6. 메서드 참조

#1.인강/0.자바/7.자바-고급3편

메서드 참조가 필요한 이유

이번에는 특정 상황에서 람다를 좀 더 편리하게 사용할 수 있는 **메서드 참조(Method References)**에 대해 알아보자.

예제1

먼저 코드를 통해 메서드 참조가 필요한 이유를 알아보자.

```
package methodref.start;

import java.util.function.BinaryOperator;

public class MethodRefStartV1 {

    public static void main(String[] args) {
        BinaryOperator<Integer> add1 = (x, y) -> x + y;
        BinaryOperator<Integer> add2 = (x, y) -> x + y;

        Integer result1 = add1.apply(1, 2);
        System.out.println("result1 = " + result1);

        Integer result2 = add2.apply(1, 2);
        System.out.println("result2 = " + result2);
    }

}
```

실행 결과

```
result1 = 3
result2 = 3
```

이 예제는 가장 기본적인 람다를 보여준다. 두 정수를 더하는 간단한 연산을 수행하는데, `add1`과 `add2`는 동일한 기능을 하는 람다를 각각 정의하고 있다.

여기에는 다음과 같은 문제점이 있다.

- 동일한 기능을 하는 람다를 여러 번 작성해야 한다.
- 코드가 중복되어 있어 유지보수가 어려울 수 있다.
- 만약 덧셈 로직이 변경되어야 한다면, 모든 람다를 각각 수정해야 한다.

예제2

```
package methodref.start;

import java.util.function.BinaryOperator;

public class MethodRefStartV2 {

    public static void main(String[] args) {
        BinaryOperator<Integer> add1 = (x, y) -> add(x, y);
        BinaryOperator<Integer> add2 = (x, y) -> add(x, y);

        Integer result1 = add1.apply(1, 2);
        System.out.println("result1 = " + result1);

        Integer result2 = add2.apply(1, 2);
        System.out.println("result2 = " + result2);
    }

    static int add(int x, int y) {
        return x + y;
    }
```

```
}
```

실행 결과

```
result1 = 3
result2 = 3
```

이번 예제에서는 코드 중복 문제를 해결했다.

- 덧셈 로직을 별도의 `add()` 메서드로 분리했다.
- 람다는 `add()` 메서드를 호출한다.
- 로직이 한 곳으로 모여 유지보수가 쉬워졌다.

남은 문제

- 람다를 작성할 때마다 `(x, y) -> add(x, y)` 형태의 코드를 반복해서 작성해야 한다.
- 매개변수를 전달하는 부분이 장황하다.

예제3

이제 메서드 참조를 사용해보자.

```
package methodref.start;

import java.util.function.BinaryOperator;

public class MethodRefStartV3 {

    public static void main(String[] args) {
        BinaryOperator<Integer> add1 = MethodRefStartV3::add; // (x, y) ->
add(x, y)
        BinaryOperator<Integer> add2 = MethodRefStartV3::add; // (x, y) ->
add(x, y)

        Integer result1 = add1.apply(1, 2);
        System.out.println("result1 = " + result1);

        Integer result2 = add2.apply(1, 2);
        System.out.println("result2 = " + result2);
```

```
    }

    static int add(int x, int y) {
        return x + y;
    }
}
```

실행 결과

```
result1 = 3
result2 = 3
```

여기서는 **메서드 참조(Method Reference)** 문법인 `클래스명::메서드명`을 사용하여 `(x, y) -> add(x, y)`라는 람다를 더욱 간단하게 표현했다. 이는 내부적으로 `(x, y) -> add(x, y)`와 동일하게 작동한다.

```
BinaryOperator<Integer> add1 = (x, y) -> add(x, y);
BinaryOperator<Integer> add1 = MethodRefStartV3::add;
```

메서드 참조의 장점

- 메서드 참조를 사용하면 코드가 더욱 간결해지고, 가독성이 향상된다.
- 더 이상 매개변수를 명시적으로 작성할 필요가 없다.
  - 컴파일러가 자동으로 매개변수를 매칭한다.
- 별도의 로직 분리와 함께 재사용성 역시 높아진다.

메서드 참조란?

메서드 참조는 쉽게 말해서, **"이미 정의된 메서드를 그대로 참조하여 람다 표현식을 더 간결하게 작성하는 방법"** 이라고 할 수 있다. 예를 들어 `(x, y) -> add(x, y)`라는 람다는 사실상 매개변수 `x`, `y`를 그대로 `add` 메서드에 전달하기만 하는 코드이므로, `클래스명::메서드명` 형태의 메서드 참조로 간단히 표현할 수 있다. 이렇게 하면 불필요한 매개변수 선언 없이 코드가 깔끔해지고, 가독성도 높아진다.

