HF JAVA 3판 12장. 람다와 스트림

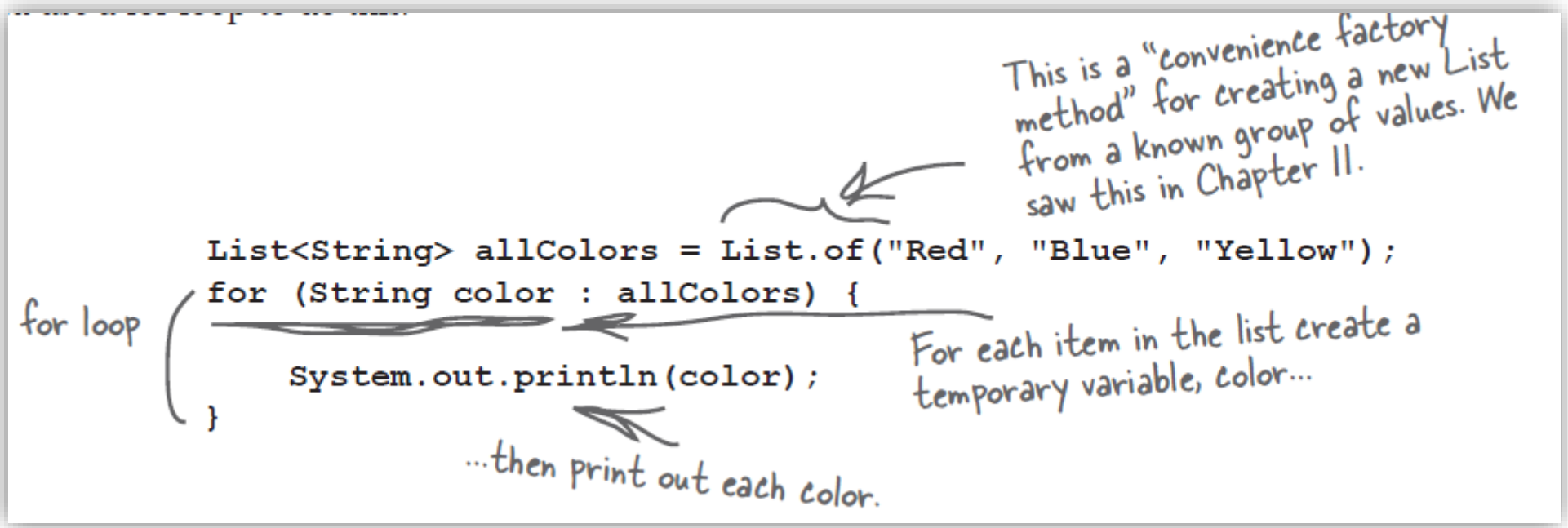
동의대학교 컴퓨터소프트웨어공학과

학습 목표

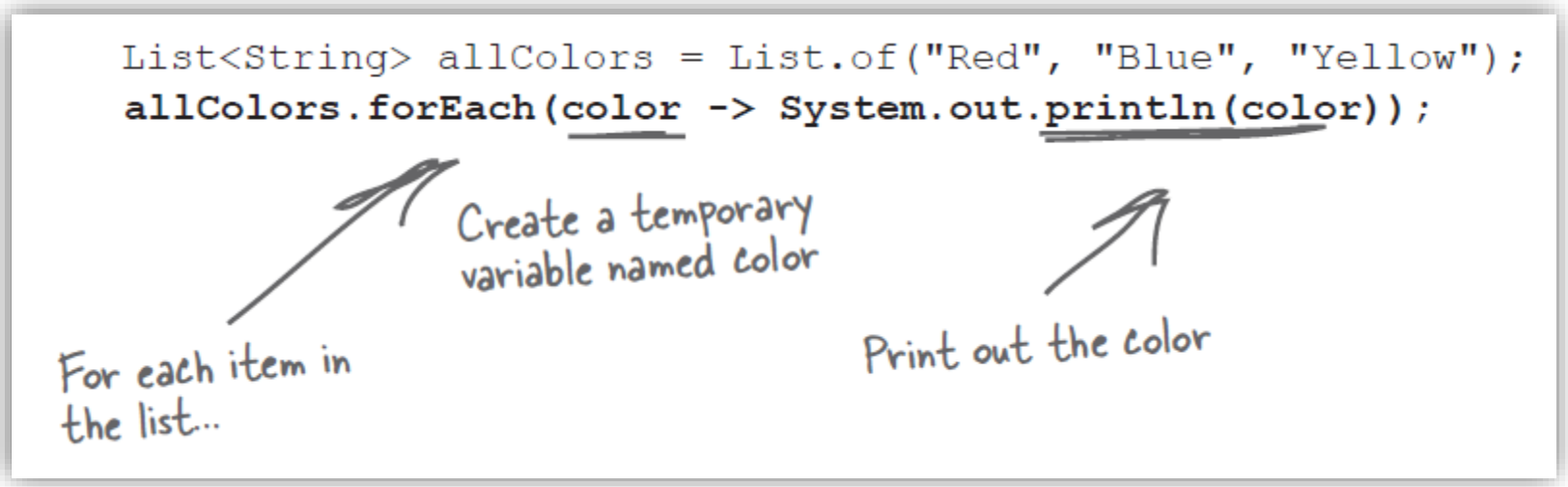
- Stream API 이해와 활용
  - Java 8에서 도입
  - 데이터 필터링, 변환, 집계 등의 작업을 간결하고 직관적인 코드로 작성하여 코드의 가독성과 유지보수성 향상
  - 람다 표현식과 함께 사용되어 기존의 복잡한 코드 구조를 단순화하고, 선언적 프로그래밍 방식을 채택하여 "무엇을 할지"에 집중할 수 있음.
  - 중간 연산은 실제로 종료 연산이 호출될 때까지 실행되지 않음.
  - 병렬 스트림을 사용하면 멀티 코어 환경에서 데이터를 병렬로 처리하여 성능 극대화 가능
- 람다 표현식의 이해와 활용
  - Java 8에서 도입
  - 익명 클래스의 복잡한 구문을 간단하게 표현할 수 있어 코드가 간결해짐.
  - 코드의 길이가 줄어들고, 함수의 의도가 명확하게 드러나므로 가독성 향상
  - 함수형 프로그래밍 스타일을 지원하여, 코드를 함수처럼 다룰 수 있음.
  - 스트림 API를 함께 사용하면 병렬 처리 가능

컴퓨터에게 HOW가 아니라 WHAT을 알려주자!

- 어떻게?
  - allColors에 있는 모든 color 원소에 대하여 color를 출력하시오.
  - enhanced for loop을 사용하여 해야 할 일을 하시오.



- 무엇을?
  - allColors의 각 원소 color에 대하여 color를 출력하시오.
  - 해야 할 일을 어떻게 하는지는 forEach() 메서드에서 알아서 해야 함. → for 문에 비해 간단!



for 문을 잘못 사용하면?

```
class MixForLoops {
  public static void main(String [] args) {
    List<Integer> nums = List.of(1, 2, 3, 4, 5);
    String output = "";
    
    System.out.println(output);
  }
}
```

Candidates:

```
for (int i = 1; i < nums.size(); i++)
    output += nums.get(i) + " ";
```

```
for (Integer num : nums)
    output += nums + " ";
```

```
for (int i = 0; i <= nums.length; i++)
    output += nums.get(i) + " ";
```

```
for (int i = 0; i <= nums.size(); i++)
    output += nums.get(i) + " ";
```

