());
15 }
```

누적자로 요소 필터링

익명 클래스로 요리 정렬

정렬된 리스트를 처리하면서 요리 이름 선택

스트림이란 무엇인가?

자바 8 코드

```
1 List<String> lowCaloricDishsName =  
2     menu.stream()  
3         .filter(d → d.getCalories() < 400)  
4         .sorted(comparing(Dish::getCalories))  
5         .map(Dish::getName)  
6         .collect(toList());
```

400칼로리 이하의 요리 선택

칼로리로 요리 정렬

요리명 추출

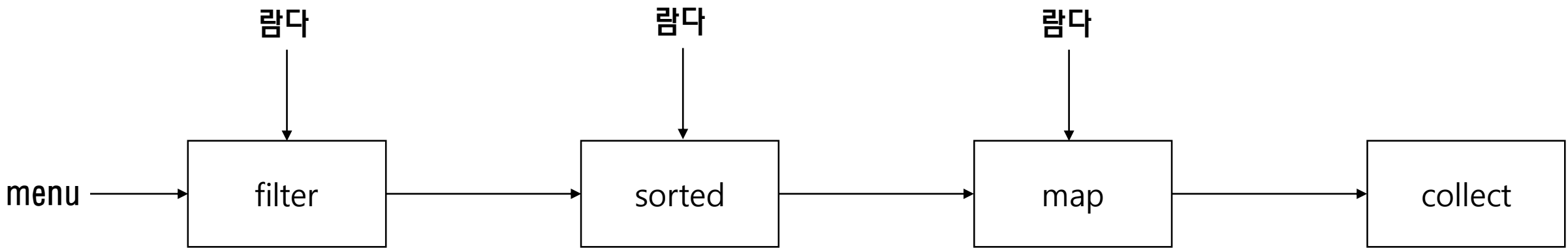
모든 요리명을 리스트에 저장

```
1 List<String> lowCaloricDishsName =  
2     menu.parallelStream()  
3         .filter(d → d.getCalories() < 400)  
4         .sorted(comparing(Dish::getCalories))  
5         .map(Dish::getName)  
6         .collect(toList());
```

stream()을 parallelStream()으로 바꾸면 이 코드를 멀티코어 아키텍처에서 병렬로 실행할 수 있다.

스트림의 새로운 기능이 소프트웨어 공학적으로 다음의 다양한 이득을 준다는 사실만 기억하자.

- 선언형으로 코드를 구현할 수 있다. 즉, 루프와 if 조건문 등의 제어 블록을 사용해서 어떻게 동작을 구현할지 지정할 필요 없이 ‘저칼로리의 요리만 선택하라’ 같은 동작의 수행을 지정할 수 있다.
- [그림 4-1]에서처럼 filter, sorted, map, collect 같은 여러 빌딩 블록 연산을 연결해서 복잡한 데이터 처리 파이프라인을 만들 수 있다. 여러 연산을 파이프라인으로 연결해도 여전히 가독성과 명확성이 유지된다.



[그림 4-1] 스트림 연산을 연결해서 스트림 파이프라인 형성

자바 8의 스트림 API의 특징을 다음처럼 요약할 수 있다.

- 선언형 : 더 간결하고 가독성이 좋아진다.
- 조립할 수 있음 : 유연성이 좋아진다.
- 병렬화 : 성능이 좋아진다.

스트림이란 정확히 뭘까?

스트림이란 ‘데이터 처리 연산을 지원하도록 소스에서 추출된 연속된 요소’로 정의할 수 있다.

데이터 처리 연산

- 스트림은 함수형 프로그래밍 언어에서 일반적으로 지원하는 연산과 데이터베이스와 비슷한 연산을 지원한다.
ex) filter, map, reduce, find, match, sort
- 스트림 연산은 순차적으로 또는 병렬로 실행할 수 있다.

소스

- 스트림은 컬렉션, 배열, I/O 자원 등의 데이터 제공 소스로부터 데이터를 소비한다.
- 정렬된 컬렉션으로 스트림을 생성하면 정렬이 그대로 유지된다.

연속된 요소

컬렉션과 마찬가지로 스트림은 특정 요소 형식으로 이루어진 연속된 값 집합의 인터페이스를 제공한다.

컬렉션	스트림
시간과 공간의 복잡성과 관련된 요소 저장 장치 및 접근 연산	filter, sorted, map 등 표현 계산식
데이터	계산

파이프라이닝(Pipelining)

- 대부분의 스트림 연산은 스트림 연산끼리 연결해서 커다란 파이프라인을 구성할 수 있도록 스트림 자신을 반환한다.
- 그 덕분에 게으름(laziness), 쇼트서킷(short-circuiting)같은 최적화를 얻을 수 있다. (5장에서 설명)
- 연산 파이프라인은 데이터 소스에 적용하는 데이터베이스 질의와 비슷하다.

내부 반복

- 반복자를 이용해서 명시적으로 반복하는 컬렉션과 달리 스트림은 내부 반복을 지원한다.

