4. 람다 활용

#1.인강/0.자바/7.자바-고급3편

필터 만들기1

람다를 처음 사용하면, 대부분 바로바로 람다를 사용하기는 어렵다. 람다에 익숙해지는데 어느정도의 시간이 걸린다.

이번 시간에는 람다를 활용하는 방법들을 알아보고, 또 다양한 문제를 풀어보면서 람다에 익숙해지는 시간을 가져보자.

필터1

먼저 람다를 사용하지 않고, 짝수만 거르기, 홀수만 거르기 메서드를 각각 따로 작성해보자.

```
package lambda.lambda5.filter;

import java.util.ArrayList;
import java.util.List;

public class FilterMainV1 {

    public static void main(String[] args) {
        List<Integer> numbers = List.of(1,2,3,4,5,6,7,8,9,10);

        // 짝수만 거르기
        List<Integer> evenNumbers = filterEvenNumber(numbers);
        System.out.println("evenNumbers = " + evenNumbers);
```

```
        // 홀수만 거르기
        List<Integer> oddNumbers = filterOddNumber(numbers);
        System.out.println("oddNumbers = " + oddNumbers);
    }

    static List<Integer> filterEvenNumber(List<Integer> numbers) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer number : numbers) {
            boolean testResult = number % 2 == 0;
            if (testResult) {
                filtered.add(number);
            }
        }
        return filtered;
    }

    static List<Integer> filterOddNumber(List<Integer> numbers) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer number : numbers) {
            boolean testResult = number % 2 == 1;
            if (testResult) {
                filtered.add(number);
            }
        }
        return filtered;
    }
}
```

실행 결과

```
evenNumbers = [2, 4, 6, 8, 10]
oddNumbers = [1, 3, 5, 7, 9]
```

여기서는 `filterEvenNumber()`, `filterOddNumber()` 두 메서드를 각각 만들어서 문제를 해결했다.

문제

- `FilterMainV1` 클래스를 복사해서 `FilterMainV2` 클래스를 만들자.
- 앞서 작성한 `filterEvenNumber()`, `filterOddNumber()` 두 메서드 대신에 `filter()` 라는 하나의 메서드만 사용해서 중복을 제거하자.

- 람다를 활용하자.

힌트

`filterEvenNumber()`, `filterOddNumber()`에서 다른 부분은 다음 코드이다.

```
boolean testResult = number % 2 == 0; // filterEvenNumber()
boolean testResult = number % 2 == 1; // filterOddNumber()
```

- 이 코드 조각을 `Predicate`로 전달해서 처리하면 된다.
- 변수가 `Integer number` 하나이고, 반환 타입이 `boolean`이다.
  - `Function<Integer, Boolean>` 또는 `Predicate<Integer>`를 선택하면 된다.
  - 조건이 맞는지 확인(`test`)하는 용도이므로 `Predicate`가 더 적절하다.

필터2

```
package lambda.lambda5.filter;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class FilterMainV2 {

    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // 짝수만 거르기
        Predicate<Integer> evenPredicate = n -> n % 2 == 0;
        List<Integer> evenNumbers = filter(numbers, evenPredicate);
        System.out.println("evenNumbers = " + evenNumbers);

        // 홀수만 거르기
        Predicate<Integer> oddPredicate = n -> n % 2 == 1;
        List<Integer> oddNumbers = filter(numbers, oddPredicate);
        System.out.println("oddNumbers = " + oddNumbers);
```

```
    }

    static List<Integer> filter(List<Integer> numbers, Predicate<Integer>
predicate) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer number : numbers) {
            boolean testResult = predicate.test(number);
            if (testResult) {
                filtered.add(number);
            }
        }
        return filtered;
    }

}
```

- `Predicate<Integer>`를 `filter()`에 인자로 넘긴다.
  - 짝수: `Predicate<Integer> evenPredicate = n -> n % 2 == 0`
  - 홀수: `Predicate<Integer> oddPredicate = n -> n % 2 == 1`
- `boolean testResult = predicate.test(number)`을 사용해서 넘긴 코드 조각을 `filter()` 안에서 실행한다.
  - for 루프를 통해 리스트 안에 있는 각각의 항목에 람다가 모두 적용된다.
  - 결과가 참이면 새로운 리스트에 담기고 거짓이면 새로운 리스트에 담기지 않는다.

실행 결과

```
evenNumbers = [2, 4, 6, 8, 10]
oddNumbers = [1, 3, 5, 7, 9]
```

필터3

코드를 조금 다듬어보자.

```
package lambda.lambda5.filter;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
```

```
public class FilterMainV3 {

    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // 짝수만 거르기
        List<Integer> evenNumbers = filter(numbers, n1 -> n1 % 2 == 0);
        System.out.println("evenNumbers = " + evenNumbers);

        // 홀수만 거르기
        List<Integer> oddNumbers = filter(numbers, n -> n % 2 == 1);
        System.out.println("oddNumbers = " + oddNumbers);
    }

    static List<Integer> filter(List<Integer> numbers, Predicate<Integer>
predicate) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer number : numbers) {
            if (predicate.test(number)) {
                filtered.add(number);
            }
        }
        return filtered;
    }

}
```

- `evenPredicate`, `oddPredicate`, `testResult` 변수를 제거했다.
- 해당 변수들은 학습의 이해를 돕기 위해 만든 것으로 꼭 필요한 변수는 아니다.
- 특히 람다의 경우 주로 간단한 식을 사용하므로, 복잡할 때를 제외하고는 변수를 잘 만들지 않는다.
  - `evenPredicate`, `oddPredicate`를 제거하니 가독성이 더 좋아진 것을 확인할 수 있다.

필터 만들기2

앞서 만든 `filter()` 메서드는 매개변수가 `List<Integer> numbers, Predicate<Integer> predicate`이다.

따라서 숫자 리스트에 있는 값을 필터링 하는 모든 곳에서 사용할 수 있다.

다양한 곳에서 활용할 수 있으므로, 별도의 유틸리티 클래스로 만들어보자.

필터4

```
package lambda.lambda5.filter;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class IntegerFilter {

    public static List<Integer> filter(List<Integer> list, Predicate<Integer>
predicate) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer num : list) {
            if (predicate.test(num)) {
                filtered.add(num);
            }
        }
        return filtered;
    }
}
```

```
package lambda.lambda5.filter;

import java.util.List;

public class FilterMainV4 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1,2,3,4,5,6,7,8,9,10);

        // 짝수만 거르는 필터
        List<Integer> evenNumbers = IntegerFilter.filter(numbers, n -> n % 2
== 0);
        System.out.println("evenNumbers = " + evenNumbers);

        // 홀수만 거르는 필터
        List<Integer> oddNumbers = IntegerFilter.filter(numbers, n -> n % 2 ==
1);
```

```
        System.out.println("oddNumbers = " + oddNumbers);
    }
}
```

실행 결과

```
evenNumbers = [2, 4, 6, 8, 10]
oddNumbers = [1, 3, 5, 7, 9]
```

- 범용성 있게 다양한 곳에서 사용할 수 있는 `IntegerFilter` 클래스를 만들었다.
- 하지만 `Integer` 숫자에만 사용할 수 있는 한계가 있다.