정리

메서드 참조는 이미 정의된 메서드를 람다로 변환하여 더욱 간결하게 사용할 수 있도록 해주는 문법적 편의 기능이다.
메서드 참조를 사용하면 이미 정의된 메서드를 장황한 람다 대신 간단하고 직관적으로 사용할 수 있다.
이처럼 **람다를 작성할 때, 이미 정의된 메서드를 그대로 호출하는 경우**라면 메서드 참조를 통해 더욱 직관적이고 간결한 코드를 작성할 수 있다.

메서드 참조1 - 시작

메서드 참조(Method Reference)는 **"이미 정의된 메서드를 람다처럼 간결하게 표현"** 할 수 있게 해주는 문법이다.
즉, 람다 내부에서 단순히 어떤 메서드(정적/인스턴스/생성자 등)를 호출만하고 있을 경우, 다음과 같은 형태로 **메서드 참조**를 사용할 수 있다.

```
(x, y) -> 클래스명.메서드명(x, y) // 기존 람다
클래스명::메서드명 // 메서드 참조
```

이때 **람다**와 **메서드 참조**는 **동등하게** 동작한다.
쉽게 이야기해서 메서드 참조는 람다가 단순히 어떤 메서드만 호출하는 경우, 이를 축약해주는 문법이라고 이해하면 된다.

메서드 참조의 4가지 유형

1. 정적 메서드 참조
2. 특정 객체의 인스턴스 메서드 참조
3. 생성자 참조
4. 임의 객체의 인스턴스 메서드 참조

1. 정적 메서드 참조

- 설명: 이름 그대로 정적(static) 메서드를 참조한다.
- 문법: `클래스명::메서드명`
- 예: `Math::max`, `Integer::parseInt` 등

2. 특정 객체의 인스턴스 메서드 참조

- 설명: 이름 그대로 특정 객체의 인스턴스 메서드를 참조한다.
- 문법: `객체명::인스턴스메서드명`
- 예: `person::introduce`, `person::getName` 등

3. 생성자 참조

- 설명: 이름 그대로 생성자를 참조한다.
- 문법: `클래스명::new`
- 예: `Person::new`

4. 임의 객체의 인스턴스 메서드 참조

- 설명: 첫 번째 매개변수(또는 해당 람다가 받을 대상)가 그 메서드를 호출하는 객체가 된다.
- 문법: `클래스명::인스턴스메서드명`
- 예: `Person::introduce`, 같은 람다: `(Person p) -> p.introduce()`

여기서 1 ~ 3은 직관적으로 이해가 될 것이고, 4. 임의 객체의 인스턴스 메서드 참조는 잘 이해가 되지 않을 것이다.
이 기능은 예제를 보아야 이해가 되므로 뒤에서 자세히 알아보자.

이제 코드를 통해 하나씩 알아보자. 먼저 쉬운 1 ~ 3을 알아보고 이후에 4번을 알아보겠다.

예제1

먼저 예제를 위해 간단한 `Person` 객체를 하나 만들어두자.

```
package methodref;

public class Person {

    private String name;

    // 생성자
    public Person() {
        this("Unknown");
    }

    // 생성자, 매개변수
    public Person(String name) {
        this.name = name;
    }

    // 정적 메서드
    public static String greeting() {
        return "Hello";
    }

    // 정적 메서드, 매개변수
    public static String greetingWithName(String name) {
        return "Hello " + name;
    }

    public String getName() {
        return name;
```

```
    }

    // 인스턴스 메서드
    public String introduce() {
        return "I am " + name;
    }

    // 인스턴스 메서드, 매개변수
    public String introduceWithNumber(int number) {
        return "I am " + name + ", my number is " + number;
    }

}
```

```
package methodref;

import java.util.function.Supplier;

public class MethodRefEx1 {

    public static void main(String[] args) {
        // 1. 정적 메서드 참조
        Supplier<String> staticMethod1 = () -> Person.greeting();
        Supplier<String> staticMethod2 = Person::greeting; // 클래스::정적메서드

        System.out.println("staticMethod1: " + staticMethod1.get());
        System.out.println("staticMethod2: " + staticMethod2.get());

        // 2. 특정 객체의 인스턴스 참조
        Person person = new Person("Kim");
        Supplier<String> instanceMethod1 = () -> person.introduce();
        Supplier<String> instanceMethod2 = person::introduce; // 객체::인스턴스메서
드

        System.out.println("instanceMethod1: " + instanceMethod1.get());
        System.out.println("instanceMethod2: " + instanceMethod2.get());

        // 3. 생성자 참조
        Supplier<Person> newPerson1 = () -> new Person();
        Supplier<Person> newPerson2 = Person::new; // 클래스::new
```

```
        System.out.println("newPerson1: " + newPerson1.get());
        System.out.println("newPerson2: " + newPerson2.get());
    }
}
```

실행 결과

```
staticMethod1: Hello
staticMethod2: Hello
instanceMethod1: I am Kim
instanceMethod2: I am Kim
newPerson1: methodref.Person@6442b0a6
newPerson2: methodref.Person@60f82f98
```

1. Person::greeting

- 정적 메서드를 참조한 예시이다.
- `() -> Person.greeting()`을 `Person::greeting`으로 간단히 표현했다.