Possible output:

```
1 2 3 4 5
```

```
Compiler error
```

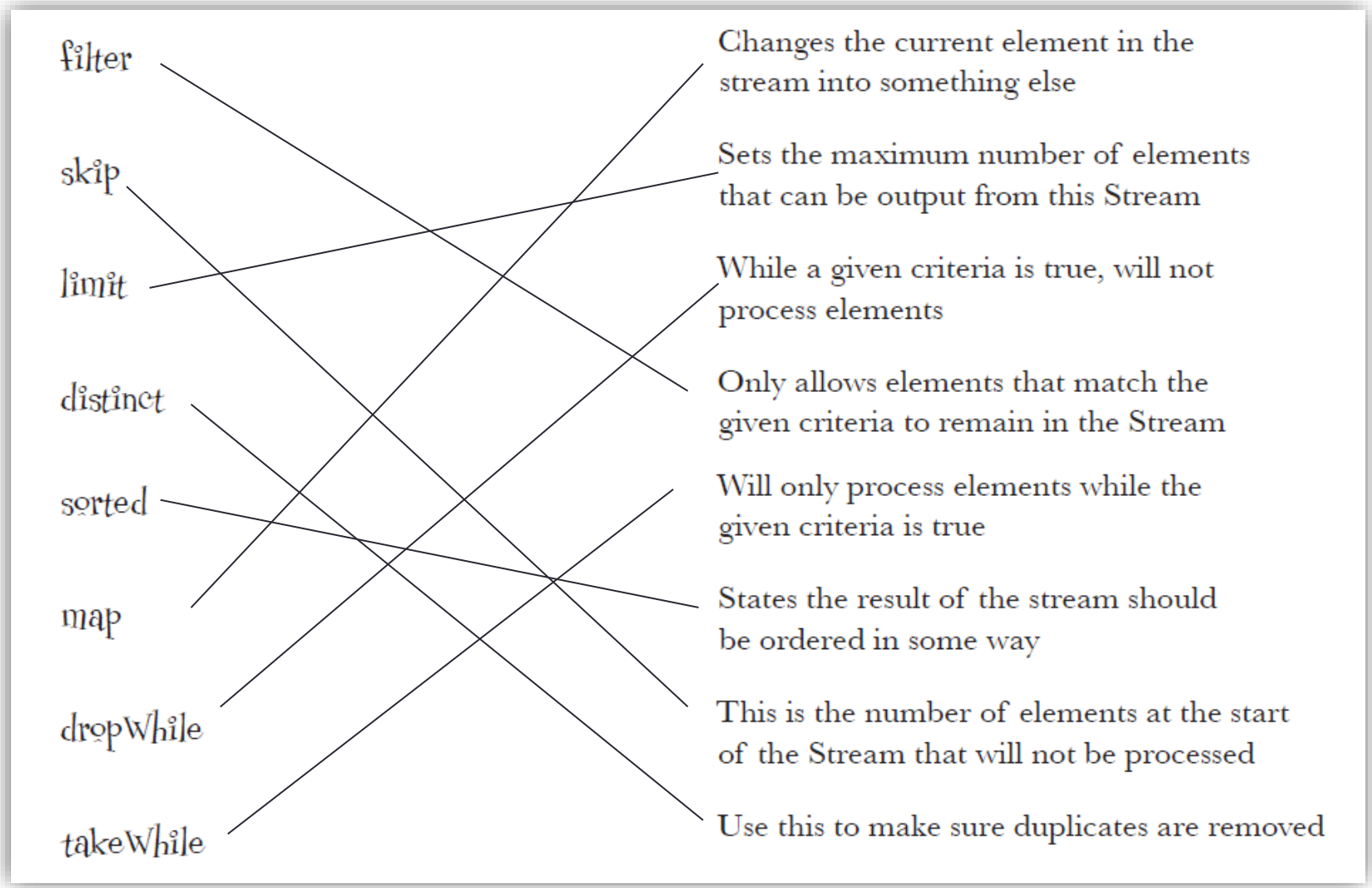
```
2 3 4 5
```

```
Exception thrown
```