필터5 - 제네릭 도입

제네릭을 사용하면 클래스 코드의 변경 없이 다양한 타입을 적용할 수 있다.

앞서 만든 `IntegerFilter`에 제네릭을 도입해보자.

```
package lambda.lambda5.filter;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class GenericFilter {
    public static <T> List<T> filter(List<T> list, Predicate<T> predicate) {
        List<T> result = new ArrayList<>();
        for (T num : list) {
            if (predicate.test(num)) {
                result.add(num);
            }
        }
        return result;
    }
}
```

- 제네릭 `<T>`를 선언하고, `Integer`로 되어 있는 부분을 `T`로 변경하면 된다.

```
package lambda.lambda5.filter;
```

```
import java.util.List;

public class FilterMainV5 {
    public static void main(String[] args) {
        // 숫자 사용 필터
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> numbersResult = GenericFilter.filter(numbers, n -> n % 2
== 0);
        System.out.println("numbersResult = " + numbersResult); // [2, 4, 6,
8, 10]

        // 문자 사용 필터
        List<String> strings = List.of("A", "BB", "CCC");
        List<String> stringsResult = GenericFilter.filter(strings, s ->
s.length() >= 2);
        System.out.println("stringsResult = " + stringsResult); // [BB, CCC]
    }
}
```

실행 결과

```
numbersResult = [2, 4, 6, 8, 10]
stringsResult = [BB, CCC]
```

- 제네릭을 도입한 덕분에 `Integer`, `String` 같은 다양한 타입의 리스트에 필터링 기능을 사용할 수 있게 되었다.
- `GenericFilter`는 제네릭을 사용할 수 있는 모든 타입의 리스트를 람다 조건으로 필터링 할 수 있다. 따라서 매우 유연한 필터링 기능을 제공한다.

맵 만들기1

맵(map)은 대응, 변환을 의미하는 매핑(mapping)의 줄임말이다.

매핑은 어떤 것을 다른 것으로 변환하는 과정을 의미한다.

프로그래밍에서는 각 요소를 다른 값으로 변환하는 작업을 매핑(mapping, map)이라 한다.

쉽게 이야기해서 어떤 하나의 데이터를 다른 데이터로 변환하는 작업이라고 생각하면 된다.

리스트에 있는 특정 값을 다른 값으로 매핑(변환)해보자.

맵1

```
package lambda.lambda5.map;

import java.util.ArrayList;
import java.util.List;

public class MapMainV1 {

    public static void main(String[] args) {
        List<String> list = List.of("1", "12", "123", "1234");

        // 문자열을 숫자로 변환
        List<Integer> numbers = mapStringToInteger(list);
        System.out.println("numbers = " + numbers);

        // 문자열의 길이
        List<Integer> lengths = mapStringToLength(list);
        System.out.println("lengths = " + lengths);
    }

    static List<Integer> mapStringToInteger(List<String> list) {
        List<Integer> numbers = new ArrayList<>();
        for (String s : list) {
            Integer value = Integer.valueOf(s);
            numbers.add(value);
        }
        return numbers;
    }

    static List<Integer> mapStringToLength(List<String> list) {
        List<Integer> numbers = new ArrayList<>();
        for (String s : list) {
            Integer value = s.length();
            numbers.add(value);
        }
        return numbers;
    }
}
```

실행 결과

```
numbers = [1, 12, 123, 1234]
lengths = [1, 2, 3, 4]
```

- **문자열을 숫자로 변환**
  - 리스트에 있는 문자(`String`) "1", "12", "123", "1234"를 숫자(`Integer`) 1, 12, 123, 1234로 변환했다.
- **문자열을 문자열의 길이로 변환**
  - 리스트에 있는 문자(`String`) "1", "12", "123", "1234"를 각 문자의 길이(`Integer`)인 1, 2, 3, 4로 변환했다.

여기서는 `mapStringToInteger()`, `mapStringToLength()` 두 메서드를 각각 만들어서 문제를 해결했다.

문제

- `MapMainV1` 클래스를 복사해서 `MapMainV2` 클래스를 만들자.
- 앞서 작성한 `mapStringToInteger()`, `mapStringToLength()` 두 메서드 대신에 `map()`이라는 하나의 메서드만 사용해서 중복을 제거하자.
- 람다를 활용하자.

힌트

`mapStringToInteger()`, `mapStringToLength()`에서 다른 부분은 다음 코드이다.

```
Integer value = Integer.valueOf(s); // mapStringToInteger()
Integer value = s.length(); // mapStringToLength()
```

- 이 코드 조각을 람다로 전달해서 처리하면 된다.
- 변수가 `String s` 하나이고, 반환 타입이 `Integer`이다.
  - `Function<String, Integer>`를 선택하면 된다.

맵2

```java
package lambda.lambda5.map;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class MapMainV2 {

    public static void main(String[] args) {
        List<String> list = List.of("1", "12", "123", "1234");

        // 문자열을 숫자로 변환
        Function<String, Integer> toNumber = s -> Integer.valueOf(s);
        List<Integer> numbers = map(list, toNumber);
        System.out.println("numbers = " + numbers);

        // 문자열의 길이
        Function<String, Integer> toLength = s -> s.length();
        List<Integer> lengths = map(list, toLength);
        System.out.println("lengths = " + lengths);
    }

    static List<Integer> map(List<String> list, Function<String, Integer>
mapper) {
        List<Integer> numbers = new ArrayList<>();
        for (String s : list) {
            Integer value = mapper.apply(s);
            numbers.add(value);
        }
        return numbers;
    }

}
```

- `Function<String, Integer>`를 `map()`에 인자로 넘긴다.
  - 문자열을 숫자로: `Function<String, Integer> toNumber = s -> Integer.valueOf(s)`
  - 문자열을 길이로: `Function<String, Integer> toLength = s -> s.length()`
- `Integer value = mapper.apply(s)`을 사용해서 넘긴 코드 조각을 `map()` 안에서 실행한다.
  - for 루프를 통해 리스트 안에 있는 각각의 항목에 람다가 모두 적용된다.
  - 여기서 각각의 `String` 항목이 람다를 통해 `Integer`로 변환된다.

실행 결과

```
numbers = [1, 12, 123, 1234]
lengths = [1, 2, 3, 4]
```

맵3

코드를 정리해보자.

```
package lambda.lambda5.map;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class MapMainV3 {

    public static void main(String[] args) {
        List<String> list = List.of("1", "12", "123", "1234");

        // 문자열을 숫자로 변환
        List<Integer> numbers = map(list, s -> Integer.valueOf(s));
        System.out.println("numbers = " + numbers);

        // 문자열의 길이
        List<Integer> lengths = map(list, s -> s.length());
        System.out.println("lengths = " + lengths);
    }

    static List<Integer> map(List<String> list, Function<String, Integer>
mapper) {
        List<Integer> numbers = new ArrayList<>();
        for (String s : list) {
            numbers.add(mapper.apply(s));
        }
        return numbers;
    }
```

```
}
```

- `toNumber`, `toLength`, `value` 변수를 제거했다.

맵 만들기2

앞서 만든 `map()` 메서드는 매개변수가 `List<String> list, Function<String, Integer> mapper`이다.