2. person::introduce

- 특정 객체(`person`)의 인스턴스 메서드를 참조한 예시이다.
- `() -> person.introduce()`를 `person::introduce`로 간략화했다.

3. Person::new

- 생성자를 참조한 예시이다.
- `() -> new Person()`을 `Person::new`로 대체했다.

이렇게 이미 정의된 메서드를 호출하기만 하는 람다는 메서드 참조로 간단히 표현할 수 있다.

메서드 참조에서 ()를 사용하지 않는 이유

- 참고로 메서드 참조의 문법을 잘 보면 뒤에 메서드 명 뒤에 `()`가 없다.
  - 예를 들어서 `Person::greeting()`이 아니라, `Person::greeting`으로 표현한다.
- `()`는 메서드를 즉시 호출한다는 의미를 가진다. 여기서 `()`가 없는 것은 메서드 참조를 하는 시점에는 메서드를 호출하는게 아니라 단순히 **메서드의 이름으로 해당 메서드를 참조만 한다는 뜻**이다.
  - 해당 메서드의 실제 호출 시점은 함수형 인터페이스를 통해서 이후에 이루어진다.

메서드 참조2 - 매개변수1

이번에는 매개변수가 있을 때 메서드를 어떻게 참조하는지 알아보자.

예제2

```
package methodref;

import java.util.function.Function;

// 매개변수 추가
public class MethodRefEx2 {

    public static void main(String[] args) {
        // 1. 정적 메서드 참조
        Function<String, String> staticMethod1 = name ->
Person.greetingWithName(name);
        // 클래스::정적메서드
        Function<String, String> staticMethod2 = Person::greetingWithName;

        System.out.println("staticMethod1 = " + staticMethod1.apply("Kim"));
        System.out.println("staticMethod2 = " + staticMethod2.apply("Kim"));

        // 2. 인스턴스 메서드 참조(특정 객체의 인스턴스 메서드 참조)
        Person instance = new Person("Kim");
        Function<Integer, String> instanceMethod1 = n ->
instance.introduceWithNumber(n);
        // 객체::인스턴스메서드
        Function<Integer, String> instanceMethod2 =
instance::introduceWithNumber;

        System.out.println("instanceMethod1 = " + instanceMethod1.apply(1));
        System.out.println("instanceMethod2 = " + instanceMethod2.apply(1));

        // 3. 생성자 참조
        Function<String, Person> supplier1 = name -> new Person(name);
        Function<String, Person> supplier2 = Person::new; // 클래스::new

        System.out.println("newPerson = " + supplier1.apply("Kim"));
        System.out.println("newPerson = " + supplier2.apply("Lee"));
```

```
    }
}
```

실행 결과

```
staticMethod1 = Hello Kim
staticMethod2 = Hello Kim
instanceMethod1 = I am Kim, my number is 1
instanceMethod2 = I am Kim, my number is 1
newPerson = methodref.Person@3cda1055
newPerson = methodref.Person@7a5d012c
```

이 예제에서는 **매개변수가 있는 메서드 참조**를 다룬다.
메서드 참조의 경우 매개변수를 생략한다. 매개변수가 여러개라면 순서대로 전달된다.

정적 메서드 참조 (매개변수가 있는 경우)

- `Function<String, String> staticMethod1 = name -> Person.greetingWithName(name)`
- `Function<String, String> staticMethod2 = Person::greetingWithName`

특정 인스턴스의 인스턴스 메서드 참조 (매개변수가 있는 경우)

- `Function<Integer, String> instanceMethod1 = n -> instance.introduceWithNumber(n)`
- `Function<Integer, String> instanceMethod2 = instance::introduceWithNumber`

생성자 참조 (매개변수가 있는 생성자):

- `Function<String, Person> supplier1 = name -> new Person(name)`
- `Function<String, Person> supplier2 = Person::new`

메서드 참조에서 매개변수를 생략하는 이유

함수형 인터페이스의 시그니처(매개변수와 반환 타입)가 이미 정해져 있고, 컴파일러가 그 시그니처를 바탕으로 메서드 참조와 연결해 주기 때문에 명시적으로 매개변수를 작성하지 않아도 자동으로 추론되어 호출된다.