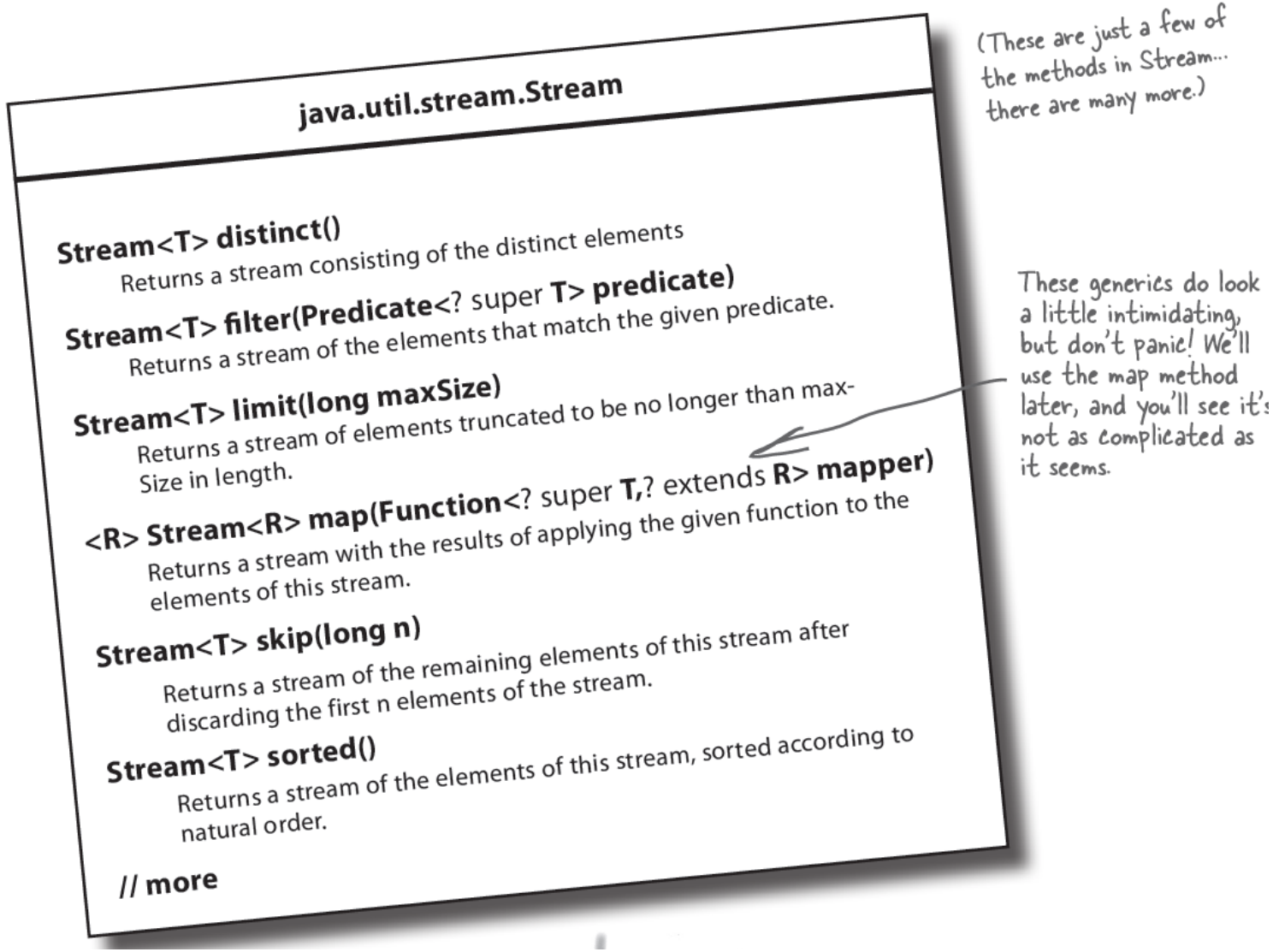
```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Match each candidate with one of the possible outputs

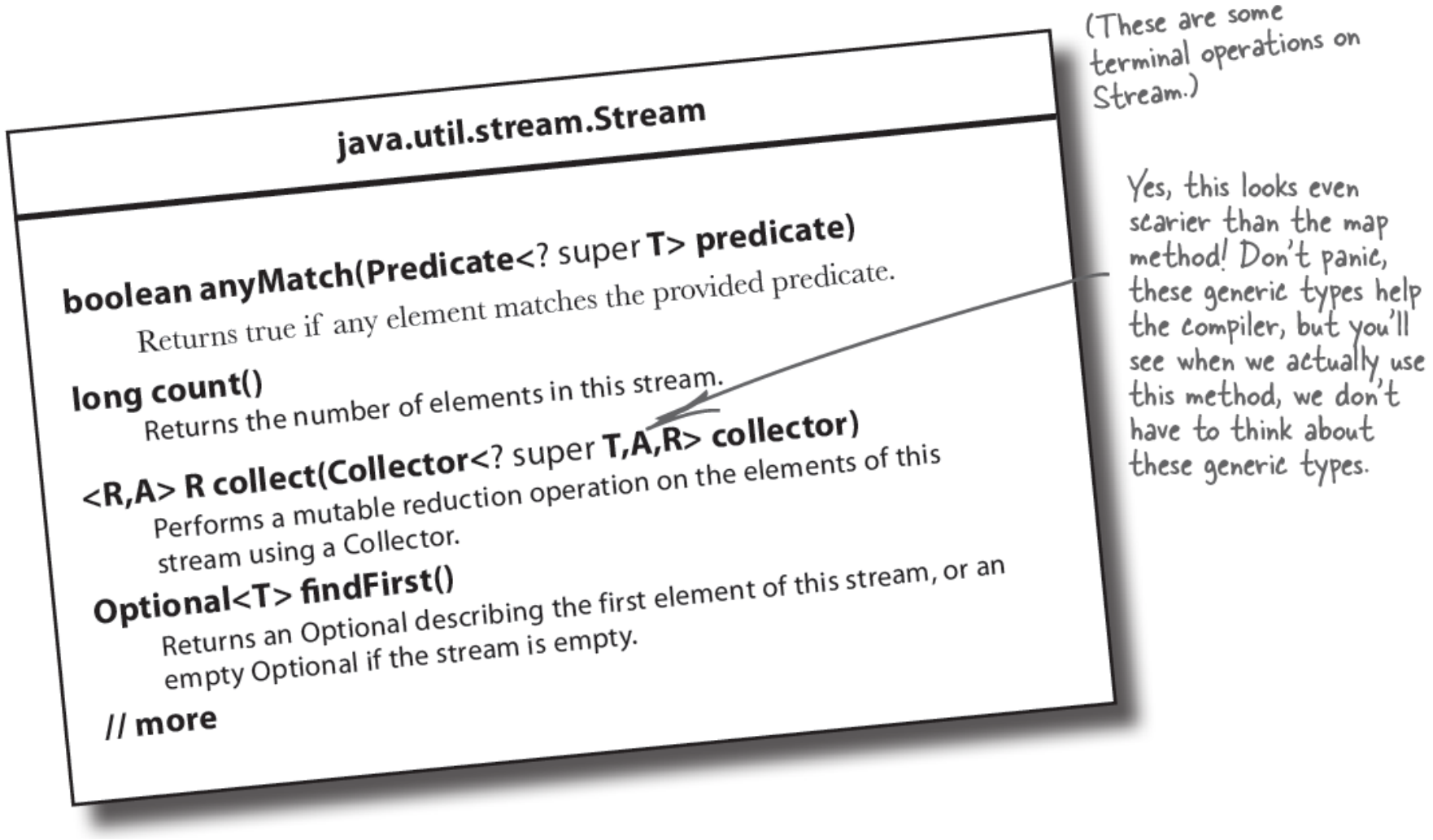
컬렉션의 기본 연산


filter
skip
limit
distinct
sorted
map
dropWhile
takeWhile
Changes the current element in the stream into something else
Sets the maximum number of elements that can be output from this Stream
While a given criteria is true, will not process elements
Only allows elements that match the given criteria to remain in the Stream
Will only process elements while the given criteria is true
States the result of the stream should be ordered in some way
This is the number of elements at the start of the Stream that will not be processed
Use this to make sure duplicates are removed

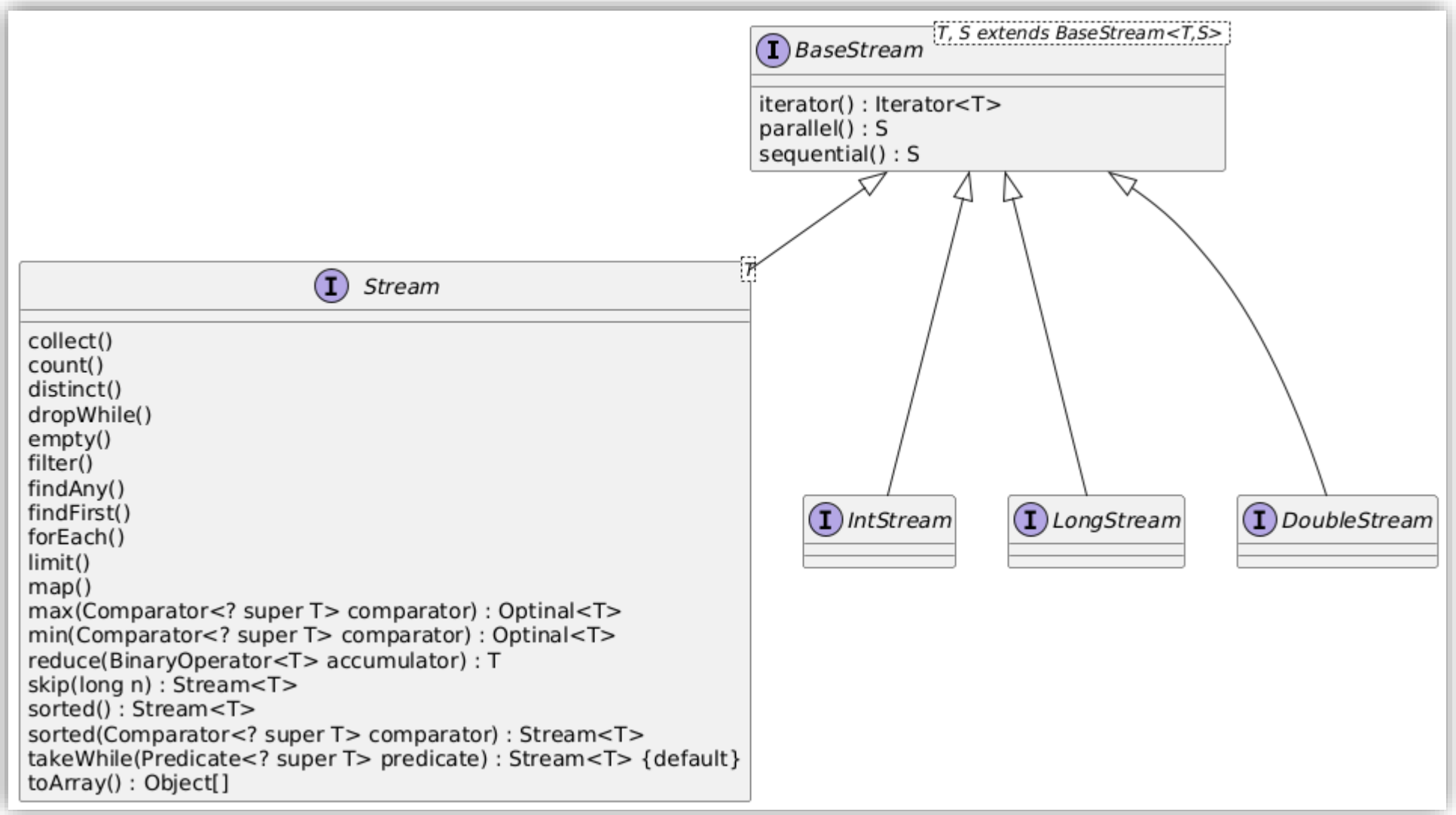
Stream API 소개: 중간 연산 (Intermediate Opeartion)



최종 연산 (Terminal Operation)



(참고) Stream 관련 인터페이스



LimitWithStream.java

```
5   package cse.oop2.ch12_3;
6
7   import java.util.List;
8   import java.util.stream.Collectors;
9   import java.util.stream.Stream;
10
11  /**
12   * pp.418-420
13   *
14   * @author Prof. Jong Min Lee
15   */
16  public class LimitWithStream {
17
18      /**
19       * @param args the command line arguments
20       */
21      public static void main(String[] args) {
22          new LimitWithStream().go();
23      }
```