```
1 List<String> threeHighCaloricDishNames =  
2     menu.stream()  
3     .filter(dish -> dish.getCalories() > 300)  
4     .map(Dish::getName)  
5     .limit(3)  
6     .collect(toList());  
7 System.out.println(threeHighCaloricDishNames);
```

메뉴(요리 리스트)에서 스트림을 얻는다.

파이프라인 연산 만들기, 첫 번째로
고칼로리 요리를 필터링한다.

결과를 다른 리스트로 저장
결과는[pork, beef, chicken]이다.

스트림이란 ‘데이터 처리 연산’을 지원하도록 소스에서 추출된 연속된 요소’로 정의할 수 있다.

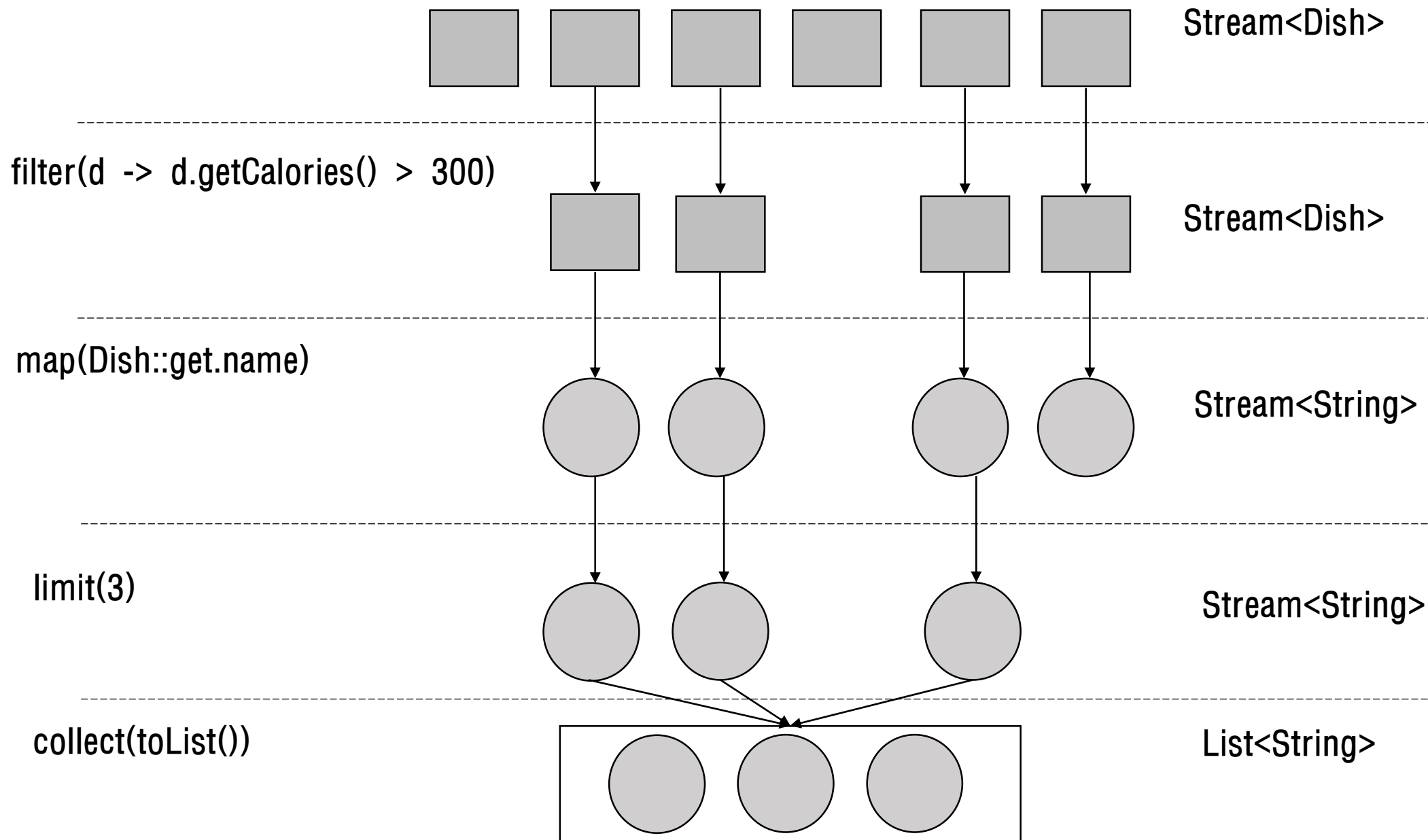
↓
filter, map, limit, collect

↓
요리 리스트(메뉴)

↓
요리 데이터

스트림 시작하기

메뉴 스트림



자바의 기존 컬렉션과 새로운 스트림 모두 연속된 요소 형식의 값을 저장하는 자료구조의 인터페이스를 제공한다.

컬렉션

- 현재 자료구조가 포함하는 모든 값을 메모리에 저장하는 자료구조