예를 들어, `Function<String, String>`이라는 함수형 인터페이스가

- 입력: `String`

- 출력: `String`

이라는 시그니처를 갖고 있기 때문에, `Person::greetingWithName`를 보면 컴파일러가 "이 `Function<String, String>`을 만족시키려면 `greetingWithName(String name)`을 호출하면 되겠구나"하고 자동으로 판단해 매개변수를 전달한다.

- `(String name) -> Person.greetingWithName(name)` 따라서 이런 람다 대신에
- `Person::greetingWithName` 이런 메서드 참조로 간결하게 표현할 수 있다.

매개변수를 포함한 메서드 호출도 메서드 참조를 사용하면 더욱 간편해진다.

메서드 참조3 - 임의 객체의 인스턴스 메서드 참조

메서드 참조는 다음 4가지 유형이 있다고 설명했다.

메서드 참조의 4가지 유형

1. 정적 메서드 참조
2. 특정 객체의 인스턴스 메서드 참조
3. 생성자 참조
4. 임의 객체의 인스턴스 메서드 참조

지금까지 1 ~ 3을 알아보았고, 이제 남은 4를 알아보자.
1 ~ 3은 직관적으로 이해하는데 어려움이 없을 것이다. 그런데 지금부터 설명할 **임의 객체의 인스턴스 메서드 참조**는 조금 특별하다. 그래서 앞에서 설명한 1 ~ 3과는 다른 방식으로 이해해야 한다.

예제 코드를 통해서 자세히 알아보자.

예제3

```
package methodref;

import java.util.function.Function;

public class MethodRefEx3 {

    public static void main(String[] args) {
        // 4. 임의 객체의 인스턴스 메서드 참조(특정 타입의)
```

```
        Person person1 = new Person("Kim");
        Person person2 = new Person("Park");
        Person person3 = new Person("Lee");

        // 람다
        Function<Person, String> fun1 = (Person person) -> person.introduce();
        System.out.println("person1.introduce = " + fun1.apply(person1));
        System.out.println("person2.introduce = " + fun1.apply(person2));
        System.out.println("person3.introduce = " + fun1.apply(person3));

        // 메서드 참조, 타입이 첫 번째 매개변수가 됨, 그리고 첫 번째 매개변수의 메서드를 호출, 나머지는 순서대로 매개변수에 전달
        Function<Person, String> fun2 = Person::introduce; // 타입::인스턴스메서드
        System.out.println("person1.introduce = " + fun2.apply(person1));
        System.out.println("person2.introduce = " + fun2.apply(person2));
        System.out.println("person3.introduce = " + fun2.apply(person3));
    }
}
```

실행 결과

```
person1.introduce = I am Kim
person2.introduce = I am Park
person3.introduce = I am Lee
person1.introduce = I am Kim
person2.introduce = I am Park
person3.introduce = I am Lee
```

람다를 먼저 분석해보자.

람다 정의

```
Function<Person, String> fun1 = (Person person) -> person.introduce()
```

- 이 람다는 `Person` 타입을 매개변수로 받는다. 그리고 매개변수로 넘겨받은 `person` 인스턴스의 `introduce()` 인스턴스 메서드를 호출한다.
- `()` 는 생략할 수 있다. 이해를 돕기 위해 남겨두었다.

람다 실행

```
System.out.println("person1.introduce = " + fun1.apply(person1));
System.out.println("person2.introduce = " + fun1.apply(person2));
System.out.println("person3.introduce = " + fun1.apply(person3));
```

- 앞서 정의한 람다는 `Function<Person, String>` 함수형 인터페이스를 사용한다. 따라서 `Person` 타입의 인스턴스를 인자로 받고, `String`을 반환한다.
- 코드에서 보면 `apply()`에 `person1`, `person2`, `person3`을 각각 전달한 것을 확인할 수 있다.
  - `person1`을 람다에 전달하면 `person1.introduce()`가 호출된다.
  - `person2`를 람다에 전달하면 `person2.introduce()`가 호출된다.
  - `person3`을 람다에 전달하면 `person3.introduce()`가 호출된다.

이 람다는 **매개변수로 지정한 특정 타입의 객체에 대해 동일한 메서드를 호출**하는 패턴을 보인다.

`(Person person) -> person.introduce()`

- **매개변수로 지정한 특정 타입**: `Person`
- **메서드**: `introduce()`

조금 더 풀어서 이야기하면 매개변수로 지정한 **특정 타입의 임의 객체의 인스턴스 메서드를 참조**한다.