```
25      public void go() {
26          // p.418
27          List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");
28          Stream<String> stream = strings.stream();
29          Stream<String> limit = stream.limit(4);
30          System.out.println("limit = " + limit);
31
32          // p.420 - 만들어진 limit 스트림의 원소 개수 반환
33          // Stream 인터페이스의 limit()은 중간 연산으로 0개 이상 호출 가능함.
34          // 중간 연산은 최종 연산 호출시 사슬처럼 연결되어 실행됨.
35          // Stream 인터페이스의 count()와 collect()는 최종 연산임.
36          long result1 = limit.count();
37          System.out.println("result1 = " + result1);
38
39          // p.420 - 스트림은 한번 사용하면 다시 사용 불가하므로
40          // 다시 스트림을 만들어 사용해야 함.
41          Stream<String> stream2 = strings.stream();
42          Stream<String> limit2 = stream2.limit(4);
43          List<String> result2 = limit2.collect(Collectors.toList());
44          System.out.println("result2 = " + result2);
```

```
46              // p.421 스트림을 연속으로 사용하는 예
47              // Collectors.toUnmodifiableList()를 사용하여 불변 리스트를 만들 수도 있음.
48              // Collectors.toSet(); Collectors.toUnmodifiableSet(), Collectors.toMap() 도 있음.
49              List<String> result3 = strings.stream()
50                      .limit(4)
51                      .collect(Collectors.toList());
52              System.out.println("result3 = " + result3);
53          }
54
55      }
```

```
--- exec:1.5.0:exec (default-cli) @ java_maven ---
limit = java.util.stream.SliceOps$1@31befd9f
result1 = 4
result2 = [I, am, a, list]
result3 = [I, am, a, list]
```

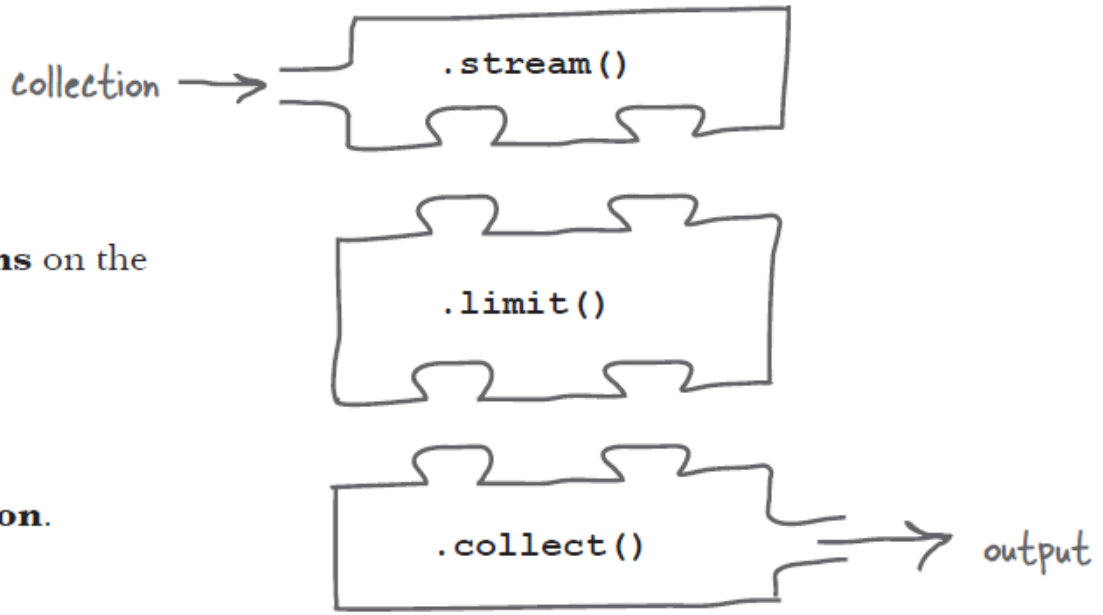
Stream 연산 동작 방식

- List와 Set 객체는 java.util.Collection 인터페이스의 stream() 메서드 이용하여 스트림으로 만들 수 있음.
- Map 객체는 java.util.Collection 인터페이스를 구현하지 않아 직접적으로 stream() 메서드를 이용할 수는 없음. map.entrySet().stream()을 사용하여 간접적으로 스트림으로 만들 수 있음.

① Get the Stream from a **source** collection.

② Call zero or more **intermediate operations** on the Stream.

③ Output the results with a **terminal operation**.



- **중간 연산**: 스트림에 대해 작용하며, 스트림을 반환함.
- **최종 연산**: 스트림에 대한 결과를 반환
- **스트림 파이프라인(stream pipeline)**: 소스(source), 중간 연산, 최종 연산이 모두 결합되어 이루는 것으로 원래 컬렉션에 대한 질의(query)를 나타냄.

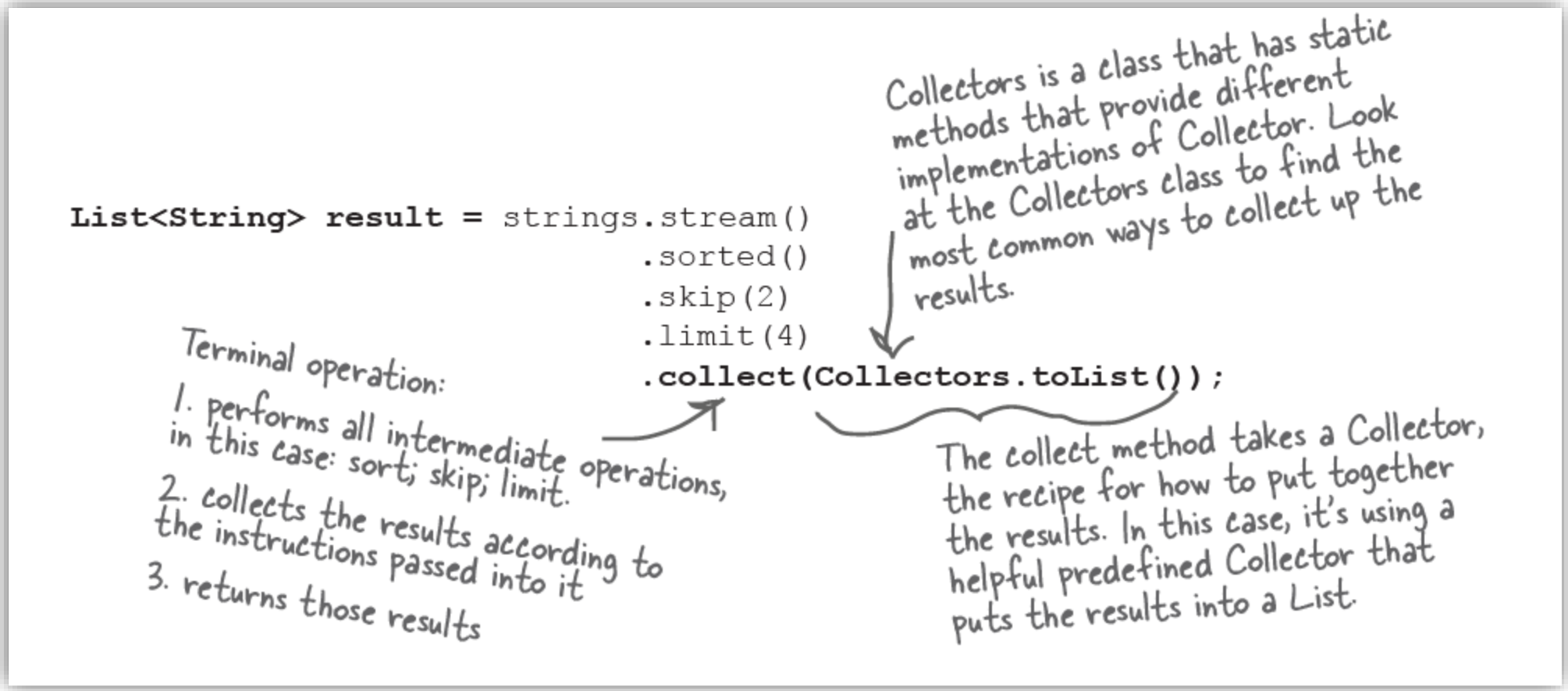
```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

