Lambda Expression

1. Lambda Expression(람다식, 람다 표현식)

1-1. 람다식이란?

- Java 8에서 도입된 기능으로, 함수형 프로그래밍을 지원하기 위한 핵심 기능
- 메서드를 하나의 식으로 표현해 익명 함수(Anonymous Function)를 생성하는 식

기본 문법

```
(parameters) → { statements }
```

1-2. 람다식의 특징

- 메서드의 이름과 반환형을 생략하여 코드를 간결하게 만듦
- 익명 클래스의 객체를 생성하는 번거로움을 줄임
- 함수형 인터페이스의 구현에 사용
- Stream API와 함께 사용하여 컬렉션을 효과적으로 처리

💡 람다식은 객체 지향 프로그래밍의 사상을 위배하기 때문에 호불호가 갈리기도한다.

1-3. 람다식 예제

기존 방식

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello World!");
    }
};
```

Lambda 사용

```
Runnable runnable = () → System.out.println("Hello World!");
```

1-4. 람다식 문법 종류

- **기본 문법:** (매개변수) -> { 실행문; }
- **매개변수가 하나인 경우:** 괄호 생략 가능 - x → { 실행문; }
- **실행문이 하나인 경우:** 중괄호 생략 가능 - (x) -> 실행문
- **매개변수 타입 생략:** 컴파일러가 추론 가능한 경우 타입 생략 가능

```
// 기본 문법
(int x, int y) -> { return x + y; }

// 매개변수가 하나인 경우
x -> { return x * x; }

// 실행문이 하나인 경우
(x, y) -> x + y

// 매개변수 타입 생략
(x, y) -> { return x + y; }
```

---

2. 함수형 인터페이스(Functional Interface)

2-1. 함수형 인터페이스란?

- 람다식을 사용하기 위해서는 인터페이스
- 단 하나의 `추상 메서드만`을 가져야 함

2-2 함수형 인터페이스 선언 방법

```
@FunctionalInterface
public interface MyFunctionalInterface {
    // 단 하나의 추상 메서드만 가질 수 있음
    void myMethod();

    //void otherMethod(); // 다른 추상 메서드 선언 시 오류 발생

}
```

- **@FunctionalInterface 어노테이션**
  - 해당 인터페이스가 함수형 인터페이스임을 명시
  - 컴파일 시 해당 인터페이스가 함수형 인터페이스가 맞는지 검사
    (추상 메서드를 하나만 포함 인터페이스인지 검사)

2-3. 함수형 인터페이스 예시

함수형 인터페이스 선언

```
@FunctionalInterface
public interface Calculator {

    public int sumTwoNumber(int a, int b);
}
```

Calculator 인터페이스 구현체 (람다식 사용 코드와 비교를 위해 생성)

```
public class CalculatorImpl implements Calculator {

    @Override
    public int sumTwoNumber(int a, int b) {
        return a + b;
    }
}
```

실행 클래스

```
public class Main{
    public static void main(String[] args) {


        /* 방법 1. 인터페이스를 상속받은 클래스를 정의하여 기능을 완성 후 사용하는 방법 */
        // CalculatorImpl 클래스 구현
        Calculator calc1 = new CalculatorImpl();
        System.out.println("calc1.sumTwoNumber(1, 2) = " + calc1.sumTwoNumber(1, 2));


        /* 방법 2. 익명클래스를 활용하여 메소드 재정의 후 사용하는 방법 */
        Calculator calc2 = new Calculator(){
            @Override
            public int sumTwoNumber(int a, int b) {
                return a + b;
            }
        };

        System.out.println("calc2.sumTwoNumber(10, 20) = " + calc2.sumTwoNumber(10, 20));


        /* 방법 3. 람다식을 활용하는 방법 */
        Calculator calc3 = (a, b) -> a + b;
        System.out.println("calc3.sumTwoNumber(30, 40) = " + calc3.sumTwoNumber(30, 40));

        Calculator calc4 = (a,b) -> a * 2 + b * 2;
        System.out.println("calc4.sumTwoNumber(10,20) = " + calc4.sumTwoNumber(10, 20));


