람다식(Lambda Expressions)

람다식은 익명함수(anonymous function)를 생성하기 위한 식으로 객체 지향 언어보다는 함수 지향에 가깝다. 자바 8 에서 람다식을 수용한 이유는 자바 코드가 매우 간결해지고, 컬렉션의 요소를 필터링하거나 매핑해서 원하는 결과를 쉽게 집계할 수 있기 때문이다. 람다식의 형태는 매개 변수를 가진 코드 블록이지만, 런타임 시에 익명 구현 객체를 생성한다.

람다식 -> 매개 변수를 가진 코드 블록 -> 익명 구현 개체
예를 들어 Runnable 인터페이스의 익명 구현 객체를 생성하는 코드

```
Runnable runnable = new Runnable(){
        public void run() { ... }
};
```

위 코드에서 익명 구현 객체를 람다식으로 표현하면 다음과 같다.

```
Runnable runnable = () -> { ... };
```

람다식은 **"(매개 변수)->{실행코드}"** 형태로 작성한다.

1. 기본 문법

함수적 스타일의 람다식 작성방법

(매개 변수, ...)->{ 실행문; ... }

int 형 매개변수 a 의 값을 콘솔에 출력하기 위한 람다식

```
(int a) -> { System.out.println(a); }
```

매개변수 타입은 런타임 시에 대입되는 값에 따라 자동으로 인식되므로 람다식에서는 매개변수의 타입을 생략할 수 있다.

```
(a) -> { System.out.println(a); }
```

하나의 매개변수만 있다면 ()를 생략할 수 있다.

```
a -> { System.out.println(a); }
```

만약 매개변수가 없다면 빈괄호를 반드시 사용해야 한다.

```
() -> { System.out.println("람다"); }
```

결과값을 반환해야 한다면 return 문으로 결과값을 지정할 수 있다.

```
(x, y) -> { return x+y; }
```

{ }에 return 문만 있다면 다음과 같이 작성할 수 있다.

```
(x, y) -> x+y
```

2. 타겟 타입과 함수적 인터페이스

람다식의 형태는 매개변수를 가진 코드 블록이기 때문에 마치 자바의 메소드를 선언하는 것처럼 보인다. 자바는 메소드를 단독으로 사용할 수 없고 클래스의 구성 멤버로 선언하기 때문에 람다식은 단순히 메소드를 선언하는 것이 아니라 이 메소드를 가지고 있는 객체를 생성한다.

인터페이스명 변수 = 람다식;

람다식은 인터페이스 변수에 대입된다. 이 말은 인터페이스의 익명 객체를 생성한다는 뜻이다. 인터페이스는 직접 객체화할 수 없기 때문에 구현 클래스가 필요하다. 람다식은 익명 구현 클래스를 생성하고 객체화 한다. 람다식은 대입될 인터페이스의 종류에 따라 작성 방법이 달라지기 때문에 람다식이 대입될 입터페이스를 람다식의 타겟 타입(target type)이라고 한다.

(1) 함수적 인터페이스(@FunctionalInterface)

람다식이 하나의 메소드를 정의하기 때문에 두 개 이상의 추상 메소드가 선언된 인터페이스는 람다식을 이용해서 구현 객체를 생성할 수 없다. 하나의 추상 메소드가 선언된 인터페이스만 람다식의 타겟 타입이 될 수 있는데, 이러한 인터페이스를 함수적 인터페이스(functional interface)라고 한다.

두 개 이상의 추상 메소드가 선언되지 않도록 인터페이스 선언 시 @FunctionalInterface 어노테이션을 붙인다.

```
@FunctionalInterface
public interface MyFunctionalInterface {
        public void method();
        public void otherMethod(); // 컴파일 오류
}
```

(2) 매개변수와 리턴값이 없는 람다식

매개변수와 리턴값이 없는 추상 메소드를 가진 함수적 인터페이스

MyFunctionalInterface.java

```
@FunctionalInterface
public interface MyFunctionalInterface {
        public void method();
}
```