List<String> result = strings.stream()

                          .sorted()

                          .limit(4)

                          .collect(Collectors.toList());

System.out.println("result = " + result);
```

Sort what's in the stream (not the original collection), using natural order, before limiting the results.

Limit the stream to just four elements.

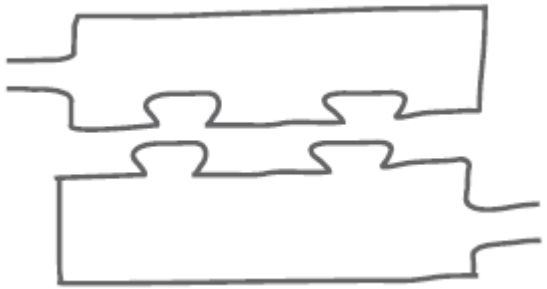
```
%java ChainedStream

result = [I, Strings, a, am]
```

List로 모으기


List<String> result = strings.stream()
.sorted()
.skip(2)
.limit(4)
.collect(Collectors.toList());
Collectors is a class that has static methods that provide different implementations of Collector. Look at the Collectors class to find the most common ways to collect up the results.
Terminal operation:
1. performs all intermediate operations, in this case: sort; skip; limit.
2. collects the results according to the instructions passed into it
3. returns those results
The collect method takes a Collector, the recipe for how to put together the results. In this case, it's using a helpful predefined Collector that puts the results into a List.

Stream 활용 가이드라인

- 스트림 파이프라인을 만들기 위해서는 적어도 첫 번째 조각과 마지막 조작은 있어야 함.



- 스트림은 재사용 불가!

```
Stream<String> limit = strings.stream()
                              .limit(4);
List<String> result = limit.collect(Collectors.toList());
List<String> result2 = limit.collect(Collectors.toList());
```

- 스트림이 작동하는 동안 기본 컬렉션을 바꿀 수는 없음.

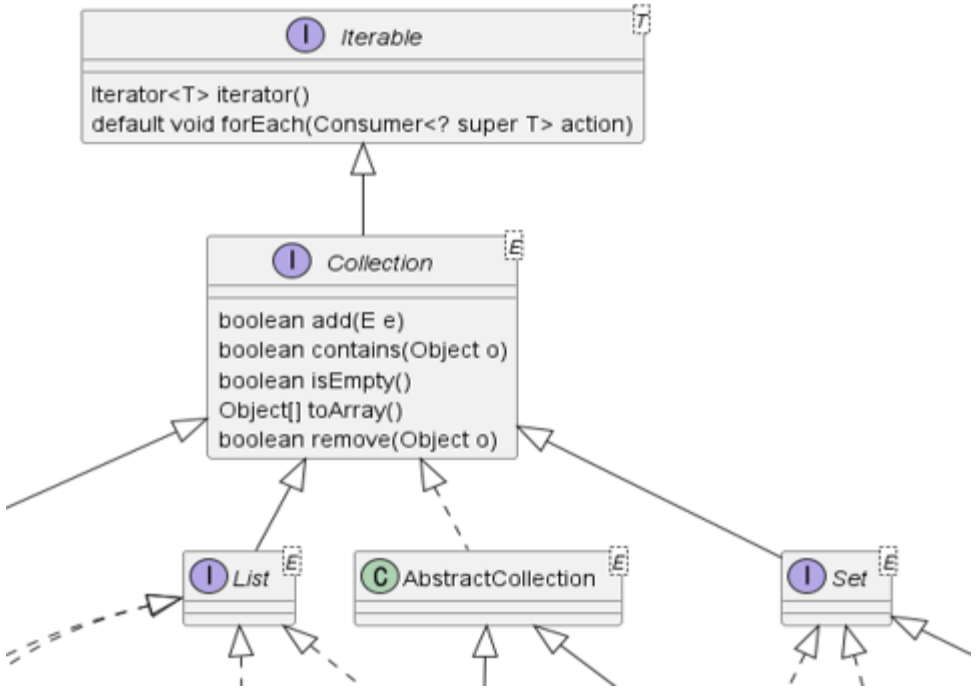
```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

Stream<String> limit = strings.stream()
                              .limit(4)
                              .collect(Collectors.toList());
System.out.println("strings = " + strings);
System.out.println("result = " + result);
```

```
%java LimitWithStream

strings = [I, am, a, list, of, Strings]
result = [I, am, a, list]
```

람다 표현식

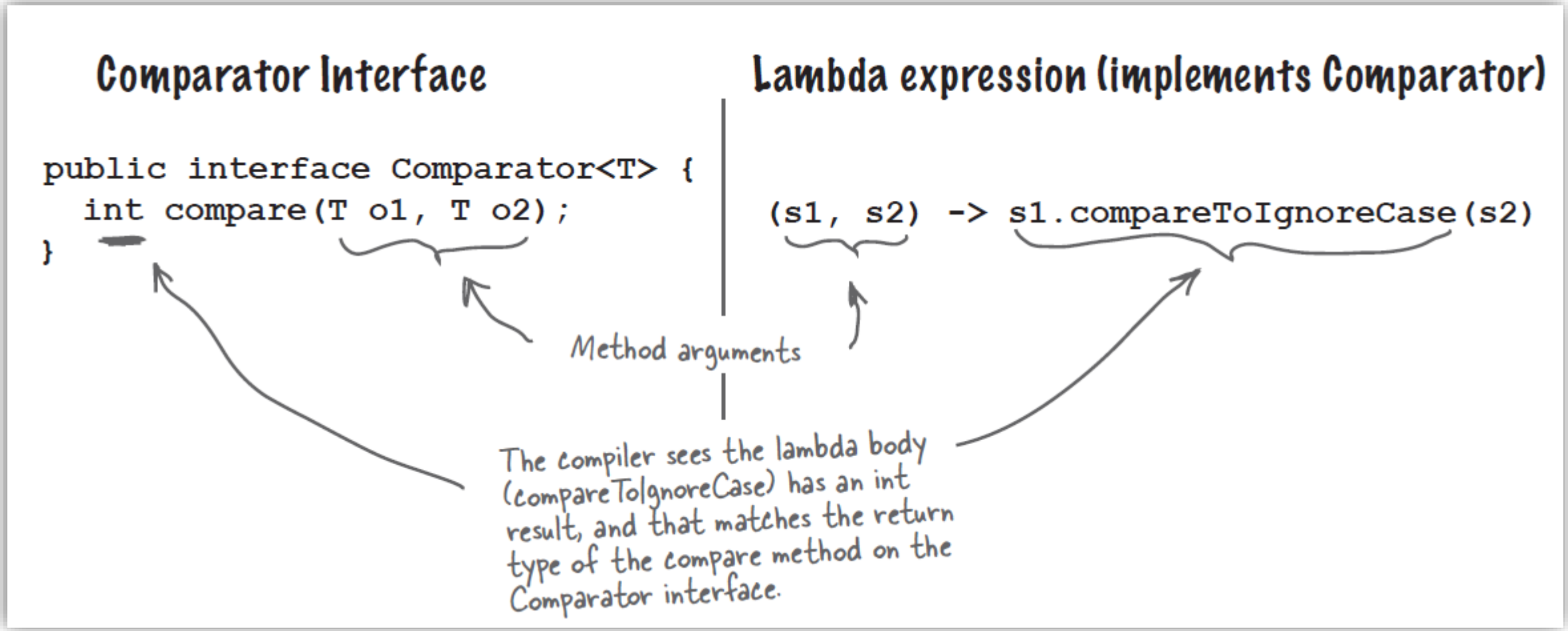
- 람다 표현식 사용 예
  - List<String> data = ...;
  - data.forEach(item -> System.out.println(item));

- Iterable 인터페이스의 forEach() 기본 메서드
  - default void forEach(Consumer<? super T> action)

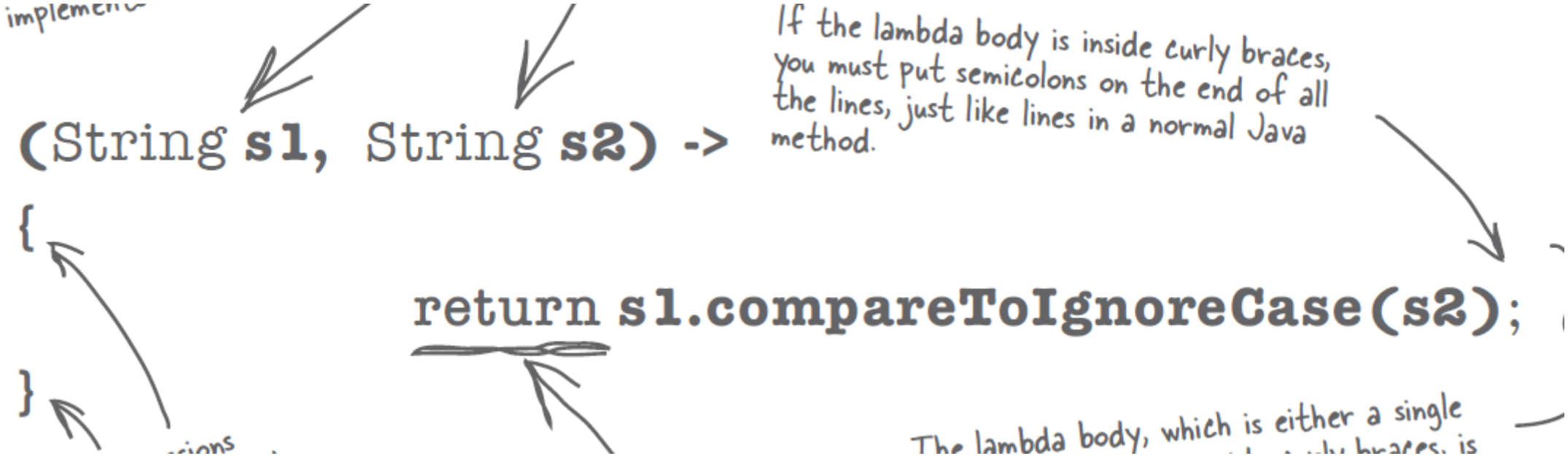


- **람다 표현식은 함수형 인터페이스를 구현한 것임.**
- 함수형 인터페이스(functional interface)는 Single Abstract Method (SAM)를 가진다!!!
  - Java 8부터는 인터페이스에 기본(default) 메서드와 정적(static) 메서드 포함 가능해짐. 즉, SAM 외에 여러 개의 기본 메서드와 정적 메서드가 들어 있어도 함수형 인터페이스라고 할 수 있음.

람다 표현식의 모양


Comparator Interface
Lambda expression (implements Comparator)
public interface Comparator<T> {
int compare(T o1, T o2);
}
(s1, s2) -> s1.compareToIgnoreCase(s2)
Method arguments
The compiler sees the lambda body (compareToIgnoreCase) has an int result, and that matches the return type of the compare method on the Comparator interface.

람다 표현식의 구조



람다에서 어떤 값을 반환하는 메서드를 오버라이드한다면
람다 본체 끝에 return 문 있어야 함.

(참고) forEach() 메서드

- ArrayList 클래스의 forEach 메서드 시그니쳐(signature)
  public void forEach(Consumer<? super E> action)

- Consumer 함수형 인터페이스
  @FunctionalInterface public interface Consumer<T>

Implementation Requirements:

The default implementation behaves as if:

```
for (T t : this)
    action.accept(t);
```

Modifier and Type	Method	Description
`void`	`accept(T t)`	Performs this operation on the given argument.
`default Consumer<T>`	`andThen(Consumer<? super T> after)`	Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

LambdaTest.java*

```
7  import java.util.ArrayList;
8  import java.util.List;
9  import java.util.function.Consumer;
10
11 /**...4 lines */
15 public class LambdaTest {
16
17     /**
18      * @param args the command line arguments
19      */
20     public static void main(String[] args) {
21         Runnable runnable = () -> System.out.println("Hello, World!");
22         runnable.run();
23
24         // Consumer 인터페이스는 스트림 API, 비동기 작업 처리, 그리고
25         // 일반적인 콜백 패턴에 모두 활용될 수 있어, 현대 Java 개발에서 매우 유용한 도구임.
26         Consumer<String> consumer = str -> System.out.println(str);
27         consumer.accept("Hello, world!!");
```

@FunctionalInterface public interface Runnable
void run() : Runs this operation.

```
29          Consumer<String> consumer2 = str -> System.out.println(str);
30          Consumer<String> consumer3 = str -> System.out.println(str + "!!!");
31          Consumer<String> consumer4 = consumer2.andThen(consumer3);
32          consumer4.accept("Test");
33
34          List<String> dataList = new ArrayList<>();
35          dataList.add("Hello");
36          dataList.add("world");
37          dataList.add("This");
38          dataList.add("is");
39          dataList.add("a");
40          dataList.add("test");
41          dataList.forEach(item -> System.out.println(item));   == new MyConsumer()
42
43          dataList.forEach(new MyConsumer());
44
45      }
46
47  }
48
49  class MyConsumer implements Consumer<String> {
50
51      @Override
53      public void accept(String t) {
54          System.out.println(t);
55      }
56
57  }
```

실행 결과

```
--- exec:1.5.0:exec (
Hello, World!
Hello, world!!
Test
Test!!!
Hello
world
This
is
a
test
Hello
world
This
is
a
test
```

람다 표현식의 형식*

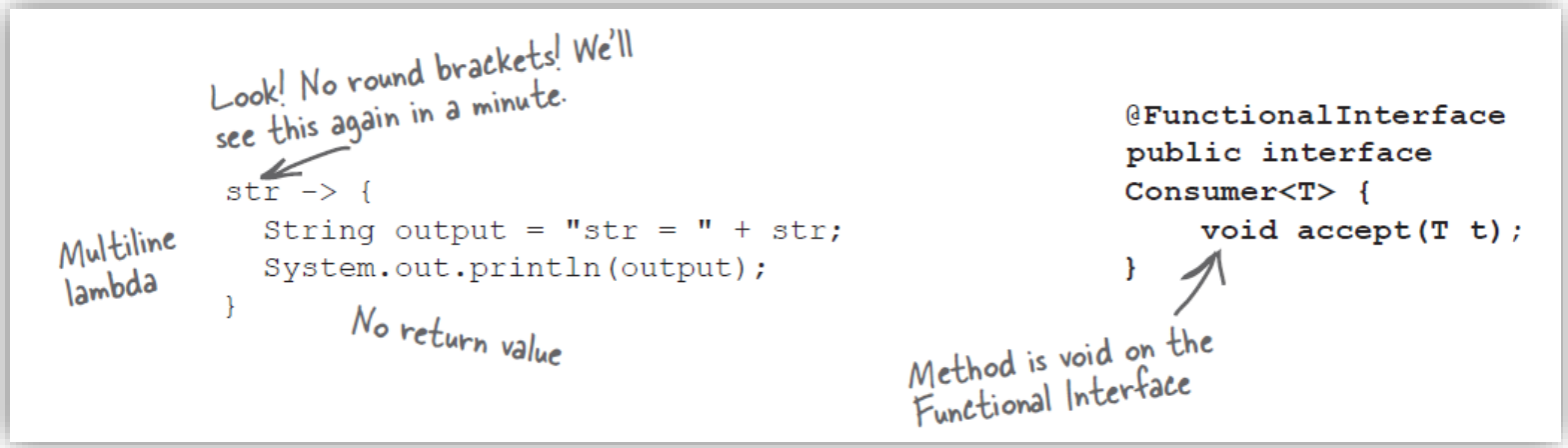
```
(str1, str2) -> {
    int len1 = str1.length();
    int len2 = str2.length();
    return len2 - len1;
}
```