따라서 문자열 리스트를 숫자 리스트로 변환(매핑)할 때 사용할 수 있다.

다양한 곳에서 활용할 수 있으므로, 별도의 유틸리티 클래스로 만들어보자.

맵4

```
package lambda.lambda5.map;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class StringToIntegerMapper {
    public static List<Integer> map(List<String> list, Function<String,
Integer> mapper) {
        List<Integer> result = new ArrayList<>();
        for (String s : list) {
            result.add(mapper.apply(s));
        }
        return result;
    }
}
```

```
package lambda.lambda5.map;

import java.util.List;

public class MapMainV4 {
```

```
    public static void main(String[] args) {
        List<String> list = List.of("1", "12", "123", "1234");

        // 문자열을 숫자로 변환
        List<Integer> numbers = StringToIntegerMapper.map(list, s ->
Integer.valueOf(s));
        System.out.println("numbers = " + numbers);

        // 문자열의 길이
        List<Integer> lengths = StringToIntegerMapper.map(list, s ->
s.length());
        System.out.println("lengths = " + lengths);
    }

}
```

실행 결과

```
numbers = [1, 12, 123, 1234]
lengths = [1, 2, 3, 4]
```

- 범용성 있게 다양한 곳에서 사용할 수 있는 `StringToIntegerMapper` 클래스를 만들었다.
- 하지만 `String` 리스트를 `Integer` 리스트로 변환할 때만 사용할 수 있는 한계가 있다.

맵5 - 제네릭 도입

제네릭을 사용하면 클래스 코드의 변경 없이 다양한 타입을 적용할 수 있다.

앞서 만든 `StringToIntegerMapper`에 제네릭을 도입해보자.

```
package lambda.lambda5.map;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class GenericMapper {
    public static <T, R> List<R> map(List<T> list, Function<T, R> mapper) {
        List<R> result = new ArrayList<>();
```

```
        for (T s : list) {
            result.add(mapper.apply(s));
        }
        return result;
    }
}
```

- 제네릭 `<T, R>`을 선언하고, `String`으로 되어 있는 부분을 `T`로, `Integer`로 되어 있는 부분을 `R`로 변경하면 된다.
  - `T`: 입력, `R`: 출력(반환)

```
package lambda.lambda5.map;

import java.util.List;

public class MapMainV5 {

    public static void main(String[] args) {
        List<String> fruits = List.of("apple", "banana", "orange");
        // String -> String
        List<String> upperFruits = GenericMapper.map(fruits, s ->
s.toUpperCase());
        System.out.println(upperFruits); // [APPLE, BANANA, ORANGE]

        // String -> Integer
        List<Integer> lengthFruits = GenericMapper.map(fruits, s ->
s.length());
        System.out.println(lengthFruits); // [5, 6, 6]

        // Integer -> String
        List<Integer> integers = List.of(1, 2, 3);
        List<String> starList = GenericMapper.map(integers, n ->
"*".repeat(n));
        System.out.println(starList); // [*, **, ***]
    }

}
```

- `String.repeat(int count)`는 자바 11에 추가된 메서드이다. 같은 문자를 `count` 수 만큼 붙여서 반환한다.

실행 결과

```
[APPLE, BANANA, ORANGE]
[5, 6, 6]
[*, **, ***]
```

- 제네릭을 도입한 덕분에 다양한 타입의 리스트의 값을 변환(매핑) 사용할 수 있게 되었다.
- `GenericMapper`는 제네릭을 사용할 수 있는 모든 타입의 리스트를 람다 조건으로 변환(매핑) 할 수 있다. 따라서 매우 유연한 매핑(변환) 기능을 제공한다.

필터와 맵 활용1

이번에는 앞서 만든 필터와 맵을 함께 활용해서 문제를 풀어보자.

필터와 맵 활용 - 문제1

- 리스트에 있는 값 중에 짝수만 남기고, 남은 짝수 값의 2배를 반환해라.
- `direct()`에 람다, 앞서 작성한 유틸리티를 사용하지 말고, `for`, `if`등으로 코드를 직접 작성해라.
- `lambda()`에 앞서 작성한 필터와 맵 유틸리티를 사용해서 코드를 작성해라.

다음 예시를 참고하자.

```
package lambda.lambda5.mystream;

import lambda.lambda5.filter.GenericFilter;
import lambda.lambda5.map.GenericMapper;

import java.util.ArrayList;
import java.util.List;

public class Ex1_Number {

    public static void main(String[] args) {
        // 짝수만 남기고, 남은 값의 2배를 반환
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        List<Integer> directResult = direct(numbers);
```

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

        List<Integer> lambdaResult = lambda(numbers);
        System.out.println("lambdaResult = " + lambdaResult);
    }

    static List<Integer> direct(List<Integer> numbers) {
        // TODO 코드 작성
    }

    static List<Integer> lambda(List<Integer> numbers) {
        // TODO 코드 작성
    }
}
```

실행 결과

```
directResult = [4, 8, 12, 16, 20]
lambdaResult = [4, 8, 12, 16, 20]
```

정답

```
package lambda.lambda5.mystream;

import lambda.lambda5.filter.GenericFilter;
import lambda.lambda5.map.GenericMapper;

import java.util.ArrayList;
import java.util.List;

public class Ex1_Number {

    public static void main(String[] args) {
        // 짝수만 남기고, 남은 값의 2배를 반환
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        List<Integer> directResult = direct(numbers);
        System.out.println("directResult = " + directResult);
```

```
        List<Integer> lambdaResult = lambda(numbers);
        System.out.println("lambdaResult = " + lambdaResult);
    }

    static List<Integer> direct(List<Integer> numbers) {
        List<Integer> result = new ArrayList<>();

        for (Integer number : numbers) {
            if (number % 2 == 0) {  // 짝수 필터링
                int numberX2 = number * 2;
                result.add(numberX2);  // 2배로 변환하여 추가
            }
        }
        return result;
    }

    static List<Integer> lambda(List<Integer> numbers) {
        List<Integer> filteredList = GenericFilter.filter(numbers, n -> n % 2
== 0);
        List<Integer> mappedList = GenericMapper.map(filteredList, n -> n *
2);
        return mappedList;
    }
}
```

실행 결과