람다식 선언과 호출

```
MyFunctionalInterface fi = () -> { ... }
fi.method();
```

MyFunctionalInterfaceEx.java

```
public class MyFunctionalInterfaceEx {
	public static void main(String[] args) {
		MyFunctionalInterface fi;

		fi = () -> {
			String str = "메소드 호출";
			System.out.println(str);
		};
		fi.method();

		fi = () -> {
			System.out.println("두번째 메소드 호출");
		};
		fi.method();

		fi = () -> {
			System.out.println("세번째 메소드 호출");
		};
		fi.method();
	}
}
```

```
<terminated> MyF
메소드 호출
두번째 메소드 호출
세번째 메소드 호출
```

(3) 매개변수가 있는 람다식

MyFunctionalInterface.java

```
@FunctionalInterface
public interface MyFunctionalInterface {
	public void method(int x);
}
```

MyFunctionalInterfaceEx.java

```
public class MyFunctionalInterfaceEx {
	public static void main(String[] args) {
		MyFunctionalInterface fi;

		fi = (x) -> {
			int number = x * 5;
```

```
                        System.out.println(number);
                };
                fi.method(3);

                fi = (x) -> {
                        System.out.println("결과 : " + x * 5);
                };
                fi.method(4);

                fi = x -> System.out.println("결과 : " + x * 7);
                fi.method(5);
        }
}
```

```
<terminated> M
15
결과 : 20
결과 : 25
```

(4) 반환 값이 있는 람다식

MyFunctionalInterface.java

```
@FunctionalInterface
public interface MyFunctionalInterface {
        public int method(int x, int y);
}
```

MyFunctionalInterfaceEx.java

```
public class MyFunctionalInterfaceEx {
        public static void main(String[] args) {
                MyFunctionalInterface fi;

                fi = (x, y) -> {
                        int res = x + y;
                        return res;
                };
                System.out.println(fi.method(3, 4));

                fi = (x, y) -> {
                        return x * y;
                };
                System.out.println("결과 : " + fi.method(5, 6));
```

```
                fi = (x, y) -> x + y;
                System.out.println(fi.method(7, 8));

                fi = (x, y) -> sum(x, y);
                // fi = (x, y) -> { return sum(x, y) };
                System.out.println(fi.method(7, 7));
        }
        private static int sum(int x, int y) {
                return (x * y);
        }
}
```

```
<terminated>
7
결과 : 30
15
49
```

3. 클래스 멤버와 로컬 변수의 사용

(1) 클래스 멤버 사용

람다식 실행 블록에는 클래스의 멤버인 필드와 메소드를 제약 없이 사용할 수 있다. 하지만 this 키워드를 사용할 때에는 주의해야 한다. **람다식에서 this 는 내부적으로 생성되는 익명 객체의 참조가 아니라 람다식을 실행한 객체의 참조이다.**

람다식에서 외부 객체와 중첩 객체의 참조를 얻어 필드 값을 출력하는 예이다. 중첩 객체 Inner 에서 람다식을 실행했기 때문에 람다식 내부에서의 this 는 중첩 객체이다.

MyFunctionalInterface.java

```
@FunctionalInterface
public interface MyFunctionalInterface {
        public void method();
}
```

UsingThis.java

```
public class UsingThis {
        public int outterField = 10;
        class Inner {
                int innerField = 20;
                void method() {
                        MyFunctionalInterface fi = () -> {
                                System.out.println("외부 변수 : " + outterField);
                                System.out.println("outterField : " + UsingThis.this.outterField + "₩n");

                                System.out.println("내부 변수 : " + innerField);
```

```
                                System.out.println("innerField : " + this.innerField + "₩n");
                        };
                        fi.method();
                }
        }
}
```

외부 객체의 참조를 얻기 위해서는 클래스명.this 를 사용한다. 람다식 내부에서는 this 는 Inner 객체르르 참조한다.

UsingThisExample.java

```
public class UsingThisExample {
        public static void main(String ... args) {
                // TODO Auto-generated method stub
                UsingThis usingThis = new UsingThis();
                UsingThis.Inner inner = usingThis.new Inner();
                inner.method();
        }
}
```

```
<terminated> UsingThisE
외부 변수 : 10
outterField : 10

내부 변수 : 20
innerField : 20
```

(2) 로컬 변수의 사용

람다식은 메소드 내부에서 주로 작성되기 때문에 로컬 익명 구현 객체를 생성시킨다. 외부 클래스의 필드나 메소드는 제한 없이 사용할 수 있으나, 메소드의 매개변수 또는 로컬변수를 사용하면 이 두 변수는 final 특서을 가져야 한다. 따라서 **매개변수 또는 로컬변수를 람다식에서 읽는 것은 허용되지만, 람다식 내부 또는 외부에서 변경할 수 없다.**