```
(str1, str2) -> str2.length() – str1.length()
```

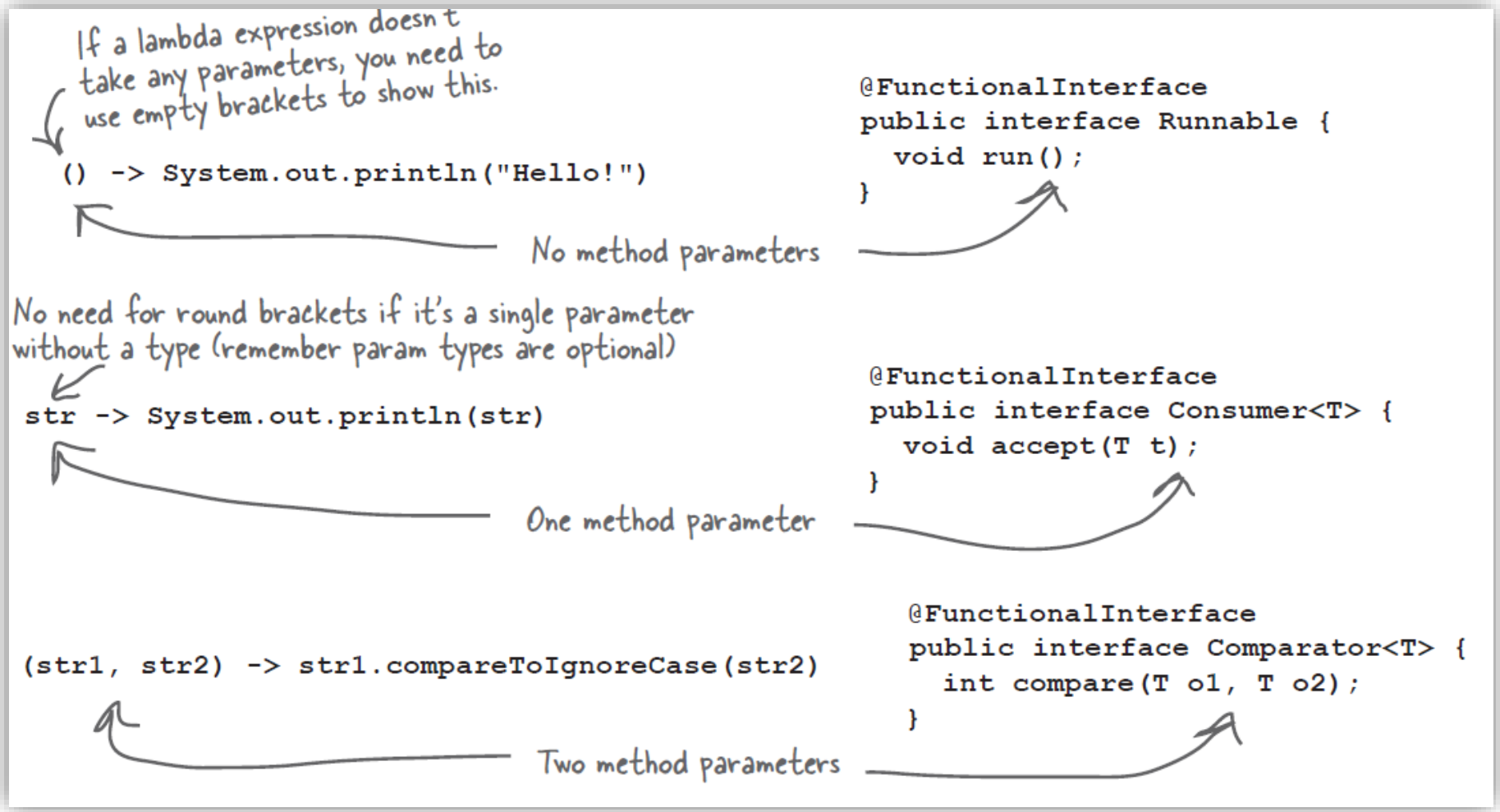
- 두 줄 이상일 경우 중괄호('{', '}')를 사용
- 메서드 내에서 반환해야 하면 return 문 사용 가능

- 한 줄 짜리 람다는 중괄호 없어도 됨.
  - return을 명시적으로 하지 않더라도 컴파일러가 알아서 반환해줌.

- 람다에서는 반환값이 없어도 됨
  - 함수형 인터페이스의 반환값 자료형이 void인 경우



- 람다의 매개변수에는 제한 없음.



Stream API의 filter() 메서드

- Stream<T> filter(Predicate<? super T> predicate)
  - @FunctionalInterface public interface Predicate<T>

Modifier and Type	Method	Description
default Predicate<T>	and(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>	isEqual(Object targetRef)	Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).
default Predicate<T>	negate()	Returns a predicate that represents the logical negation of this predicate.
static <T> Predicate<T>	not(Predicate<? super T> target)	Returns a predicate that is the negation of the supplied predicate.
default Predicate<T>	or(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	test(T t)	Evaluates this predicate on the given argument.

- 예제

```
List<String> data = List.of("This", "is", "a", "test", "Hello", "world", "Hi");
List<String> stream1 = data.stream()
                           .filter(e -> e.toLowerCase().startswith("h"))
                           .collect(Collectors.toList());


List<String> stream2 = data.stream().filter(new MyPredicate()).collect(Collectors.toList());

class MyPredicat implements Predicate<String> {
    public Boolean test(String t) {
        return t.toLowerCase().startsWith("h"));
    }
}
```

@FunctionalInterface

- 함수형 인터페이스를 나타내는 애노테이션(annotation) : @FunctionalInterface
  - (참고) ll → small L, large i
  - 옛날 코드에는 @FunctionalInterface 애노테이션을 사용하지 않는 경우도 있음.

- 함수형 인터페이스
  - **한 개의 추상 메서드**(**SAM**; Single Abstract Method)를 가지는 클래스
  - Java 8이후로는 추가적으로 정적 메서드와 기본(default) 메서드 가질 수 있음.

- (참고) **인터페이스**
  - 모든 메서드가 추상 메서드인 클래스
  - Java 8이후로는 추가적으로 정적 메서드와 기본(default) 메서드 가질 수 있음.

Song.java*

```
5   package cse.oop2.ch12_3;
6
7   public class Song {
8
9       private final String title;
10      private final String artist;
11      private final String genre;
12      private final int year;
13      private final int timesPlayed;
14
15      public Song(String title, String artist, String genre, int year, int timesPlayed) {
16          this.title = title;
17          this.artist = artist;
18          this.genre = genre;
19          this.year = year;
20          this.timesPlayed = timesPlayed;
21      }
```

```
23      public int getTimesPlayed() {
24          return timesPlayed;
25      }
26
27      public String getTitle() {
28          return title;
29      }
30
31      public String getArtist() {
32          return artist;
33      }
34
35      public String getGenre() {
36          return genre;
37      }
38
39      public int getYear() {
40          return year;
41      }
```

```
43      @Override
45      public String toString() {
46          return title + ", "
47                  + artist + ", "
48                  + genre + ", "
49                  + year + ", "
50                  + timesPlayed + "\n";
51      }
52
```

JukeboxStreams.java*

```
5   package cse.oop2.ch12_3;
6
7   import java.util.Comparator;
8   import java.util.List;
9   import java.util.Optional;
10  import java.util.Set;
11  import java.util.function.Function;
12  import java.util.stream.Collectors;
13
14  public class JukeboxStreams {
15
16      public static void main(String[] args) {
17          List<Song> songs = new Songs().getSongs();
18
19          // p.445 Rock 장르 노래 목록
20          List<Song> rockSongs = songs.stream()
21                  .filter(song -> song.getGenre().contains("Rock"))
22                  .collect(Collectors.toList());
23          System.out.println(rockSongs);
```

```
[Cassidy, Grateful Dead, Rock, 1972, 123
, 50 ways, Paul Simon, Soft Rock, 1975, 199
, Hurt, Nine Inch Nails, Industrial Rock, 1995, 257
, Hurt, Johnny Cash, Soft Rock, 2002, 392
, The Outsider, A Perfect Circle, Alternative Rock, 2004, 312
, With a Little Help from My Friends, The Beatles, Rock, 1967, 168
, Come Together, Ike & Tina Turner, Rock, 1970, 165
, With a Little Help from My Friends, Joe Cocker, Rock, 1968, 46
, Immigrant Song, Karen O, Industrial Rock, 2011, 12
, I am not a woman, I'm a god, Halsey, Alternative Rock, 2021, 384
, Immigrant song, Led Zeppelin, Rock, 1970, 484
]
```