    }
}
```

▼ 실행 결과

```
calc1.sumTwoNumber(1, 2) = 3
calc2.sumTwoNumber(10, 20) = 30
calc3.sumTwoNumber(30, 40) = 70
calc4.sumTwoNumber(10,20) = 60
```

3. java.util.function 패키지 제공 함수형 인터페이스

- JDK8에서 람다식을 추가할 때 자주 사용되는 함수형 인터페이스를 API로 제공함
- java.util.function 패키지에 존재함

주요 함수형 인터페이스

인터페이스	설명
Consumer<T>	매개변수 O, 반환값 X
Supplier<T>	매개변수 X, 반환값 O
Function<T,R>	매개변수 O, 반환값 O
Operator<T>	매개변수와 반환값의 타입이 같음
Predicate<T>	조건식을 표현하는데 사용

3-1. Consumer 계열

- 매개변수를 받아서 소비하는 함수형 인터페이스
- 반환값이 없음

인터페이스	메서드	설명
Consumer<T>	void accept(T t)	객체 T를 매개변수로 받아 소비
BiConsumer<T,U>	void accept(T t, U u)	객체 T와 U를 매개변수로 받아 소비
IntConsumer	void accept(int value)	int 값을 매개변수로 받아 소비
LongConsumer	void accept(long value)	long 값을 매개변수로 받아 소비
DoubleConsumer	void accprt(double value)	double 값을 매개변수로 받아 소비
ObjIntConsumer<T>	void accept(T t, int value)	객체 T와 int 값을 매개변수로 받아 소비
ObjLongConsumer<T>	void accept(T t, longvalue)	객체 T와 long 값을 매개변수로 받아 소비
ObjDoubleConsumer<T>	void accept(T t, doublevalue)	객체 T와 double 값을 매개변수로 받아 소비

예시 코드

```
// Consumer<T>
Consumer<String> printString = str → System.out.println(str);
printString.accept("안녕하세요"); // 안녕하세요

// BiConsumer<T,U>
BiConsumer<String, Integer> printNameAge = (name, age) →
    System.out.println(name + "님은 " + age + "살입니다");
printNameAge.accept("홍길동", 30); // 홍길동님은 30살입니다

// IntConsumer
IntConsumer printInt = value → System.out.println("정수값: " + value);
printInt.accept(100); // 정수값: 100

// LongConsumer
LongConsumer printLong = value → System.out.println("long값: " + value);
printLong.accept(100L); // long값: 100

// DoubleConsumer
DoubleConsumer printDouble = value → System.out.println("double값: " + value);
printDouble.accept(3.14); // double값: 3.14
```

```
// ObjIntConsumer<T>
ObjIntConsumer<String> printNameAndAge = (name, age) ->
    System.out.println(name + "의 나이는 " + age);
printNameAndAge.accept("김철수", 25); // 김철수의 나이는 25

// ObjLongConsumer<T>
ObjLongConsumer<String> printNameAndLong = (name, value) ->
    System.out.println(name + "의 번호는 " + value);
printNameAndLong.accept("이영희", 12345678900L); // 이영희의 번호는 12345678900

// ObjDoubleConsumer<T>
ObjDoubleConsumer<String> printNameAndScore = (name, score) ->
    System.out.println(name + "의 점수는 " + score + "점");
printNameAndScore.accept("박지성", 92.5); // 박지성의 점수는 92.5점
```

3-2. Supplier 계열

- 매개변수 없이 값을 제공하는 함수형 인터페이스

인터페이스	메서드	설명
Supplier<T>	T get()	T 타입 객체를 반환
BooleanSupplier	boolean getAsBoolean()	boolean 값을 반환
IntSupplier	int getAsInt()	int 값을 반환
LongSupplier	long getAsLong()	long 값을 반환
DoubleSupplier	double getAsDouble()	double 값을 반환

```
// Supplier<T> - 매개변수 없이 T 타입 객체를 반환
Supplier<String> messageSupplier = () -> "안녕하세요";
System.out.println(messageSupplier.get()); // 안녕하세요