```
directResult = [4, 8, 12, 16, 20]
lambdaResult = [4, 8, 12, 16, 20]
```

`direct()`와 `lambda()`는 서로 다른 프로그래밍 스타일을 보여준다.

`direct()`는 프로그램을 **어떻게** 수행해야 하는지 수행 절차를 명시한다.

- 쉽게 이야기해서 개발자가 로직 하나하나를 **어떻게** 실행해야 하는지 명시한다.
- 이런 프로그래밍 방식을 **명령형 프로그래밍**이라 한다.
- 명령형 스타일은 익숙하고 직관적이나, 로직이 복잡해질수록 반복 코드가 많아질 수 있다.

`lambda()`는 **무엇을** 수행해야 하는지 원하는 결과에 초점을 맞춘다.

- 쉽게 이야기해서 특정 조건으로 필터하고, 변환하라고 선언하면, 구체적인 부분은 내부에서 수행된다.
- 개발자는 필터하고 변환하는 것 즉 무엇을 해야 하는가에 초점을 맞춘다.
  - 예를 들어 실제 어떻게 for문과 if문 등을 사용해서 필터하고 변환할지를 개발자가 크게 신경쓰지 않는다.
- 이런 프로그래밍 방식을 **선언적 프로그래밍**이라 한다.
- 선언형 스타일은 **무엇**을 하고자 하는지가 명확히 드러난다. 따라서 코드 가독성과 유지보수가 쉬워진다.
  - 여기서는 필터하고 변환하는 것에만 초점을 맞춘다. 실제 어떻게 필터하고 변환할지는 해당 기능을 사용하는 입장에서는 신경쓰지 않는다.

명령형 vs 선언적 프로그래밍

명령형 프로그래밍 (Imperative Programming)

- **정의:** 프로그램이 **어떻게(How)** 수행되어야 하는지, 즉 **수행 절차**를 명시하는 방식이다.
- **특징:**
  - **단계별 실행:** 프로그램의 각 단계를 명확하게 지정하고 순서대로 실행한다.
  - **상태 변화:** 프로그램의 상태(변수 값 등)가 각 단계별로 어떻게 변화하는지 명시한다.
  - **낮은 추상화:** 내부 구현을 직접 제어해야 하므로 추상화 수준이 낮다.
  - **예시:** 전통적인 for 루프, while 루프 등을 명시적으로 사용하는 방식
  - **장점**: 시스템의 상태와 흐름을 세밀하게 제어할 수 있다.

선언적 프로그래밍 (Declarative Programming)

- **정의:** 프로그램이 **무엇(What)**을 수행해야 하는지, 즉 **원하는 결과**를 명시하는 방식이다.
- **특징:**
  - **문제 해결에 집중:** 어떻게(how) 문제를 해결할지보다 **무엇**을 원하는지에 초점을 맞춘다.
  - **코드 간결성:** 간결하고 읽기 쉬운 코드를 작성할 수 있다.
  - **높은 추상화:** 내부 구현을 숨기고 원하는 결과에 집중할 수 있도록 추상화 수준을 높인다.
  - **예시:** `filter`, `map` 등 람다의 고차 함수를 활용, HTML, SQL 등
- **장점**: 코드가 간결하고, 의도가 명확하며, 유지보수가 쉬운 경우가 많다.

정리

- **명령형 프로그래밍**은 프로그램이 수행해야 할 각 단계와 처리 과정을 상세하게 기술하여, **어떻게** 결과에 도달할지를 명시한다.
- **선언적 프로그래밍**은 원하는 결과나 상태를 기술하며, 그 결과를 얻기 위한 내부 처리 방식은 추상화되어 있어 개발자가 **무엇을** 원하는지에 집중할 수 있게 한다.
- 특히, **람다**와 같은 도구를 사용하면, 코드를 간결하게 작성하여 선언적 스타일로 문제를 해결할 수 있다.

필터와 맵 활용2

이번 문제는 풀기전에 다음 학생 클래스를 먼저 만들자.

```
package lambda.lambda5.mystream;

public class Student {

    private String name;
    private int score;

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

    public String getName() {
        return name;
    }

    public int getScore() {
        return score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", score=" + score +
                '}';
    }
}
```

필터와 맵 활용 - 문제2

앞서 만든 필터와 맵을 함께 활용해서 문제를 풀어보자.

- 점수가 80점 이상인 학생의 이름을 추출해라.
- `direct()`에 람다를 사용하지 않고 for, if 등의 코드를 직접 작성해라.
- `lambda()`에 앞서 작성한 필터와 맵을 사용해서 코드를 작성해라.

다음 예시를 참고하자.

```
package lambda.lambda5.mystream;

import lambda.lambda5.filter.GenericFilter;
import lambda.lambda5.map.GenericMapper;

import java.util.ArrayList;
import java.util.List;

public class Ex2_Student {

    public static void main(String[] args) {
        // 점수가 80점 이상인 학생의 이름을 추출해라.
        List<Student> students = List.of(
                new Student("Apple", 100),
                new Student("Banana", 80),
                new Student("Berry", 50),
                new Student("Tomato", 40)
        );

        List<String> directResult = direct(students);
        System.out.println("directResult = " + directResult);

        List<String> lambdaResult = lambda(students);
        System.out.println("lambdaResult = " + lambdaResult);
    }

    private static List<String> direct(List<Student> students) {
    }

    private static List<String> lambda(List<Student> students) {
    }
}
```

실행 결과

```
directResult = [Apple, Banana]
lambdaResult = [Apple, Banana]
```

정답

```
package lambda.lambda5.mystream;

import lambda.lambda5.filter.GenericFilter;
import lambda.lambda5.map.GenericMapper;

import java.util.ArrayList;
import java.util.List;

public class Ex2_Student {

    public static void main(String[] args) {
        // 점수가 80점 이상인 학생의 이름을 추출해라.
        List<Student> students = List.of(
                new Student("Apple", 100),
                new Student("Banana", 80),
                new Student("Berry", 50),
                new Student("Tomato", 40)
        );

        List<String> directResult = direct(students);
        System.out.println("directResult = " + directResult);

        List<String> lambdaResult = lambda(students);
        System.out.println("lambdaResult = " + lambdaResult);
    }

    private static List<String> direct(List<Student> students) {
        List<String> highScoreNames = new ArrayList<>();
        for (Student student : students) {
            if (student.getScore() >= 80) {
                String name = student.getName();
                highScoreNames.add(name);
            }
        }
        return highScoreNames;
    }

    // 1. filter: 점수가 80점 이상인 학생들만 필터
```

```
    // 2. map: 필터링된 학생 객체에서 이름만 추출
    private static List<String> lambda(List<Student> students) {
        List<Student> filtered = GenericFilter.filter(students, s ->
s.getScore() >= 80);
        List<String> mapped = GenericMapper.map(filtered, s -> s.getName());
        return mapped;
    }
}
```

실행 결과

```
directResult = [Apple, Banana]
lambdaResult = [Apple, Banana]
```

lambda()를 분석해보자.

처음 데이터는 다음과 같다.

```
Student["Apple", 100]
Student["Banana", 80]
Student["Berry", 50]
Student["Tomato", 40]
```

필터 조건

80점 이상의 학생만 필터한다.

`(Student s) -> s.getScore() >= 80`

```
Student["Apple", 100] -> O
Student["Banana", 80] -> O
Student["Berry", 50]  -> X
Student["Tomato", 40] -> X
```

매핑 조건

학생(`Student`)을 학생의 이름(`String name`)으로 매핑한다.

`(Student s) -> s.getName()`

```
Student["Apple", 100] -> "Apple"
Student["Banana", 80] -> "Banana"
```

최종 결과

```
"Apple"
"Banana"
```

- 앞서 만든 필터와 맵 유틸리티와 람다 덕분에 매우 편리하게 리스트를 필터링 하고 변환(매핑)할 수 있었다.

`direct()`는 **어떻게** 수행해야 하는지 수행 절차를 명시한다.

- 코드를 보면 구체적으로 **어떻게** 필터링 하고 이름을 추출하는지, `for`, `if` 등을 통해 수행 절차를 구체적으로 지시한다.

`lambda()` 코드는 선언적이다.

- 요구사항인 "점수가 80점 이상인 학생의 이름을 추출해라"를 다음과 같이 선언적으로 해결했다.
  - 점수가 80점 이상인 학생을 필터링 해라
    - `GenericFilter.filter(students, s -> s.getScore() >= 80)`
  - 학생의 이름을 추출해라.
    - `GenericMapper.map(filtered, s -> s.getName())`
  - 이 코드를 보면 구체적으로 **어떻게** 필터링 하고 이름을 추출하는지 보다는 요구사항에 맞추어 **무엇**을 하고 싶은지에 초점을 맞춘다.

람다를 사용한 덕분에, 코드를 간결하게 작성하고, 선언적 스타일로 문제를 해결할 수 있었다.

스트림 만들기1

지금까지는 필터와 맵 기능을 별도의 유틸리티에서 각각 따로 제공했다.