```
25          // p.447 노래 장르 목록
26          List<String> genres = songs.stream()
27                  .map(song -> song.getGenre())
28                  .collect(Collectors.toList());
29          System.out.println(genres);
30
31          // p.449-1 장르 중복 제거
32          List<String> genres2 = songs.stream()
33                  .map(song -> song.getGenre())
34                  .distinct()
35                  .collect(Collectors.toList());
36          System.out.println(genres2);
37
38          // p.449-2 비틀즈의 "With a Little ..." 노래를 커버한 적이 있는 가수 이름 목록
39          String songTitle = "With a Little Help from My Friends";
40          List<String> result = songs.stream()
41                  .filter(song -> song.getTitle().equals(songTitle))
42                  .map(song -> song.getArtist())
43                  .filter(artist -> !artist.equals("The Beatles"))
44                  .collect(Collectors.toList());
45          System.out.println(result);
```

```
[Electronic, R&B, Rock, Soft Rock, Industrial Rock, Electronic, Soft Rock, Electronic, Alternative Rock, Rock
[Electronic, R&B, Rock, Soft Rock, Industrial Rock, Alternative Rock, Blues rock, Pop, Latin]
[Joe Cocker]
```

```
47          // p.450 함수형 인터페이스 Function 사용
48          Function<Song, String> getGenre = song -> song.getGenre();
49          List<String> genres3 = songs.stream()
50                  .map(getGenre)
51                  .distinct()
52                  .collect(Collectors.toList());
53          System.out.println(genres3);
54
55          // p.450 람다 표현식 대신 Method reference(메서드 레퍼런스) 사용
56          Function<Song, String> getGenre4 = Song::getGenre;
57          List<String> genres4 = songs.stream()
58                  .map(getGenre4) //  or .map(Song::getGenre)
59                  .distinct()
60                  .collect(Collectors.toList());
61          System.out.println(genres4);
62
63          // p.450 메서드 레퍼런스와 Comparator 인터페이스의 comparingInt() 등 정적 메서드 결합
64          List<Song> result2 = songs.stream()
65                  .sorted(Comparator.comparingInt(Song::getYear)) // <==
66                  .collect(Collectors.toList());
67          System.out.println(result2);
```

```
[Electronic, R&B, Rock, Soft Rock, Industrial Rock, Alternative Rock, Blues rock, Pop, Latin]
[Electronic, R&B, Rock, Soft Rock, Industrial Rock, Alternative Rock, Blues rock, Pop, Latin]
[With a Little Help from My Friends, The Beatles, Rock, 1967, 168
, Come Together, The Beatles, Blues rock, 1968, 173
, With a Little Help from My Friends, Joe Cocker, Rock, 1968, 46
, Come Together, Ike & Tina Turner, Rock, 1970, 165
, Immigrant song, Led Zeppelin, Rock, 1970, 484
, What's Going On, Gaye, R&B, 1971, 420
, Cassidy, Grateful Dead, Rock, 1972, 123
, 50 ways, Paul Simon, Soft Rock, 1975, 199
, Hurt, Nine Inch Nails, Industrial Rock, 1995, 257
, Breathe, The Prodigy, Electronic, 1996, 337
, Silence, Delerium, Electronic, 1999, 134
```

```
69          // p.451 Set 객체 생성
70          Set<String> genres5 = songs.stream()
71                  .map(song -> song.getGenre())
72                  .collect(Collectors.toSet());
73          System.out.println(genres5);
74
75          // p.451 Collectors.joining() 관련 메서드 사용하여 String 만들기
76          // CSV 문자열 만들 때 유용한 기능.
77          String joinedString = songs.stream()
78                  .map(Song::getGenre)
79                  .distinct()
80                  .collect(Collectors.joining(","));
81          System.out.println(joinedString);
82
83          // p.452 원하는 정보 존재 여부 확인
84          boolean exists = songs.stream()
85                  .anyMatch(s -> s.getGenre().equals("R&B"));
86          System.out.println("R&B exists: " + exists);
87
88          // p.451 특정 항목 찾기
89          Optional<Song> songOf1995 = songs.stream()
90                  .filter(s -> s.getYear() == 1995)
91                  .findFirst();
92          if (songOf1995.isPresent()) {
93              System.out.println("Song of 1995 = " + songOf1995.get().toString());
94          }
```

```
[Pop, Soft Rock, Rock, Alternative Rock, Latin, Electronic, Industrial Rock, Blues rock, R&B]
Electronic,R&B,Rock,Soft Rock,Industrial Rock,Alternative Rock,Blues rock,Pop,Latin
R&B exists: true
Song of 1995 = Hurt, Nine Inch Nails, Industrial Rock, 1995, 257
```

```
96              // p.451 항목 개수 세기
97              long numberOfDistinctSongs = songs.stream()
98                      .map(Song::getArtist)
99                      .distinct()
100                     .count();
                System.out.println("유일한 아티스트 수 = " + numberOfDistinctSongs);
102
103         }
104     }
```

```
유일한 아티스트 수 = 21
```

```
106     class Songs {
107
108         public List<Song> getSongs() {
109             return List.of(
110                     new Song("$10", "Hitchhiker", "Electronic", 2016, 183),
111                     new Song("Havana", "Camila Cabello", "R&B", 2017, 324),
112                     new Song("Cassidy", "Grateful Dead", "Rock", 1972, 123),
113                     new Song("50 ways", "Paul Simon", "Soft Rock", 1975, 199),
114                     new Song("Hurt", "Nine Inch Nails", "Industrial Rock", 1995, 257),
115                     new Song("Silence", "Delerium", "Electronic", 1999, 134),
116                     new Song("Hurt", "Johnny Cash", "Soft Rock", 2002, 392),
117                     new Song("Watercolour", "Pendulum", "Electronic", 2010, 155),
118                     new Song("The Outsider", "A Perfect Circle", "Alternative Rock", 2004, 312),
119                     new Song("With a Little Help from My Friends", "The Beatles", "Rock", 1967, 168),
120                     new Song("Come Together", "The Beatles", "Blues rock", 1968, 173),
121                     new Song("Come Together", "Ike & Tina Turner", "Rock", 1970, 165),
122                     new Song("With a Little Help from My Friends", "Joe Cocker", "Rock", 1968, 46),
123                     new Song("Immigrant Song", "Karen O", "Industrial Rock", 2011, 12),
124                     new Song("Breathe", "The Prodigy", "Electronic", 1996, 337),
125                     new Song("What's Going On", "Gaye", "R&B", 1971, 420),
126                     new Song("Hallucinate", "Dua Lipa", "Pop", 2020, 75),
127                     new Song("Walk Me Home", "P!nk", "Pop", 2019, 459),
128                     new Song("I am not a woman, I'm a god", "Halsey", "Alternative Rock", 2021, 384),
129                     new Song("Pasos de cero", "Pablo Albor□n", "Latin", 2014, 117),
130                     new Song("Smooth", "Santana", "Latin", 1999, 244),
131                     new Song("Immigrant song", "Led Zeppelin", "Rock", 1970, 484));
132         }
133
134     }
```

요약

- java.util.stream.Stream 인터페이스 소개
  - 주요 메서드: distinct(), filter(), limit(), map(), skip(), sorted(), anyMatch(), count(), collect(), findFirst(), findLast() 등
- 스트림 파이프라인: 소스(source), 중간 연산(intermediate operation), 최종 연산(terminal operation)
- 스트림 활용 가이드라인: 소스와 최종 연산 필요, 스트림 재사용 불가, 소스 수정 불가
- 람다 표현식: 함수형 인터페이스(SAM) 구현
- Java 8 이후로는 인터페이스에 기본 메서드와 정적 메서드 사용 가능해짐.
- 리스트에 있는 데이터에 대하여 스트림을 활용하여 필요한 연산 수행
  - filter(), map(), reduce(), sorted(), count(), collect(), sum() 등