// BooleanSupplier - boolean 값을 반환
BooleanSupplier isWorkingHour = () -> {
    int hour = LocalDateTime.now().getHour();
    return hour >= 9 && hour <= 18;
};
System.out.println(isWorkingHour.getAsBoolean()); // 현재 시간이 근무시간이면 true

// IntSupplier - int 값을 반환
IntSupplier randomInt = () -> (int)(Math.random() * 100);
System.out.println(randomInt.getAsInt()); // 0~99 사이의 랜덤 숫자 출력

// LongSupplier - long 값을 반환
LongSupplier timestamp = () -> System.currentTimeMillis();
System.out.println(timestamp.getAsLong()); // 현재 시간을 밀리초로 출력

// DoubleSupplier - double 값을 반환
DoubleSupplier pi = () -> Math.PI;
System.out.println(pi.getAsDouble()); // 3.141592653589793 출력
```

3-3. Function 계열

- 매개변수를 받아서 다른 타입으로 변환하여 반환하는 함수형 인터페이스

인터페이스	메서드	설명
Function<T,R>	R apply(T t)	T 타입을 매개변수로 받아 R 타입을 반환
BiFunction<T,U,R>	R apply(T t, U u)	T와 U를 매개변수로 받아 R을 반환
ToIntFunction<T>	int applyAsInt(T t)	T를 매개변수로 받아 int를 반환
ToLongFunction<T>	long applyAsLong(T t)	T를 매개변수로 받아 long을 반환
ToDoubleFunction<T>	double applyAsDouble(T t)	T를 매개변수로 받아 double을 반환
IntFunction<R>	R apply(int value)	int를 매개변수로 받아 R을 반환
LongFunction<R>	R apply(long value)	long을 매개변수로 받아 R을 반환
DoubleFunction<R>	R apply(double value)	double을 매개변수로 받아 R을 반환
IntToLongFunction	long applyAsLong(int value)	int를 매개변수로 받아 long을 반환
IntToDoubleFunction	double applyAsDouble(int value)	int를 매개변수로 받아 double을 반환
LongToIntFunction	int applyAsInt(long value)	long을 매개변수로 받아 int를 반환
LongToDoubleFunction	double applyAsDouble(long value)	long을 매개변수로 받아 double을 반환
DoubleToIntFunction	int applyAsInt(double value)	double을 매개변수로 받아 int를 반환
DoubleToLongFunction	long applyAsLong(double value)	double을 매개변수로 받아 long을 반환

```
// Function<T,R> - T 타입을 받아서 R 타입으로 변환
Function<String, Integer> strLength = str -> str.length();
System.out.println(strLength.apply("Hello")); // 5

// BiFunction<T,U,R> - T와 U를 받아서 R로 변환
BiFunction<String, Integer, String> formatAge = (name, age) ->
    name + "는 " + age + "살입니다.";
System.out.println(formatAge.apply("홍길동", 20)); // 홍길동는 20살입니다.

// ToIntFunction<T> - T를 받아서 int로 변환
ToIntFunction<String> parseToInt = str -> Integer.parseInt(str);
System.out.println(parseToInt.applyAsInt("123")); // 123

// ToLongFunction<T> - T를 받아서 long으로 변환
ToLongFunction<String> parseToLong = str -> Long.parseLong(str);
System.out.println(parseToLong.applyAsLong("123")); // 123L

// ToDoubleFunction<T> - T를 받아서 double로 변환
ToDoubleFunction<String> parseToDouble = str -> Double.parseDouble(str);
System.out.println(parseToDouble.applyAsDouble("123.45")); // 123.45

// IntFunction<R> - int를 받아서 R로 변환
IntFunction<String> intToString = num -> String.valueOf(num);
System.out.println(intToString.apply(123)); // "123"

// LongFunction<R> - long을 받아서 R로 변환
LongFunction<String> longToString = num -> String.valueOf(num);
System.out.println(longToString.apply(123L)); // "123"

// DoubleFunction<R> - double을 받아서 R로 변환
DoubleFunction<String> doubleToString = num -> String.format("%.2f", num);
System.out.println(doubleToString.apply(123.456)); // "123.46"