- **매개변수로 지정한 특정 타입**: `Person`
- **임의 객체**: `person1`, `person2`, `person3`, 또는 `Person` 타입을 구현한 어떠한 객체든 해당 람다에 전달할 수 있음
- **인스턴스 메서드**: `introduce()`

이렇게 특정 타입의 임의 객체에 대해 동일한 인스턴스 메서드를 호출하는 패턴을 메서드 참조로 손쉽게 표현할 수 있다.

```
Function<Person, String> fun1 = (Person person) -> person.introduce() // 람다
Function<Person, String> fun2 = Person::introduce // 메서드 참조 (타입::인스턴스메서드)
```

임의 객체의 인스턴스 메서드 참조

(Reference to an instance method of an arbitrary object of a particular type)

이런 메서드 참조를 **특정 타입의 임의 객체의 인스턴스 참조**라 한다.

여기서는 줄여서 **임의 객체의 인스턴스 참조**라 하겠다.

임의 객체의 인스턴스 참조는 `클래스명::인스턴스메서드`, 예) `Person::introduce`와 같이 사용한다.

- 주의! 왼쪽이 **클래스명**이고, 오른쪽이 **인스턴스 메서드**이다!

여기서 사용한 `introduce()`는 인스턴스 메서드이다.

```
public class Person {
    // 인스턴스 메서드
    public String introduce() {
        return "I am " + name;
    }
}
```

`Person::introduce`와 같이 선언하면 다음과 같은 람다가 된다.

```
Person::introduce
1. 왼쪽에 지정한 클래스를 람다의 첫 번째 매개변수로 사용한다.
(Person person)

2. 오른쪽에 지정한 '인스턴스 메서드'를 첫 번째 매개변수를 통해 호출한다.
(Person person) -> person.introduce()
```

정리

앞서 우리가 학습한 내용을 다시 한번 정리해보자.

1. 정적 메서드 참조 `클래스명::클래스메서드(Person::greeting)`
2. 특정 객체의 인스턴스 메서드 참조: `객체명::인스턴스메서드(person::introduce)`
3. 생성자 참조: `클래스명::new(Person::new)`
4. 임의 객체의 인스턴스 메서드 참조: `클래스명::인스턴스메서드(Person::introduce)`

여기서 **2. 특정 객체의 인스턴스 메서드 참조**와 **4. 임의 객체의 인스턴스 메서드 참조**가 좀 헷갈릴 수 있다. 참고로 두 기능은 완전히 다른 기능이다.

먼저 둘의 문법이 다르다. 둘다 "인스턴스 메서드"를 호출하지만, 하나는 객체명을 사용하고, 하나는 클래스명을 사용한다.

- **특정 객체의 인스턴스 메서드 참조**: `객체명::인스턴스메서드(person::introduce)`
- **임의 객체의 인스턴스 메서드 참조**: `클래스명::인스턴스메서드(Person::introduce)`

둘의 차이를 조금 더 자세히 알아보자.

2. 특정 객체의 인스턴스 메서드 참조

이 기능은 메서드 참조를 선언할 때 부터 이름 그대로 특정 객체(인스턴스)를 지정해야 한다.

```
person::introduce // 메서드 참조: 인스턴스 person을 지정한다.
() -> person.introduce() // 람다: 지정한 person의 인스턴스 메서드를 사용한다.

// 실행 시점: 이미 지정된 인스턴스가 사용된다.
instanceMethod1.get()
```

특정 객체의 인스턴스 메서드 참조는 선언 시점부터 이미 인스턴스가 지정되어 있다. 따라서 람다를 실행하는 시점에 인스턴스를 변경할 수 없다.

4. 임의 객체의 인스턴스 메서드 참조

이 기능은 메서드 참조를 선언할 때는 어떤 객체(인스턴스)가 대상이 될 지 모른다.

```
Person::introduce // 메서드 참조: Person이라는 타입만 지정한다. 어떤 인스턴스가 사용될지는 아직 모른다.
(Person person) -> person.introduce() // 람다: 매개변수로 넘어오는 person 인스턴스의 메서드를 사용

// 실행 시점: 실행 시점에 인스턴스를 외부에서 전달해서 변경할 수 있다.
fun1.apply(person1)
fun1.apply(person2)
fun1.apply(person3)
```

임의 객체의 인스턴스 메서드 참조는 선언 시점에 호출할 인스턴스를 지정하지 않는다. 대신에 호출 대상을 매개변수로 선언해두고, 실행 시점에 호출할 인스턴스를 받는다. **실행 시점이 되어야 어떤 객체가 호출되는지 알 수 있으므로 "임의 객체의 인스턴스 메서드 참조"라 한다.**

둘의 핵심적인 차이는 메서드 참조나 람다를 정의하는 시점에 호출할 대상 인스턴스가 고정되는 것인지 아닌지에 달려 있다.