```
List<Student> filtered = GenericFilter.filter(students, s -> s.getScore() >=
80);
List<String> mapped = GenericMapper.map(filtered, s -> s.getName());
```

그래서 두 기능을 함께 사용할 때, 필터된 결과를 다시 맵에 전달하는 번거로운 과정을 거쳐야했다.

이번에는 앞서 만든 필터와 맵을 함께 편리하게 사용할 수 있도록 하나의 객체에 기능을 통합해보자.

스트림1

필터와 맵을 사용할 때를 떠올려보면 데이터들이 흘러가면서 필터되고, 매핑된다. 그래서 마치 데이터가 물 흐르듯이 흘러간다는 느낌을 받았을 것이다. 참고로 흐르는 좁은 시냇물을 영어로 스트림이라 한다.
(IO에서 배운 스트림도 같은 뜻이다. 데이터가 흘러간다는 뜻이다. 하지만 여기서 설명하는 스트림이 IO 스트림은 아니다.)
이렇듯 데이터가 흘러가면서 필터도 되고, 매핑도 되는 클래스의 이름을 스트림(`Stream`)이라고 짓자.

```
package lambda.lambda5.mystream;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

public class MyStreamV1 {

    private List<Integer> internalList;

    public MyStreamV1(List<Integer> internalList) {
        this.internalList = internalList;
    }

    public MyStreamV1 filter(Predicate<Integer> predicate) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer element : internalList) {
            if (predicate.test(element)) {
                filtered.add(element);
            }
        }
        return new MyStreamV1(filtered);
    }

    public MyStreamV1 map(Function<Integer, Integer> mapper) {
        List<Integer> mapped = new ArrayList<>();
        for (Integer element : internalList) {
            mapped.add(mapper.apply(element));
        }
        return new MyStreamV1(mapped);
```

```
    }

    public List<Integer> toList() {
        return internalList;
    }

}
```

- 예제에서 스트림은 자신의 데이터 리스트를 가진다. 여기서는 쉽게 설명하기 위해 `Integer`를 사용했다.
- 스트림은 자신의 데이터를 필터(`filter`)하거나 매핑(`map`)해서 새로운 스트림을 만들 수 있다.
  - 예를 들어서 필터를 하면 필터된 데이터를 기반으로 새로운 스트림이 만들어진다.
  - 예를 들어서 매핑을 하면 매핑된 데이터를 기반으로 새로운 스트림이 만들어진다.
- 스트림은 내부의 데이터 리스트를 `toList()`로 반환할 수 있다.
- `filter()`, `map()`에 앞서 개발한 `GenericFilter`, `GenericMapper`의 기능을 사용해도 되지만, 여기서는 직접 작성하겠다.

이렇게 만든 스트림을 활용해보자.

먼저 짝수만 남기고 남은 값의 2배를 반환하는 기능을 개발해보자.

- **필터**: 짝수만 남긴다.
- **매핑**: 값을 2배 곱한다.

```
package lambda.lambda5.mystream;

import java.util.List;

public class MyStreamV1Main {

    public static void main(String[] args) {
        // 짝수만 남기고, 남은 값의 2배를 반환
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        returnValue(numbers);
    }

    static void returnValue(List<Integer> numbers) {
        MyStreamV1 stream = new MyStreamV1(numbers);
        MyStreamV1 filteredStream = stream.filter(n -> n % 2 == 0);
        MyStreamV1 mappedStream = filteredStream.map(n -> n * 2);
        List<Integer> result = mappedStream.toList();
        System.out.println("result1 = " + result);
    }
```

```
}
```

실행 결과

```
result1 = [4, 8, 12, 16, 20]
```

`returnValue()`를 통해 어떻게 실행되는지 순서대로 알아보자.

데이터 변환 과정

```
// 1. stream[1,2,3,4,5,6,7,8,9,10]
MyStreamV1 stream = new MyStreamV1(numbers);

// 2. stream[1,2,3,4,5,6,7,8,9,10] -> filteredStream[2,4,6,8,10]
MyStreamV1 filteredStream = stream.filter(n -> n % 2 == 0);

// 3. filteredStream[2,4,6,8,10] -> mappedStream[4,8,12,16,20]
MyStreamV1 mappedStream = filteredStream.map(n -> n * 2);

// 4. mappedStream[4,8,12,16,20] -> [4,8,12,16,20]
List<Integer> result = mappedStream.toList();
```

1. 스트림 생성

```
// 1. stream[1,2,3,4,5,6,7,8,9,10]
MyStreamV1 stream = new MyStreamV1(numbers);
```

- `new MyStreamV1(numbers)`으로 스트림 객체를 생성한다. 이때 내부(`internalList`)에는 `1~10`의 데이터가 들어있다.

2. 필터 적용

```
// 2. stream[1,2,3,4,5,6,7,8,9,10] -> filteredStream[2,4,6,8,10]
MyStreamV1 filteredStream = stream.filter(n -> n % 2 == 0);
```

- `stream.filter(n -> n % 2 == 0)` 필터를 적용한다.
- 그 결과로 새로운 `MyStreamV1`이 만들어지고 그 내부에는 필터된 결과인 `[2,4,6,8,10]`이 존재한다.

3. 맵 적용

```
// 3. filteredStream[2,4,6,8,10] -> mappedStream[4,8,12,16,20]
MyStreamV1 mappedStream = filteredStream.map(n -> n * 2);
```

- `filteredStream.map(n -> n * 2)` 맵을 적용한다.
- 그 결과로 새로운 `MyStreamV1`이 만들어지고, 그 내부에는 매핑된 결과인 `[4,8,12,16,20]`이 존재한다.

4. 리스트로 변환

```
// 4. mappedStream[4,8,12,16,20] -> [4,8,12,16,20]
List<Integer> result = mappedStream.toList();
```

- `mappedStream.toList()`를 호출해서 스트림에 있는 리스트를 반환한다.
- 그 결과로 스트림 내부에 있는 리스트를 받을 수 있다. 결과는 `[4,8,12,16,20]`이다.

메서드 체인

`returnValue()` 코드를 보자.

