람다 활용

소유자	종수 김
태그	

필터 만들기

Predicate를 이용하여, 짝수 홀수를 나누는 필터.

```
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);

        List<Integer> filter = filter(numbers, (x) -> x % 2 == 0);
        List<Integer> filter2 = filter(numbers, (x) -> x % 2 == 1);

        System.out.println(filter);
        System.out.println(filter2);

    }

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

```
    }
}
```

필터 만들기 - 2

유틸 클래스와 Generic Filter

Generic을 통해 어떤 타입이 들어오더라도 필터 기능을 하는 유틸 클래스를 제공가능.

```
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 x : list) {
            if (predicate.test(x)) {
                result.add(x);
            }
        }
        return result;
    }
}



package lambda.lambda5.filter;

import java.util.List;

public class FilterMainV4 {
    public static void main(String[] args) {
        List<Integer> genericNumbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> filter = IntegerFilter.filter(genericNumbers, (x) -> x % 2 == 0
        List<Integer> filter2 = IntegerFilter.filter(genericNumbers, (x) -> x % 2 ==

        System.out.println(filter);
```

```
        System.out.println(filter2);

        // 문자 사용 필터
        List<String> strings = List.of("A", "BB", "CCC");
        List<String> stringResult = GenericFilter.filter(strings, string -> string.leng
        System.out.println(stringResult);

    }

}
```

맵 만들기 - 1

Map은 대응, 변환을 의미하는 매핑의 줄임말.

어떤 것을 다른 것으로 변환하는 과정.

즉, 어떤 하나의 데이터를 다른 데이터로 변환하는 작업

```
package lambda.lambda5.map;

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

public class MapMainV1 {
    public static void main(String[] args) {
        List<String> list = List.of("1", "12", "123", "1234");

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

        List<Integer> integers = map(list, str -> str.length());
        System.out.println(integers);
    }

    private static List<Integer> map(List<String> list, Function<String, Integer>
```

```
        List<Integer> result = new ArrayList<>();
        for (String str : list) {
            Integer x = function.apply(str);
            result.add(x);
        }

        return result;
    }
    private static List<Integer> mapStringToLength(List<String> list) {

        List<Integer> result = new ArrayList<>();
        for (String str : list) {
            result.add(str.length());
        }

        return result;
    }

    private static List<Integer> mapStringToInger(List<String> list) {
        List<Integer> result = new ArrayList<>();
        for (String str : list) {
            Integer integer = Integer.valueOf(str);
            result.add(integer);
        }
        return result;
    }



}
```

맵 만들기 - 2

Generic을 이용하여, Util Class로 리팩토링

```
package lambda.lambda5.map;

import java.util.List;
```

```
import java.util.function.Function;

public class MapV1 {
    public static void main(String[] args) {
        List<String> list = List.of("1", "12", "123", "1234");
        List<Integer> numbers = GenericMap.map(list, str -> Integer.parseInt(str)
        List<String> strings = GenericMap.map(list, str -> "***" + str + "***");
        List<Integer> length = GenericMap.map(list, str -> str.length());
        List<Character> first = GenericMap.map(list, str -> str.charAt(0));

        System.out.println(numbers);
        System.out.println(strings);
        System.out.println(length);
        System.out.println(first);
    }
}

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> uppers = GenericMap.map(fruits, str -> str.toUpperCase());
        System.out.println(uppers);

        // String → Integer
        List<Integer> length = GenericMap.map(fruits, str -> str.length());
        System.out.println(length);

        // Integer → String
        List<Integer> integers = List.of(1, 2, 3);
        List<String> stars = GenericMap.map(integers, n -> "*".repeat(n));
        System.out.println(stars);
    }
```

```
}



package lambda.lambda5.map;