이름 그대로

- **특정 객체의 인스턴스 메서드 참조**는 선언 시점에 메서드를 호출할 **특정 객체**가 고정된다.
- **임의 객체의 인스턴스 메서드 참조**는 선언 시점에 메서드를 호출할 특정 객체가 고정되지 않는다. 대신에 실행 시점에 인자로 넘긴 **임의의 객체**가 사용된다.

임의 객체의 인스턴스 메서드 참조는 정확히는 "**특정 타입**의 **임의 객체**의 인스턴스 참조"라 한다. 각 단어를 풀어보면 다

음과 같다.

- **특정 타입**: 타입은 선언시에 특정한 타입으로 고정된다. 여기서는 `Person` 타입으로 고정했다.
- **임의 객체**: 메서드를 호출할 객체가 고정되지 않는다. `Person` 타입을 사용하는 임의 객체가 실행 시점에 사용될 수 있다.

임의 객체의 인스턴스 메서드 참조는 꼭 필요해 보이지는 않는데, 왜 이런 기능을 자바가 제공할까? 사실 메서드 참조 중에는 이 기능이 가장 많이 사용된다. 다양한 활용 예를 통해 자세히 알아보자.

메서드 참조4 - 활용1

임의 객체의 인스턴스 메서드 참조가 실제 어떻게 사용되는지 알아보자.

예제4

```
package methodref;

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

public class MethodRefEx4 {

    public static void main(String[] args) {
        List<Person> personList = List.of(
                new Person("Kim"),
                new Person("Park"),
                new Person("Lee")
        );

        List<String> result1 = mapPersonToString(personList, (Person p) ->
p.introduce());
        List<String> result2 = mapPersonToString(personList,
Person::introduce);
        System.out.println("result1 = " + result1);
        System.out.println("result2 = " + result2);

        List<String> upperResult1 = mapStringToString(result1, (String s) ->
```

```
s.toUpperCase());
        List<String> upperResult2 = mapStringToString(result2,
String::toUpperCase);
        System.out.println("upperResult1 = " + upperResult1);
        System.out.println("upperResult2 = " + upperResult2);
    }

    static List<String> mapPersonToString(List<Person> personList,
Function<Person, String> fun) {
        List<String> result = new ArrayList<>();
        for (Person p : personList) {
            String applied = fun.apply(p);
            result.add(applied);
        }
        return result;
    }

    static List<String> mapStringToString(List<String> strings,
Function<String, String> fun) {
        List<String> result = new ArrayList<>();
        for (String s : strings) {
            String applied = fun.apply(s);
            result.add(applied);
        }
        return result;
    }

}
```

실행 결과

```
result1 = [I am Kim, I am Park, I am Lee]
result2 = [I am Kim, I am Park, I am Lee]
upperResult1 = [I AM KIM, I AM PARK, I AM LEE]
upperResult2 = [I AM KIM, I AM PARK, I AM LEE]
```

이 예제는 메서드 참조를 활용하여 리스트의 데이터를 변환하는 예시를 보여준다.

코드 분석

`mapPersonToString()` 메서드

- `Person` 객체 리스트를 받아서 `String` 리스트로 변환하는 메서드이다.
- 변환 로직을 `Function<Person, String>`으로 받아 유연하게 처리한다.

```
// 람다 사용
List<String> result1 = mapPersonToString(personList, (Person p) ->
p.introduce());
// 메서드 참조 사용
List<String> result2 = mapPersonToString(personList, Person::introduce);
```

여기서는 **임의 객체의 인스턴스 메서드 참조**를 사용했다.

- `Person::introduce`는 `클래스::인스턴스메서드`이다. 이런 형식이 나오면 임의 객체의 인스턴스 메서드 참조로 이해하면 된다.
- 따라서 왼쪽에 지정한 `Person`이 첫 번째 매개변수가 되고, 오른쪽에 지정한 인스턴스 메서드를 해당 매개변수가 호출한다.
  - 위의 람다와 같은 형식이다.
- `fun.apply(p)` 메서드에 `new Person("Kim")`, `new Person("Park")`, `new Person("Lee")` 각각의 인스턴스가 인자로 넘어가면서 해당 인스턴스의 `introduce()` 메서드가 호출된다.

`mapStringToString()` 메서드

- `String` 리스트를 다른 형태의 `String` 리스트로 변환하는 메서드이다.
- 변환 로직을 `Function<String, String>`으로 받는다.

```
// 람다 사용
List<String> upperResult1 = mapStringToString(result1, (String s) ->
s.toUpperCase());
// 메서드 참조 사용
List<String> upperResult2 = mapStringToString(result2, String::toUpperCase);
```

여기서는 **임의 객체의 인스턴스 메서드 참조**를 사용했다.