```
MyStreamV1 stream = new MyStreamV1(numbers);
MyStreamV1 filteredStream = stream.filter(n -> n % 2 == 0);
MyStreamV1 mappedStream = filteredStream.map(n -> n * 2);
List<Integer> result = mappedStream.toList();
```

- 스트림 객체를 통해 필터와 맵을 편리하게 사용할 수 있는 것은 맞지만, 이전에 `GenericFilter`, `GenericMapper`와 비교해서 크게 편리해진 것 같지는 않다.

이전에 사용한 코드

```
List<Student> filtered = GenericFilter.filter(students, s -> s.getScore() >=
80);
List<String> mapped = GenericMapper.map(filtered, s -> s.getName());
```

우리가 만든 `MyStreamV1`은 `filter`, `map`을 호출할 때 자기 자신의 타입을 반환한다. 따라서 자기 자신의 메서드

를 연결해서 호출할 수 있다.

methodChain()을 추가하자.

```
package lambda.lambda5.mystream;

import java.util.List;

public class MyStreamV1Main {

    public static void main(String[] args) {
        // 짝수만 남기고, 남은 값의 2배를 반환
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        returnValue(numbers);
        methodChain(numbers); // 추가
    }

    private static void returnValue(List<Integer> numbers) {
        MyStreamV1 stream = new MyStreamV1(numbers);
        MyStreamV1 filteredStream = stream.filter(n -> n % 2 == 0);
        MyStreamV1 mappedStream = filteredStream.map(n -> n * 2);
        List<Integer> result = mappedStream.toList();
        System.out.println("result1 = " + result);
    }

    // 추가
    static void methodChain(List<Integer> numbers) {
        List<Integer> result = new MyStreamV1(numbers)
                .filter(n -> n % 2 == 0)
                .map(n -> n * 2)
                .toList();

        System.out.println("result2 = " + result);
    }
}
```

메서드 체인 방식을 사용한 코드

```
List<Integer> result = new MyStreamV1(numbers)
        .filter(n -> n % 2 == 0)
        .map(n -> n * 2)
```

```
        .toList();
```

자기 자신의 타입을 반환한 덕분에 메서드를 연결하는 메서드 체인 방식을 사용할 수 있다.

덕분에 지저분한 변수들을 제거하고, 깔끔하게 필터와 맵을 사용할 수 있게 되었다.

참고로 `methodChain()`의 작동 방식은 `returnValue()`와 완전히 동일하다. 단지 중간 변수들이 없을 뿐이다.

> **참고**
>
> 메서드 체인 방식이 잘 이해가 되지 않으면 **자바 중급1편 - String 클래스 - "메서드 체인닝 - Method Chaining"**을 복습하자.

스트림 만들기2

앞서 만든 스트림을 조금 더 다듬어보자.

여기서는 정적 팩토리(static factory) 메서드를 추가하자.

스트림2

```
package lambda.lambda5.mystream;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

// static factory 추가
public class MyStreamV2 {

    private List<Integer> internalList;

    private MyStreamV2(List<Integer> internalList) {
        this.internalList = internalList;
    }

    // static factory
    public static MyStreamV2 of(List<Integer> internalList) {
        return new MyStreamV2(internalList);
    }
```

```
    public MyStreamV2 filter(Predicate<Integer> predicate) {
        List<Integer> filtered = new ArrayList<>();
        for (Integer element : internalList) {
            if (predicate.test(element)) {
                filtered.add(element);
            }
        }
        return MyStreamV2.of(filtered);
    }

    public MyStreamV2 map(Function<Integer, Integer> mapper) {
        List<Integer> mapped = new ArrayList<>();
        for (Integer element : internalList) {
            mapped.add(mapper.apply(element));
        }
        return MyStreamV2.of(mapped);
    }

    public List<Integer> toList() {
        return internalList;
    }

}
```

- 기존 생성자를 외부에서 사용하지 못하도록 `private`으로 설정했다.
- 이제 `MyStreamV2`를 생성하려면 `of()` 메서드를 사용해야 한다.

```
package lambda.lambda5.mystream;

import java.util.List;

public class MyStreamV2Main {

    public static void main(String[] args) {
        // 짝수만 남기고, 남은 값의 2배를 반환
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        List<Integer> result = MyStreamV2.of(numbers)
                .filter(n -> n % 2 == 0)
                .map(n -> n * 2)
                .toList();
```

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

}
```

- `MyStreamV2.of()`로 정적 팩토리 메서드를 사용하도록 변경했다.

실행 결과

```
result = [4, 8, 12, 16, 20]
```

정적 팩토리 메서드 - static factory method

정적 팩토리 메서드는 객체 생성을 담당하는 static 메서드로, 생성자(constructor) 대신 인스턴스를 생성하고 반환하는 역할을 한다. 즉, 일반적인 생성자(Constructor) 대신에 클래스의 인스턴스를 생성하고 초기화하는 로직을 캡슐화하여 제공하는 정적(static) 메서드이다.

주요 특징은 다음과 같다.

- **정적 메서드:** 클래스 레벨에서 호출되며, 인스턴스 생성 없이 접근할 수 있다.
- **객체 반환:** 내부에서 생성한 객체(또는 이미 존재하는 객체)를 반환한다.
- **생성자 대체:** 생성자와 달리 메서드 이름을 명시할 수 있어, 생성 과정의 목적이나 특징을 명확하게 표현할 수 있다.
- **유연한 구현:** 객체 생성 과정에서 캐싱, 객체 재활용, 하위 타입 객체 반환 등 다양한 로직을 적용할 수 있다.

생성자는 이름을 부여할 수 없다. 반면에 정적 팩토리 메서드는 의미있는 이름을 부여할 수 있어, 가독성이 더 좋아지는 장점이 있다.
참고로 인자들을 받아 간단하게 객체를 생성할 때는 주로 `of(...)`라는 이름을 사용한다.

예시) 회원 등급별 생성자가 다른 경우

```
// 일반 회원 가입시 이름, 나이, 등급
new Member("회원1", 20, NORMAL);

// VIP 회원 가입시 이름, 나이, 등급, 선물 주소지
new Member("회원1", 20, VIP, "선물 주소지");
```

- 예를 들어 VIP 회원의 경우 객체 생성시 선물 주소지가 추가로 포함된다고 가정하자.

- 이런 부분을 생성자만 사용해서 처리하기는 햇갈릴 수 있다.

```
// 일반 회원 가입시 인자 2개
Member.createNormal("회원1", 20)

// VIP 회원 가입시 인자 3개
Member.createVip("회원2", 20, "선물 주소지")
```

- 정적 팩토리를 사용하면 메서드 이름으로 명확하게 회원과 각 회원에 따른 인자를 구분할 수 있다.

추가로 객체를 생성하기 전에 이미 있는 객체를 찾아서 반환하는 것도 가능하다.

예) `Integer.valueOf()` : `-128 ~ 127` 범위는 내부에 가지고 있는 `Integer` 객체를 반환한다.

> **참고**: 정적 팩토리 메서드를 사용하면 생성자에 이름을 부여할 수 있기 때문에 보통 가독성이 더 좋아진다. 하지만 반대로 이야기하면 이름도 부여해야 하고, 준비해야 하는 코드도 더 많다. 객체의 생성이 단순한 경우에는 생성자를 직접 사용하는 것이 단순함의 관점에서 보면 더 나은 선택일 수 있다.