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

public class GenericMap {
    public static <T, R> List<R> map(List<T> list, Function<T, R> function) {
        List<R> result = new ArrayList<>();
        for (T t : list) {
            result.add(function.apply(t));
        }
        return result;
    }
}
```

- Generic을 도입한 덕분에 다양한 타입의 리스트의 값을 변환(매핑) 사용할 수 있게 됨.
- GenericMap은 제네릭을 사용할 수 있는 모든 타입의 리스트를 람다 조건으로 변환 할 수 있어, 매우 유연하게 사용 가능.

필터와 맵 활용

문제

1. 리스트에 있는 값 중에 짝수만 남기고, 남은 짝수 값의 2배를 반환
2. 람다 활용 없이, 직접 코드 작성
    a. direct()
    b. 재활용이 힘듦. → 코드가 분리되어있지 않기 때문에
    c. 코드가 복잡해짐
3. 람다
    a. 코드는 간단해졌으나, 재활용이 힘듦

```
package lambda.lambda5.mystream;

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

public class Ex1_Number {
    public static void main(String[] args) {
        List<Integer> list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        System.out.println(direct(list));

        List<Integer> filter = filter(list, x -> x % 2 == 0);
        System.out.println(filter);

        List<Integer> map = map(filter, x -> x * 2);
        System.out.println(map);

        System.out.println(lambda(list));
    }

    public static List<Integer> direct(List<Integer> list) {
        List<Integer> result = new ArrayList<>();
        for (Integer x : list) {
            if (x % 2 == 0) {
                result.add(x * 2);
            }
        }
        return result;
    }

    public static List<Integer> lambda(List<Integer> list) {
        List<Integer> even = even(list, x -> x % 2 == 0);
        List<Integer> multi = multi(even, x -> x * 2);
        return multi;
    }

    public static List<Integer> even(List<Integer> list, Predicate<Integer> predi
```

```
        List<Integer> result = new ArrayList<>();
        for(Integer x : list) {
            if (predicate.test(x)) {
                result.add(x);

            }
        }
        return result;
    }

    public static List<Integer> multi(List<Integer> list, Function<Integer, Integer
        List<Integer> result = new ArrayList<>();
        for (Integer x : list) {
            result.add(function.apply(x));
        }
        return result;
    }

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

    public static <T, R> List<R> map(List<T> list, Function<T, R> function) {
        List<R> result = new ArrayList<>();
        for (T x : list) {
            result.add(function.apply(x));
        }
        return result;
    }
}
```

- direct(), lambda()는 서로 다른 프로그래밍 방식
    - direct : 프로그램을 '어떻게' 수행해야 하는지 수행 절차를 명시

        - 개발자가 로직 하나하나를 어떻게 실행해야 하는지를 명시
        - 명령형 프로그래밍 방식
        - 익숙하고 직관적이지만, 로직이 복잡해질수록 반복 코드가 많아짐.
    - lambda : '무엇을' 수행해야 하는지 원하는 결과에 초점을 맞춤.
        - 특정 조건으로 필터링하고, 변환하라고 '선언'하면 구체적인 부분은 내부에서 수행.
        - 개발자는 필터하고 변환하는 것, 즉 '무엇을 해야하는가'에만 초점.
            - for를 통해 반복문을 수행하고 if를 통해 조건문을 수행하고 어쩌고는 신경 쓸바가 아님.
        - 선언적 프로그래밍 방식
        - 무엇을 하고자 하는지가 명확하며, 세부가 어떻게 되었던, 결과적으로는 수행한 결과가 나옴.
    - 내부 로직을 몰라도 되기에 선언적 프로그래밍이 조금 더 알아보기가 편함.(필터해, 매핑 해! 가 로직에 눈에 보이기 때문.)
        - 물론 그 내부에 필터, 맵 등은 명령형 프로그래밍 형식으로 되어있겠으나, 추상화 되어있는 함수를 사용하는 입장에서는 내부까지는 몰라도 되기 때문.

명령형 프로그래밍 vs 선언적 프로그래밍

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 활용 - 2

1 리팩토링

- 제네릭을 통해 타입 재활용성을 높였고, 어떤 필터링 로직, 매핑 로직이 들어오더라도 재활용 가능.