// IntToLongFunction - int를 받아서 long으로 변환
IntToLongFunction intToLong = num -> (long) num;
System.out.println(intToLong.applyAsLong(123)); // 123L

// IntToDoubleFunction - int를 받아서 double로 변환
IntToDoubleFunction intToDouble = num -> (double) num;
```

```
System.out.println(intToDouble.applyAsDouble(123)); // 123.0

// LongToIntFunction - long을 받아서 int로 변환
LongToIntFunction longToInt = num -> (int) num;
System.out.println(longToInt.applyAsInt(123L)); // 123

// LongToDoubleFunction - long을 받아서 double로 변환
LongToDoubleFunction longToDouble = num -> (double) num;
System.out.println(longToDouble.applyAsDouble(123L)); // 123.0

// DoubleToIntFunction - double을 받아서 int로 변환
DoubleToIntFunction doubleToInt = num -> (int) num;
System.out.println(doubleToInt.applyAsInt(123.45)); // 123

// DoubleToLongFunction - double을 받아서 long으로 변환
DoubleToLongFunction doubleToLong = num -> (long) num;
System.out.println(doubleToLong.applyAsLong(123.45)); // 123L
```

3-4. Operator 계열

- 매개변수와 반환 값의 타입이 같은 함수형 인터페이스

인터페이스	메서드	설명
UnaryOperator<T>	T apply(T t)	T 타입의 매개변수를 받아 동일한 타입 반환
BinaryOperator<T>	T apply(T t1, T t2)	동일한 타입의 매개변수 두 개를 받아 동일한 타입 반환
IntUnaryOperator	int applyAsInt(int operand)	int 매개변수를 받아 int 반환
IntBinaryOperator	int applyAsInt(int left, int right)	int 매개변수 두 개를 받아 int 반환
LongUnaryOperator	long applyAsLong(long operand)	long 매개변수를 받아 long 반환
LongBinaryOperator	long applyAsLong(long left, long right)	long 매개변수 두 개를 받아 long 반환
DoubleUnaryOperator	double applyAsDouble(double operand)	double 매개변수를 받아 double 반환
DoubleBinaryOperator	double applyAsDouble(double left, double right)	double 매개변수 두 개를 받아 double 반환

```
// UnaryOperator<T> - T 타입의 매개변수를 받아 동일한 타입 반환
UnaryOperator<String> toUpperCase = str -> str.toUpperCase();
System.out.println(toUpperCase.apply("hello")); // HELLO

// BinaryOperator<T> - 동일한 타입의 매개변수 두 개를 받아 동일한 타입 반환
BinaryOperator<Integer> sum = (a, b) -> a + b;
System.out.println(sum.apply(10, 20)); // 30

// IntUnaryOperator - int 매개변수를 받아 int 반환
IntUnaryOperator square = n -> n * n;
System.out.println(square.applyAsInt(5)); // 25

// IntBinaryOperator - int 매개변수 두 개를 받아 int 반환
IntBinaryOperator multiply = (x, y) -> x * y;
System.out.println(multiply.applyAsInt(4, 5)); // 20

// LongUnaryOperator - long 매개변수를 받아 long 반환
LongUnaryOperator increment = n -> n + 1;
System.out.println(increment.applyAsLong(100L)); // 101

// LongBinaryOperator - long 매개변수 두 개를 받아 long 반환
LongBinaryOperator subtract = (x, y) -> x - y;
```

```
System.out.println(subtract.applyAsLong(100L, 50L)); // 50

// DoubleUnaryOperator - double 매개변수를 받아 double 반환
DoubleUnaryOperator half = n -> n / 2;
System.out.println(half.applyAsDouble(10.0)); // 5.0