스트림 만들기3

앞서 우리가 만든 스트림은 `Integer` 숫자만 사용할 수 있었다. 이제 제네릭을 추가할 때다.

스트림3

```
package lambda.lambda5.mystream;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

// Generic 추가
public class MyStreamV3<T> {

    private List<T> internalList;
```

```
    private MyStreamV3(List<T> internalList) {
        this.internalList = internalList;
    }

    // static factory
    public static <T> MyStreamV3<T> of(List<T> internalList) {
        return new MyStreamV3<>(internalList);
    }

    public MyStreamV3<T> filter(Predicate<T> predicate) {
        List<T> filtered = new ArrayList<>();
        for (T element : internalList) {
            if (predicate.test(element)) {
                filtered.add(element);
            }
        }
        return MyStreamV3.of(filtered);
    }

    public <R> MyStreamV3<R> map(Function<T, R> mapper) {
        List<R> mapped = new ArrayList<>();
        for (T element : internalList) {
            mapped.add(mapper.apply(element));
        }
        return MyStreamV3.of(mapped);
    }

    public List<T> toList() {
        return internalList;
    }

}
```

- `MyStreamV3`은 내부에 `List<T> internalList`를 가진다. 따라서 `MyStreamV3<T>`로 선언한다.
- `map()`은 `T`를 다른 타입인 `R`로 반환한다. `R`을 사용하는 곳은 `map` 메서드 하나이므로 `map` 메서드 앞에 추가로 제네릭 `<R>`을 선언한다.

```
package lambda.lambda5.mystream;

import java.util.List;
```

```java
public class MyStreamV3Main {

    public static void main(String[] args) {
        List<Student> students = List.of(
                new Student("Apple", 100),
                new Student("Banana", 80),
                new Student("Berry", 50),
                new Student("Tomato", 40)
        );

        // 점수가 80점 이상인 학생의 이름을 추출해라.
        List<String> result1 = ex1(students);
        System.out.println("result1 = " + result1);

        // 점수가 80점 이상이면서, 이름이 5글자인 학생의 이름을 대문자로 추출해라.
        List<String> result2 = ex2(students);
        System.out.println("result2 = " + result2);
    }

    static List<String> ex1(List<Student> students) {
        return MyStreamV3.of(students)
                .filter(s -> s.getScore() >= 80)
                .map(s -> s.getName())
                .toList();
    }

    static List<String> ex2(List<Student> students) {
        return MyStreamV3.of(students)
                .filter(s -> s.getScore() >= 80)
                .filter(s -> s.getName().length() == 5)
                .map(s -> s.getName())
                .map(name -> name.toUpperCase())
                .toList();
    }
}
```

실행 결과

```
result1 = [Apple, Banana]
result2 = [APPLE]
```

- 제네릭을 도입한 덕분에 `MyStreamV3`은 `Student`를 `String`으로 변환할 수 있었다.
- `ex2()`는 필터와 맵을 연속해서 사용할 수 있다는 것을 보여주는 예다. 메서드 체인 덕분에 필요한 기능을 얼마든지 연결해서 사용할 수 있다.

스트림 만들기4

이번에는 스트림의 최종 결과까지 스트림에서 함께 처리하도록 개선해보자.

스트림의 최종 결과를 다음과 같이 하나씩 출력해야 하는 요구사항이 있다.

```
package lambda.lambda5.mystream;

import java.util.List;

public class MyStreamLoopMain {

    public static void main(String[] args) {
        List<Student> students = List.of(
                new Student("Apple", 100),
                new Student("Banana", 80),
                new Student("Berry", 50),
                new Student("Tomato", 40)
        );

        List<String> result = MyStreamV3.of(students)
                .filter(s -> s.getScore() >= 80)
                .map(s -> s.getName())
                .toList();
        // 외부 반복
        for (String s : result) {
            System.out.println("name: " + s);
        }
    }

}
```

실행 결과

```
name: Apple
name: Banana
```

이 경우 결과 리스트를 for 문을 통해 하나씩 반복하며 출력하면 된다.

그런데 생각해보면 `filter`, `map` 등도 스트림 안에서 데이터 리스트를 하나씩 처리(함수를 적용)하는 기능이다. 따라서 최종 결과를 출력하는 일도 스트림 안에서 처리할 수 있을 것 같다.

기존 `MyStreamV3`에 스트림에 `forEach()`라는 메서드를 추가하자.

```
package lambda.lambda5.mystream;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class MyStreamV3<T> {

    private List<T> internalList;

    private MyStreamV3(List<T> internalList) {
        this.internalList = internalList;
    }

    // static factory
    public static <T> MyStreamV3<T> of(List<T> internalList) {
        return new MyStreamV3<>(internalList);
    }

    public MyStreamV3<T> filter(Predicate<T> predicate) {
        List<T> filtered = new ArrayList<>();
        for (T element : internalList) {
            if (predicate.test(element)) {
                filtered.add(element);
            }
        }
        return MyStreamV3.of(filtered);
    }
```

```
    public <R> MyStreamV3<R> map(Function<T, R> mapper) {
        List<R> mapped = new ArrayList<>();
        for (T element : internalList) {
            mapped.add(mapper.apply(element));
        }
        return MyStreamV3.of(mapped);
    }

    public List<T> toList() {
        return internalList;
    }

    // 추가
    public void forEach(Consumer<T> consumer) {
        for (T element : internalList) {
            consumer.accept(element);
        }
    }

}
```

- `forEach` 메서드는 최종 데이터에 사용할 예정이다. 최종 데이터는 반환할 것 없이 요소를 하나하나 소비만 하면 되므로 `Consumer`를 사용했다.

이렇게 만든 `forEach`를 사용해보자.

```
package lambda.lambda5.mystream;

import java.util.List;

public class MyStreamLoopMain {

    public static void main(String[] args) {
        List<Student> students = List.of(
                new Student("Apple", 100),
                new Student("Banana", 80),
                new Student("Berry", 50),
                new Student("Tomato", 40)
        );

        List<String> result = MyStreamV3.of(students)
                .filter(s -> s.getScore() >= 80)
```

```
                .map(s -> s.getName())
                .toList();

        // 외부 반복
        for (String s : result) {
            System.out.println("name: " + s);
        }

        // 추가
        MyStreamV3.of(students)
                .filter(s -> s.getScore() >= 80)
                .map(s -> s.getName())
                .forEach(name -> System.out.println("name: " + name));
    }

}
```

실행 결과

```
name: Apple
name: Banana
name: Apple
name: Banana
```

forEach 사용 코드

```
MyStreamV3.of(students)
        .filter(s -> s.getScore() >= 80)
        .map(s -> s.getName())
        .forEach(s -> System.out.println("name: " + s)); // 내부 반복
```

- `forEach()`에 함수를 전달해서 각각의 데이터 리스트를 출력하도록 했다. 이것은 최종 데이터이므로 반환할 것은 없고, 각각의 데이터를 받아서 소비만 하면 된다. 따라서 `Consumer`를 사용했다.

내부 반복 vs 외부 반복

스트림을 사용하기 전에 일반적인 반복 방식은 `for` 문, `while` 문과 같은 반복문을 직접 사용해서 데이터를 순회하는 **외부 반복(External Iteration)** 방식이었다. 예를 들어 다음 코드처럼 **개발자가 직접** 각 요소를 반복하며 처리한다.