- 즉, 컬렉션의 모든 요소는 컬렉션에 추가하기 전에 계산되어야 한다.

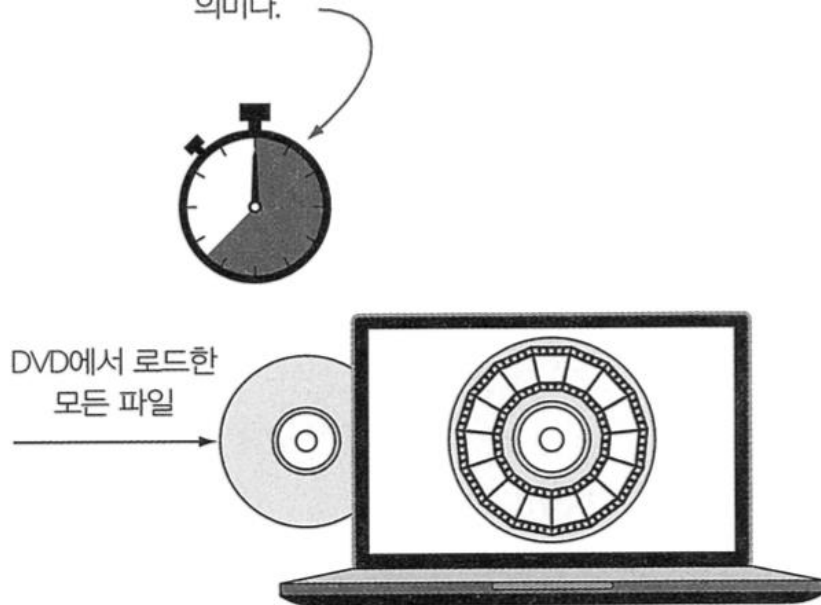
스트림

- 이론적으로 요청할 때만 요소를 계산하는 고정된 자료구조
- 스트림에 요소를 추가하거나 스트림에서 요소를 제거할 수 없다.
- 사용자가 요청하는 값만 스트림에서 추출한다는 것이 핵심
- 결과적으로 스트림은 생산자와 소비자 관계를 형성한다.

데이터를 언제 계산하느냐가 컬렉션과 스트림의 가장 큰 차이라고 할 수 있다.

자바 8의 컬렉션은 DVD에 저장된
영화에 비유할 수 있다.

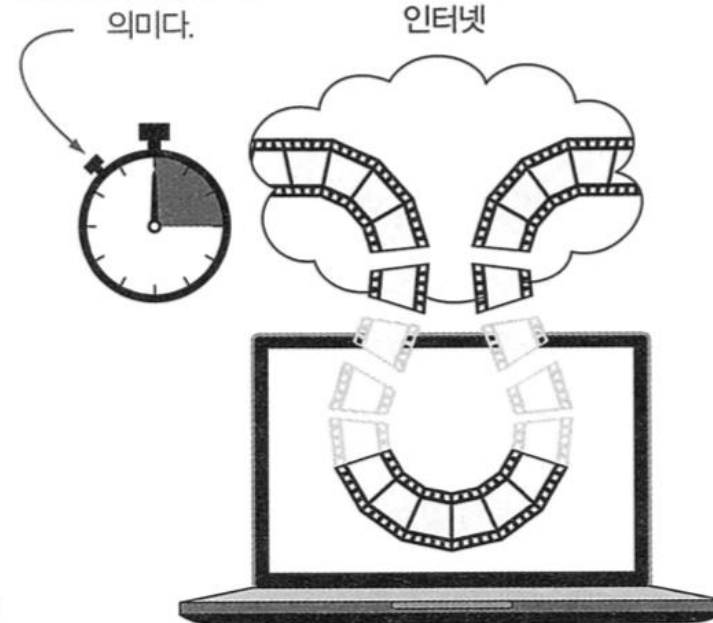
'적극적 생성'이란 모든 값을
계산할 때까지 기다린다는
의미다.



DVD처럼 컬렉션은 현재 자료구조에 포함된
모든 값을 계산한 다음에 컬렉션에 추가할 수
있다.

자바 8의 스트림은 인터넷으로 스트리밍하는
영화에 비유할 수 있다.

'게으른 생성'이란 필요할
때만 값을 계산한다는
의미다.



스트리밍 비디오처럼 필요할 때 값을
계산한다.

딱 한 번만 탐색할 수 있다.

반복자와 마찬가지로 스트림도 한 번만 탐색할 수 있다.

즉, 탐색된 스트림의 요소는 소비된다.