```
package lambda.lambda5.mystream;

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

```
public class Ex1_Number {
    public static void main(String[] args) {
        List<Integer> list = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        System.out.println(direct(list));

        List<Integer> filter = filter(list, x -> x % 2 == 0);
        System.out.println(filter);

        List<Integer> map = map(filter, x -> x * 2);
        System.out.println(map);

        System.out.println(lambda(list));

        List<String> strings = List.of("AA","BB","CC","ddd","eee","fff");
        List<String> stringsFilter = filter(strings, str -> str.length() > 2);
        List<String> stringsMap = map(stringsFilter, str -> str.toUpperCase());
        System.out.println(stringsMap);

    }


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

    public static <T, R> List<R> map(List<T> list, Function<T, R> function) {
        List<R> result = new ArrayList<>();
        for (T x : list) {
            result.add(function.apply(x));
        }
        return result;
    }
}
```

2. 학생 클래스를 만들고,
    a. 점수가 80점 이상인 학생의 이름
    b. direct()
    c. lambda 활용

```
package lambda.lambda5.mystream;

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

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

public class Ex2_Main {
    public static void main(String[] args) {
        // 점수가 80점 이상인 학생의 이름을 추출해라. List<Student> students = List.o
        List<Student> students = List.of(
                new Student("Apple", 100),
                new Student("Banana", 80),
                new Student("Berry", 50),
                new Student("Tomato", 40)
        );

        direct(students);

        List<String> result = map(filter(students, student -> student.getScore() >
        Runnable runnable = () -> result.forEach(System.out::println);
        lambda(runnable);

        List<String> res = lambda(students);
        res.forEach(System.out::println);

    }

    public static List<Student> filter (List<Student> students, Predicate<Studer
```

```
    List<Student> filteredStudents = new ArrayList<>();
    for (Student student : students) {
        if (predicate.test(student)) {
            filteredStudents.add(student);
        }
    }
    return filteredStudents;
}

public static List<String> map (List<Student> students, Function<Student, S
    List<String> result = new ArrayList<>();
    for(Student student : students) {
        result.add(function.apply(student));
    }
    return result;
}

public static List<String> lambda (List<Student> students) {
    List<Student> filteredStudents = GenericFilter.filter(students, student -> s
    List<String> mapStudents = GenericMap.map(filteredStudents, student ->
    return mapStudents;
}

public static void lambda(Runnable runnable) {
    runnable.run();
}

public static void direct(List<Student> students) {
    List<Student> result = new ArrayList<>();
    for (Student student : students) {
        if (student.getScore() >= 80) {
            result.add(student);
        }
    }

    for (Student student : result) {
        System.out.println(student.getName());
    }
```

```
    }
}
```

- 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 만들기 - 1

필터와 맵 기능을 별도의 유틸리티에서 사용한다면,

두 기능을 함께 사용할 때 필터된 결과를 다시 맵에 전달하는 과정을 거쳐야함.

필터와 맵을 사용할 때를 떠올려보면 데이터들이 흘러가면서 필터되고, 매핑 됨.
데이터가 흘러가면서 필터도 되고, 매핑도 되는 클래스의 이름을 스트림이라고 칭함.

```
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> filteredList = new ArrayList<>();
        for (Integer element : internalList) {
            if (predicate.test(element)) {
                filteredList.add(element);
            }
        }
        return new MyStreamV1(filteredList);
    }

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

        return new MyStreamV1(filteredList);
    }

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

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);
    }
```

```
    private static void returnValue(List<Integer> numbers) {
        MyStreamV1 stream = new MyStreamV1(numbers);
        MyStreamV1 filter = stream.filter(x -> x % 2 == 0);
        MyStreamV1 map = filter.map((n -> n * 2));
        System.out.println(map.toList());
    }
}
```

1. Stream 객체 생성
2. 필터 적용
3. 맵 적용
4. 리스트 변환

- 체이닝이 가능하지 않을까?

```
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);
        methodChain(numbers);
    }

    private static void methodChain(List<Integer> numbers) {
        List<Integer> list = new MyStreamV1(numbers)
                .filter(x -> x % 2 == 0)
                .map(x -> x * 2)
                .toList();

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

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

- 지저분한 변수를 제거하고, 가독성을 높임.

스트림 만들기 - 2

정적 팩토리 메서드 (static factory) 추가

객체 생성을 담당하는 static 메서드로, 생성자 대신 인스턴스를 생성하고 반환하는 역할.