외부 반복

```
List<String> result = ...
for (String s : result) {
    System.out.println("name: " + s);
}
```

스트림에서 제공하는 `forEach()` 메서드로 데이터를 처리하는 방식은 **내부 반복(Internal Iteration)** 이라고 부른다. 외부 반복처럼 직접 반복 제어문을 작성하지 않고, 반복 처리를 스트림 내부에 위임하는 방식이다. **스트림 내부에서** 요소들을 순회하고, 우리는 처리 로직(람다)만 정의해주면 된다.

내부 반복

```
MyStreamV3.of(students)
    .filter(s -> s.getScore() >= 80)
    .map(s -> s.getName())
    .forEach(s -> System.out.println("name: " + s)); // 내부 반복
```

- 반복 제어를 스트림이 대신 수행하므로, 사용자는 반복 로직을 신경 쓸 필요가 없다.
- 코드가 훨씬 간결해지며, **선언형 프로그래밍 스타일을 적용**할 수 있다.

정리

- 내부 반복 방식은 반복의 제어를 스트림에게 위임하기 때문에 코드가 간결해진다. 즉, **개발자는 "어떤 작업"을 할지를 집중적으로 작성하고, "어떻게 순회할지"는 스트림이 담당**하도록 하여 생산성과 가독성을 높일 수 있다. 한마디로 **선언형 프로그래밍 스타일**이다.
- **외부 반복**은 개발자가 직접 반복 구조를 제어하는 반면, **내부 반복**은 반복을 내부에서 처리한다. 따라서 코드의 가독성과 유지보수성을 향상시킨다.

내부 반복 vs 외부 반복 선택

많은 경우 내부 반복을 사용할 수 있다면 내부 반복이 선언형 프로그래밍 스타일로 직관적이기 때문에 더 나은 선택이다. 다만 때때로 외부 반복을 선택하는 것이 더 나은 경우도 있다.

외부 반복을 선택하는 것이 더 나은 경우

- 단순히 한두 줄 수행만 필요한 경우
- 반복 제어에 대한 복잡하고 세밀한 조정이 필요할 경우

단순히 한두 줄 수행만 필요한 경우

예를 들어, 어떤 리스트를 순회하며 그대로 출력만 하면 되는 등의 매우 간단한 작업을 할 때는, 굳이 스트림을 사용할 필요가 없다.

```
// 외부 반복 - for 문
List<String> items = List.of("Apple", "Banana", "Berry");
for (String item : items) {
    System.out.println(item);
}
```

이 예시와 같이 반복문 자체가 한두 줄로 끝나고, 특별한 연산(필터링, 변환 등)이 없다면, 다음 코드 보다 `for` 문이 한눈에 이해하기 쉬울 수 있다.

```
List<String> items = List.of("Apple", "Banana", "Berry");
MyStreamV3.of(items)
    .forEach(s -> System.out.println("name: " + s)); // 내부 반복
```

반복 제어에 대한 복잡하고 세밀한 조정이 필요할 경우

예를 들어 반복 중에 특정 조건을 만나면 바로 반복을 멈추거나, 일부만 건너뛰고 싶다면 `break`, `continue`등을 사용하는 외부 반복이 단순하다.

정리

- 연속적인 필터링, 매핑, 집계가 필요할 때는 스트림을 사용한 내부 반복이 선언적이고 직관적이다.
- 아주 간단한 반복이거나, 중간에 `break`, `continue`가 들어가는 흐름 제어가 필요한 경우는 외부 반복이 더 간결하고 빠르게 이해될 수 있다.

정리

- **명령형(Imperative) vs 선언형(Declarative) 프로그래밍**
  - **명령형 프로그래밍**은 **어떻게(How)** 문제를 해결할지 로직 단계별로 명령(지시)을 상세히 기술한다. 주로 `for`, `if`와 같은 제어문을 사용하며, 로직이 복잡해질수록 중복 코드가 늘어날 수 있다.
  - **선언형 프로그래밍**은 **무엇(What)**을 해야 하는지에 집중한다. 예를 들어 "짝수만 필터하고, 그 값을 2배로

변환"처럼 **원하는 결과**만 기술하면, 내부의 세부 로직(어떻게 필터링하고 변환하는지)은 외부에서 신경 쓰지 않는다. 이는 코드 가독성과 유지보수성을 높일 수 있다.

- **Filter와 Map**
  - 조건에 맞는 값만 **선별**하는 작업을 **필터**라고 하고, 값을 다른 값으로 **변환**하는 과정을 **매핑(Map)**이라고 한다.
  - 자바에서 제공하는 `Predicate`, `Function` 같은 표준 함수형 인터페이스를 사용하여 람다 형식으로 필터와 맵을 자유롭게 조합할 수 있다.
  - 필터와 맵을 유틸리티 메서드(`GenericFilter`, `GenericMapper`)로 분리해두면 다양한 타입(`T`)에 대해 재사용할 수 있어 코드 중복을 줄이고 선언형 프로그래밍 스타일을 쉽게 적용할 수 있다.
- **Stream (스트림)**
  - 필터와 맵을 포함한 여러 연산을 연속해서 적용하기 위해, 이를 하나의 흐름(스트림)으로 표현한 것이다.
  - 스트림을 사용하면 **메서드 체인** 방식으로 `filter()`, `map()`, `forEach()` 등을 연결해 호출할 수 있으므로, 중간 변수를 만들 필요 없이 깔끔하게 데이터를 가공할 수 있다.
  - **내부 반복(Internal Iteration)**을 지원해, 개발자가 명시적으로 `for` 루프를 작성하지 않고도 반복 처리 로직을 스트림 내부에 위임할 수 있다. 이로써 간결하고 직관적인 코드 작성이 가능하다.
- **내부 반복 vs 외부 반복**
  - **외부 반복**: 기존의 `for`, `while` 루프처럼, 개발자가 반복 제어를 직접 담당하며 명령형 스타일이다. 중간에 `break`, `continue` 등이 들어가는 로직을 구현하기 쉽다.
  - **내부 반복**: 스트림의 `forEach`처럼, 반복 제어를 스트림에 맡기고 개발자는 "어떤 작업을 할지(`Consumer`)"만 정의하면 된다. 이는 선언형 프로그래밍 스타일로써 코드가 짧고 의도가 명확하다.
- **정적 팩토리 메서드 (static factory method)**
  - 객체 생성 과정을 메서드로 캡슐화하여 가독성을 높이는 기법이다. 예) `MyStreamV3.of(list)`.
  - 생성자에 이름을 붙이지 못하는 한계를 보완하며, 객체 캐싱 또는 하위 타입 객체 반환 같은 유연한 로직을 적용할 수 있다.

필터(Filter)와 **맵(Map)**, 그리고 이를 포괄적으로 사용할 수 있는 **스트림(Stream)**을 적극적으로 활용하면 **선언형 프로그래밍** 스타일로 직관적인 코드를 작성할 수 있다.
반면에 더 세밀한 제어가 필요하다면 **명령형 프로그래밍**을 선택할 수도 있으므로, 상황에 따라 두 방식을 적절히 활용하면 된다.