- `String::toUpperCase`는 `클래스::인스턴스메서드`이다. 이런 형식이 나오면 임의 객체의 인스턴스 메서드 참조로 이해하면 된다.
- 따라서 왼쪽에 지정한 `String`이 첫 번째 매개변수가 되고, 오른쪽에 지정한 인스턴스 메서드를 해당 매개변수가 호출한다.
  - 위의 람다와 같은 형식이다.
- `fun.apply(s)` 메서드에 `"I am Kim"`, `"I am Park"`, `"I am Lee"` 각각의 `String` 인스턴스가 인자로 넘어가면서 해당 인스턴스의 `toUpperCase()` 메서드가 호출된다.

람다 대신에 메서드 참조를 사용한 덕분에 코드가 더 간결해지고, 의도가 더 명확하게 드러나는 것을 확인할 수 있다.

- `mapPersonToString(personList, Person::introduce)`
  - `Person` 리스트에 있는 각각의 `Person` 인스턴스에 `introduce`를 호출하고 그 결과를 리스트로 반환
- `mapStringToString(result2, String::toUpperCase)`
  - `String` 리스트에 있는 각각의 `String` 인스턴스에 `toUpperCase`를 호출하고 그 결과를 리스트로 반환

우리가 앞서 만든 스트림을 사용하면 리스트에 들어있는 다양한 데이터를 더 쉽게 변환할 수 있을 것 같다. 한 번 시도해보자.

메서드 참조5 - 활용2

이번에는 스트림에 메서드 참조를 활용해보자.

예제5

```
package methodref;

import lambda.lambda5.mystream.MyStreamV3;

import java.util.List;

public class MethodRefEx5 {

    public static void main(String[] args) {
        List<Person> personList = List.of(
                new Person("Kim"),
                new Person("Park"),
                new Person("Lee")
        );

        List<String> result1 = MyStreamV3.of(personList)
                .map(person -> person.introduce())
                .map(str -> str.toUpperCase())
                .toList();
        System.out.println("result1 = " + result1);
```

```
        List<String> result2 = MyStreamV3.of(personList)
                .map(Person::introduce) // person -> person.introduce()
                .map(String::toUpperCase) // (String name) ->
name.toUpperCase()
                .toList();
        System.out.println("result2 = " + result2);
    }
}
```

- `MyStreamV3`는 람다 활용에서 이전에 만든 클래스이다.

실행 결과

```
result1 = [I AM KIM, I AM PARK, I AM LEE]
result2 = [I AM KIM, I AM PARK, I AM LEE]
```

코드 분석

람다 표현식 사용

```
List<String> result1 = MyStreamV3.of(personList)
        .map(person -> person.introduce())
        .map(name -> name.toUpperCase())
        .toList();
```

람다를 사용하여 각 `Person` 객체에 대해 `introduce()` 메서드를 호출한 후, 결과 문자열을 모두 대문자로 변환한다.

메서드 참조 사용

```
List<String> result2 = MyStreamV3.of(personList)
        .map(Person::introduce)    // person -> person.introduce()
        .map(String::toUpperCase)  // name -> name.toUpperCase()
        .toList();
```

위와 동일한 처리 과정을 메서드 참조로 간략하게 표현했다.

메서드 참조의 장점

메서드 참조를 사용하면 람다 표현식을 더욱 직관적으로 표현할 수 있으며, 각 처리 단계에서 호출되는 메서드가 무엇인지 쉽게 파악할 수 있다. 이처럼 **람다**로도 충분히 표현할 수 있지만, 내부적으로 호출만 하는 간단한 람다라면 **메서드 참**

조가 더 짧고 명확하게 표현될 수 있다. 이런 방식은 코드 가독성을 높이는 장점이 있다. 물론 메서드 참조 방식에 익숙해지는데 어느정도 시간은 필요하다.

메서드 참조6 - 매개변수2

이번에는 임의 객체의 인스턴스 메서드 참조에서 매개변수가 늘어나면 어떻게 되는지 알아보자.