```
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> filteredList = new ArrayList<>();
        for (Integer element : internalList) {
            if (predicate.test(element)) {
                filteredList.add(element);
            }
        }
        return new MyStreamV2(filteredList);
    }

    public MyStreamV2 map(Function<Integer, Integer> mapper) {
        List<Integer> filteredList = new ArrayList<>();
```

```
        for (Integer element : internalList) {
            filteredList.add(mapper.apply(element));
        }

        return new MyStreamV2(filteredList);
    }

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

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> list = MyStreamV2.of(numbers)
                .filter(n -> n % 2 == 0)
                .map(n -> n * 2)
                .toList();

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

- 생성자를 생성하지 못하도록 private
- V2를 생성하기 위해서는 of 메서드를 사용해야함.

주요 특징

1. 정적 메서드 : 클래스 레벨에서 호출되며, 인스턴스 생성 없이 접근 가능
2. 객체 반환 : 내부에서 생성한 객체(이미 존재하는)를 반환 가능.

3. 생성자 대체 : 생성자와 달리 메서드 이름을 명시할 수 있어, 생성 과정의 목적이나 특징을 명확하게 표현 가능
4. 유연한 구현 : 객체 생성 과정에서 캐싱, 객체 재활용, 하위 타입 객체 반환 등 다양한 로직 적용 가능.

- 생성자는 이름을 부여할 수 없음. 정적 팩토리 메서드는 의미있는 이름을 부여할 수 있음.

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 만들기 - 3

제네릭 추가

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

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

```
        return MyStreamV3.of(filteredList);
    }

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


package lambda.lambda5.mystream;

import java.util.List;

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 이름 추출
        List<String> result1 = ex1(students);
        System.out.println("result1 = " + result1);

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

    private 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(s -> s.toUpperCase())
```

```
            .toList();
    }

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

- `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.

- 제네릭을 도입한 덕분에 MyStreamV3는 Student를 String으로 변환할 수 있음.
- 또한, 스트림(흐름)이라는 이름에 걸맞게 여러 메서드를 체이닝하여 하나의 흐름처럼 사용 가능.

스트림 만들기 - 4

최종 결과까지 스트림에서 출력.

```
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;

// 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> filteredList = new ArrayList<>();
        for (T element : internalList) {
            if (predicate.test(element)) {
                filteredList.add(element);
            }
        }
        return MyStreamV3.of(filteredList);
    }

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

        return MyStreamV3.of(filteredList);
    }

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

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

```
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> list = MyStreamV3.of(students)
                .filter(s -> s.getScore() >= 80)
                .map(s -> s.getName())
                .toList();

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

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

- Consumer를 통해, 반환 값은 있고, 리턴 값은 없는 소비형 메서드를 만든 후 내부 반복을 통해 결과를 출력.

내부 반복 vs 외부 반복

스트림을 사용하기 전에 일반적인 반복 방식은 for문, while문과 같은 반복문을 직접 사용해서 데이터를 순회하는 외부 반복(External Iteration) 방식인데, 개발자가 직접 각 요소를 반

복하며 처리

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

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

```
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 없음.
- 코드가 훨씬 간결해지며, 선언형 프로그래밍 스타일을 적용할 수 있음.

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 반복이 더 직관적인 선언형 스타일이나, 외부 반복이 더 나은 경우가 있음.
  - break, continue등이 사용해야 할 때,
  - 반복문 자체가 한 두줄로 끝나고, 특별한 연산이 없다면 등.

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 **맵(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.