```
1 List<String> title = Arrays.asList("Java8", "In", "Action");  
2 Stream<String> s = title.stream()  
3   s.forEach(System.out::println);  
4   s.forEach(System.out::println);
```

← title의 각 단어를 출력

← java.lang.IllegalStateException:
스트림이 이미 소비되었거나 닫힘

컬렉션 인터페이스를 사용하려면 사용자가 직접 요소를 반복해야 한다.

이를 **외부 반복**이라 한다.

스트림 라이브러리는 (반복을 알아서 처리하고 결과 스트림값을 어딘가에 저장해주는)

내부 반복을 사용한다.

외부 반복과 내부 반복

컬렉션: for-each 루프를 이용하는 외부 반복

```
1 List<String> names = new ArrayList<>();
2 for(Dish dish: menu) {
3     names.add(dish.getName());
4 }
```

메뉴 리스트를 명시적으로 순차 반복한다.


이름을 추출해서 리스트에 추가한다.

컬렉션: 내부적으로 숨겨졌던 반복자를 사용한 외부 반복

```
1 List<String> names = new ArrayList<>();
2 Iterator<String> iterator = menu.iterator();
3 while(iterator.hasNext()) {
4     Dish dish = iterator.next();
5     names.add(dish.getName());
6 }
```

명시적 반복

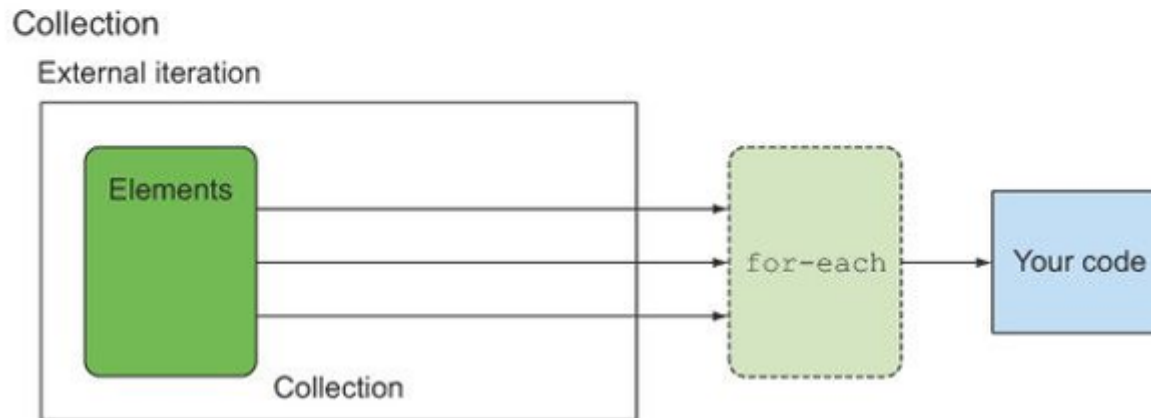
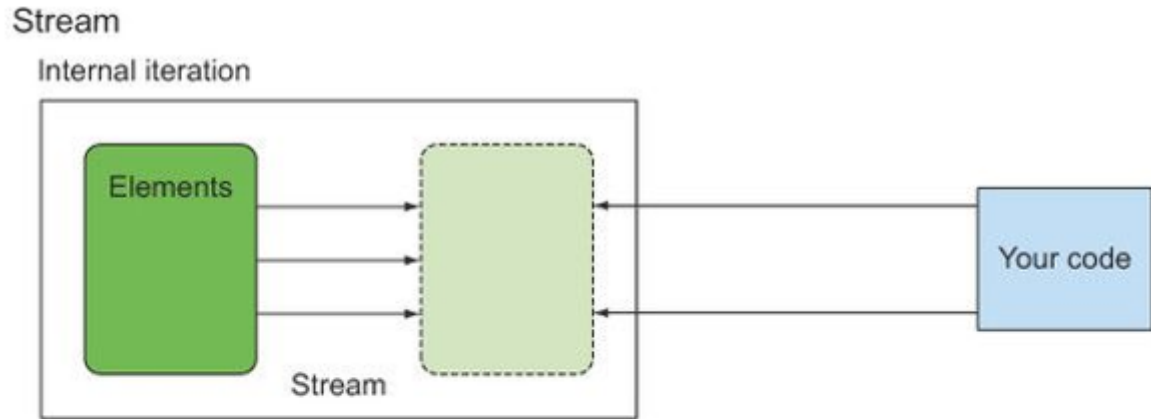
스트림: 내부 반복



```
1 List<String> names = menu.stream()  
2     .map(Dish::getName)  
3     .collect(toList());
```

map 메서드를 getName 메서드로 파라미터화해서 요리명을 추출한다.

파이프라인을 실행한다. 반복자는 필요없다.



- 내부 반복을 이용하면 작업을 투명하게 병렬로 처리하거나 더 최적화된 다양한 순서로 처리할 수 있다.
- 반면 for-each를 사용하는 외부 반복에서는 병렬성을 스스로 관리해야 한다.
- 병렬성을 포기하던지 synchronized로 시작해야한다. 힘들고 긴 전쟁을 시작함을 의미.

어떤 스트림 동작을 사용해 다음 코드를 리팩터링할 수 있을지 생각해보자.

```
1    List<String> highCaloricDishes = new ArrayList<>();
2    Iterator<Dish> iterator = menu.iterator();
3    while(iterator.hasNext()) {
4        Dish dish = iterator.next();
5        if(dish.getCalories() > 300) {
6            highCaloricDishes.add(dish.getName());
7        }
8    }
9    System.out.println(highCaloricDishes);
```