예제6

이 예제는 임의 객체의 인스턴스 메서드 참조에서 매개변수가 여러 개인 경우의 메서드 참조 사용법을 보여준다.

```
package methodref;

import java.util.function.BiFunction;

// 매개변수 추가
public class MethodRefEx6 {

    public static void main(String[] args) {
        // 4. 임의 객체의 인스턴스 메서드 참조(특정 타입의)
        Person person = new Person("Kim");

        // 람다
        BiFunction<Person, Integer, String> fun1 =
                (Person p, Integer number) -> p.introduceWithNumber(number);

        System.out.println("person.introduceWithNumber = " +
fun1.apply(person, 1));

        // 메서드 참조, 타입이 첫 번째 매개변수가 됨, 그리고 첫 번째 매개변수의 메서드를 호출
        // 나머지는 순서대로 매개변수에 전달
        // 타입::메서드명
        BiFunction<Person, Integer, String> fun2 =
Person::introduceWithNumber;
        System.out.println("person.introduceWithNumber = " +
fun1.apply(person, 1));

    }
}
```

실행 결과

```
person.introduceWithNumber = I am Kim, my number is 1
person.introduceWithNumber = I am Kim, my number is 1
```

코드 분석

```
// 람다 사용
BiFunction<Person, Integer, String> fun1 =
    (Person p, Integer number) -> p.introduceWithNumber(number);

// 메서드 참조 사용
BiFunction<Person, Integer, String> fun2 =
    Person::introduceWithNumber;
```

- `BiFunction<Person, Integer, String>` 인터페이스를 사용하여 `(Person, Integer) -> String` 형태의 람다/메서드 참조를 구현한다.
- `fun1` 에서는 람다를 사용하여 `p.introduceWithNumber(number)` 를 호출한다.
- `fun2` 에서는 `Person::introduceWithNumber` 라는 메서드 참조를 사용한다. 첫 번째 매개변수(`Person`)가 메서드를 호출하는 객체가 되고, 두 번째 매개변수(`Integer`)가 `introduceWithNumber()` 의 실제 인자로 전달된다. 첫 번째 이후의 매개변수는 모두 순서대로 실제 인자로 전달된다.

이처럼 **임의 객체의 인스턴스 메서드 참조**는 함수형 인터페이스의 시그니처에 따라

- 첫 번째 인자를 **호출 대상 객체**로
- 나머지 인자들은 순서대로 **해당 메서드의 매개변수**로 전달한다.

정리

메서드 참조의 필요성

- **람다에서 이미 정의된 메서드를 단순히 호출**하기만 하는 경우, 메서드 참조로 더 간결하게 표현할 수 있다.
- 코드 중복을 줄이고 가독성을 높여주며, 유지보수 측면에서도 편리하다.

1. **간결성**

    - 람다 표현식을 더욱 간단하게 표현할 수 있다.
    - 매개변수를 생략한다.
    - 특히 메서드 체이닝에서 코드가 매우 깔끔해진다.
2. **가독성**
    - 메서드 이름을 직접 사용하여 의도가 더 명확해진다.
    - 복잡한 람다 표현식을 단순화할 수 있다.
3. **유연성**
    - 다양한 함수형 인터페이스와 함께 사용할 수 있다.
    - 스트림 API와 결합하여 강력한 데이터 처리 가능하다. (`MyStream`을 떠올려보자.)
4. **재사용성**
    - 기존 메서드를 람다 표현식으로 쉽게 변환할 수 있다.
    - 동일한 메서드를 여러 컨텍스트에서 재사용할 수 있다.

메서드 참조의 4가지 유형

- 정적 메서드 참조: `클래스명::메서드명`
- 특정 객체의 인스턴스 메서드 참조: `객체명::메서드명`
- 생성자 참조: `클래스명::new`
- 임의 객체의 인스턴스 메서드 참조: `클래스명::메서드명`
    - 클래스명으로 지정한 첫 번째 매개변수가 곧 호출 대상 객체가 된다.

임의 객체의 인스턴스 메서드 참조

- 특정 타입의 객체가 런타임에 주어지면, 그 객체를 사용해 **인스턴스 메서드**를 호출하는 방식을 간단하게 표현한다.
- 예: `(Person p) -> p.introduce()` 대신 `Person::introduce` 로 표기

활용 예시

- 컬렉션(`List`, `Map` 등)의 원소들을 변환(`map`)할 때, or 스트림(`Stream`)에서 `map`, `filter` 등을 사용할 때,
- 이미 정의된 메서드를 그대로 쓸 수 있다면 **메서드 참조**를 사용해 더욱 깔끔하고 직관적인 코드를 작성할 수 있다.

람다와의 관계

- 메서드 참조는 람다를 **메서드 호출만으로 축약한 문법**이며, 내부 동작은 사실상 동일하다.
- 람다로 표현하기에 직관적인 경우에는 람다를, 메서드 참조가 더 간결하고 읽기 쉬운 경우에는 메서드 참조를 사용하면 된다.

메서드 참조는 자바 코드의 간결함과 가독성을 향상시켜주는 유용한 기능이므로, 여러 상황에서 적극적으로 활용하는 것을 추천한다. (물론 익숙해지는데 어느정도의 시간은 필요하다)