// DoubleBinaryOperator - double 매개변수 두 개를 받아 double 반환
DoubleBinaryOperator average = (x, y) -> (x + y) / 2;
System.out.println(average.applyAsDouble(10.0, 20.0)); // 15.0
```

---

5. Predicate 계열

- 매개변수를 받아 boolean 값을 반환하는 인터페이스

인터페이스	메서드	설명
Predicate<T>	boolean test(T t)	객체 T를 매개변수로 받아 boolean 반환
BiPredicate<T,U>	boolean test(T t, U u)	객체 T와 U를 매개변수로 받아 boolean 반환
IntPredicate	boolean test(int value)	int 값을 매개변수로 받아 boolean 반환
LongPredicate	boolean test(long value)	long 값을 매개변수로 받아 boolean 반환
DoublePredicate	boolean test(double value)	double 값을 매개변수로 받아 boolean 반환

```
// Predicate<T> - 객체 T를 매개변수로 받아 boolean 반환
Predicate<String> isLongString = str -> str.length() > 5;
System.out.println(isLongString.test("Hello")); // false
System.out.println(isLongString.test("Hello World")); // true

// BiPredicate<T,U> - 객체 T와 U를 매개변수로 받아 boolean 반환
BiPredicate<String, Integer> isLengthEqual = (str, len) -> str.length() == len;
System.out.println(isLengthEqual.test("Hello", 5)); // true
System.out.println(isLengthEqual.test("World", 3)); // false

// IntPredicate - int 값을 매개변수로 받아 boolean 반환
IntPredicate isEven = num -> num % 2 == 0;
System.out.println(isEven.test(4)); // true
System.out.println(isEven.test(7)); // false

// LongPredicate - long 값을 매개변수로 받아 boolean 반환
LongPredicate isPositive = num -> num > 0;
System.out.println(isPositive.test(100L)); // true
System.out.println(isPositive.test(-50L)); // false

// DoublePredicate - double 값을 매개변수로 받아 boolean 반환
DoublePredicate isGreaterThanPi = num -> num > Math.PI;
System.out.println(isGreaterThanPi.test(4.0)); // true
System.out.println(isGreaterThanPi.test(3.0)); // false
```

---

4. 람다식 사용 시 주의사항

- 람다 표현식 내에서 사용되는 지역변수는 final이거나 effectively final이어야 함

- 너무 복잡한 로직은 일반 메서드로 분리하는 것이 좋음
- 재사용성이 필요한 경우 메서드 참조를 고려

---

5. 메서드 참조

5-1. 메서드 참조란?

- 이미 존재하는 메서드를 람다식으로 표현할 때 사용하는 축약된 표현 방식
- 람다식이 단순히 메서드를 호출하는 경우에 사용하면 코드를 더욱 간결하게 만들 수 있음
-

5-2. 메서드 참조의 유형

유형	문법	예시
정적 메서드 참조	클래스::정적메서드	String::valueOf
인스턴스 메서드 참조	객체::인스턴스메서드	str::toLowerCase
특정 타입의 인스턴스 메서드 참조	클래스::인스턴스메서드	String::length
생성자 참조	클래스::new	ArrayList::new

5-3. 메서드 참조 예시

```
// 1. 정적 메서드 참조
// 람다식: (x) -> Math.abs(x)
// 메서드 참조: Math::abs
Function<Integer, Integer> abs = Math::abs;

// 2. 인스턴스 메서드 참조
String str = "Hello";
// 람다식: () -> str.toString()
// 메서드 참조: str::toString
Supplier<String> supplier = str::toString;

// 3. 특정 타입의 인스턴스 메서드 참조
// 람다식: (String s) -> s.length()
// 메서드 참조: String::length
Function<String, Integer> length = String::length;

// 4. 생성자 참조
// 람다식: () -> new ArrayList<>()
// 메서드 참조: ArrayList::new
Supplier<ArrayList<String>> listSupplier = ArrayList::new;
```

5-4. 메서드 참조의 장점

- 코드의 가독성 향상
- 중복 코드 제거
- 기존 메서드 재사용성 증가
- 유지보수가 용이
-

5-5. 메서드 참조 사용 시 주의사항

- 메서드의 매개변수 타입과 반환 타입이 람다식의 시그니처(선언부)와 일치해야 함
- 모든 람다식을 메서드 참조로 변환할 수 있는 것은 아님
- 메서드 참조가 오히려 가독성을 해치는 경우에는 일반 람다식을 사용하는 것이 좋음