```
1 List<String> names = menu.stream()    // 메뉴에서 스트림을 얻는다.  
2                                     .filter(d -> d.getCalories() > 300) // 중간연산  
3                                     .map(Dish::getName) // 중간연산  
4                                     .limit(3) // 중간연산  
5                                     .collect(toList()); // 스트림을 리스트로 변환  
6  
7  
8
```

- filter, map, limit는 서로 연결되어 파이프라인을 형성한다.
- collect로 파이프라인을 실행한 다음에 닫는다.
- 연결할 수 있는 스트림 연산을 **중간 연산(intermediate operation)**이라고 하며, 스트림을 닫는 연산을 **최종 연산(terminal operation)**이라고 한다.

중간 연산

- filter나 sorted 같은 중간 연산은 다른 스트림을 반환.
- 여러 중간 연산을 연결해서 질의를 만들 수 있다.
- 중간 연산의 중요한 특징은 단말 연산을 스트림 파이프라인에 실행하기 전까지는 아무 연산도 수행하지 않는다는 것이다.(lazy)
- 중간 연산을 합친 다음에 합쳐진 중간 연산을 최종 연산으로 한번에 처리

```
1 List<Dish> menu = List.of(  
2     new Dish("pork", false, 800, Type.MEAT),  
3     new Dish("beef", false, 700, Type.MEAT),  
4     new Dish("chicken", false, 400, Type.MEAT),  
5     new Dish("french fries", false, 530, Type.OTHER),  
6     new Dish("rice", true, 350, Type.OTHER),  
7     new Dish("season", false, 120, Type.OTHER),  
8     new Dish("pizza", true, 550, Type.OTHER),  
9     new Dish("prawns", false, 300, Type.FISH),  
10    new Dish("salmon", false, 450, Type.FISH)  
11 );
```

```
1 List<String> names =  
2     menu.stream()  
3     .filter(d -> {  
4         System.out.println("filtering " + d.getName());  
5         return d.getCalories() > 300;  
6     })  
7     .map(d -> {  
8         System.out.println("mapping " + d.getName());  
9         return d.getName();  
10    })  
11    .limit(3)  
12    .collect(toList());  
13 System.out.println(names);
```

```
1 filtering pork  
2 mapping pork  
3 filtering beef  
4 mapping beef  
5 filtering chicken  
6 mapping chicken  
7 [pork, beef, chicken]
```


스트림의 게으른 특성 덕분에 얻은 최적화 효과

- 300 칼로리가 넘는 요리는 여러개지만 오직 처음 3개만 선택되었다.
- limit 연산을 통해 **쇼트서킷**이라 불리는 기법 덕분.

filter와 map은 서로 다른 연산이지만 한과정으로 병합되었다.

- 이 기법을 **루프 퓨전(loop fusion)** 이라고 한다.

표 4-1 중간 연산

연산	반환 형식	연산의 인수	함수 디스크립터
filter	Stream<T>	Predicate<T>	T -> Boolean
map	Stream<T>	Function<T, R>	T -> R
limit	Stream<T>		
sorted	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Stream<T>		

최종 연산

- 최종 연산은 스트림 파이프라인에서 결과를 도출한다.
- 보통 최종 연산에 의해 List, Integer, void 등 스트림 이외의 결과가 반환된다.

```
1 menu.stream().forEach(System.out::println);
```

menu에서 만든 스트림의 모든 요리를 출력한다.

표 4-2 최종 연산

연산	목적
forEach	스트림의 각 요소를 소비하면서 람다를 적용한다. void 를 반환한다.
count	스트림의 요소 개수를 반환한다. long 을 반환한다.
collect	스트림을 리듀스해서 리스트, 맵, 정수 형식의 컬렉션을 만든다.

- 스트림은 소스에서 추출된 연속 요소로, 데이터 처리 연산을 지원한다.
- 스트림은 내부 반복을 지원한다. 내부 반복은 filter, map, sorted 등의 연산으로 반복을 추상화 한다.
- 스트림에는 중간 연산과 최종 연산이 있다.
- 중간 연산은 filter와 map처럼 스트림을 반환하면서 다른 연산과 연결되는 연산이다. 중간 연산은 이용해서 파이프라인을 구성할 수 있지만 중간 연산으로는 어떤 결과도 생성할 수 없다.
- forEach나 count처럼 스트림 파이프라인을 처리해서 스트림이 아닌 결과를 반환하는 연산을 최종 연산이라고 한다.
- 스트림의 요소는 요청할 때 게으르게(lazily) 계산된다.