MyFunctionalInterface.java

```
@FunctionalInterface
public interface MyFunctionalInterface {
        public void method();
}
```

UsingLocalVariable.java

```
public class UsingLocalVariable {
        void method(int arg){       // arg 는 final 특성을 가진다.
                int localVal = 40;      // localVar 은 final 특서을 가진다.

                // arg = 20;          수정 불가
                // localVar = 30;   수정 불가
```

```
                MyFunctionalInterface fi = () -> {
                        //로컬 변수 읽기
                        System.out.println("arg : " + arg);
                        System.out.println("localVar : " + localVal + "₩n");
                };
                fi.method();
        }
}
```

UsingLocalVariableEx.java

```
public class UsingLocalVariableEx {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                UsingLocalVariable ulv = new UsingLocalVariable();
                ulv.method(10);
        }
}
```

```
<terminated> Usinc
arg : 10
localVar : 40
```

4. 표준 API 의 함수적 인터페이스

자바에서 제공되는 표준 API 에서 한 개의 추상 메소드를 가지는 인터페이스들은 모두 람다식을 이용해서 익명 객체로 표현이 가능하다. 예를 들어 스레드의 작업을 정의하는 Runnable 인터페이스는 매개 변수의 리턴 값이 없는 run() 메소드만 존재하기 때문에 다음과 같이 람다식을 이용해서 Runnable 인스턴스를 생성시킬 수 있다.

RunnableEx.java

```
public class RunnableEx {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Runnable ra = () -> {
                        for(int i=0; i<10; i++){
                                System.out.println(i);
                        }
                };
                Thread th = new Thread(ra);
                th.start();
        }
}
```

```
<terminated>
0
1
2
3
4
5
6
7
8
9
```

Thread 생성자를 호출할 때 다음과 같이 람다식을 매개값으로 대입해도 된다.

```
Thread th = new Thread(() -> {
        for(int i=0; i<10; i++){
                System.out.println(i);
        }
});
```

자바 8 부터는 빈번하게 사용되는 함수적 인터페이스는 java.util.function 표준 API 패키지로 제공된다. 이 패키지에서 제공하는 함수적 인터페이스의 목적은 메소드 또는 생성자의 매개 타입으로 사용되어 람다식을 대입할 수 있도록 하기 위해서이다. 자바 8 부터 추가되거나 변경된 API 에서 이 함수적 인터페이스들을 매개 타입으로 많이 사용한다. java.util.function 패키지의 함수적 인터페이스는 크게 Consumer, Supplier, Function, Operator, Predicate 로 구분된다. 구분 기준은 인터페이스에 선언된 추상 메소드의 매개값과 반환값의 유무이다.

종류	추상 메소드의 특징
Consumer	매개값이 있고 반환값이 없다.
Supplier	매개값이 없고 반환값이 있다.
Function	매개값이 있고 반환값이 있다. 주로 매개값을 반환값으로 매핑(타입 변환)
Operator	매개값이 있고 반환값이 없다. 주로 매개값을 연산하고 결과를 반환
Predicate	매개값이 있고 반환 타입은 boolean, 매개값을 조사해서 true/false 를 반환

(1) Consumer 함수적 인터페이스

Consumer 함수적 인터페이스의 특징은 반환값이 없는 accept() 메소드를 가지고 있다. accept() 메소드는 단지 매개값을 사용만 할 뿐 반환값이 없다.

매개변수의 타입과 수에 따라서 다음과 같은 Consumer 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Consumer<T>	void accept(T t)	객체 T 를 받아 사용
BiConsumer<T, U>	void accept(T t, U u)	객체 T 를 받아 사용
DoubleConsumer	void accept(double value)	double 값을 받아 사용
IntConsumer	void accept(int value)	int 값을 받아 사용
LongConsumer	void accept(long value)	long 값을 받아 사용
ObjDoubleConsumer<T>	void accept(T t, double value)	객체 T 와 double 값을 받아 사용
ObjIntConsumer<T>	void accept(T t, int value)	객체 T 와 int 값을 받아 사용
ObjLongConsumer<T>	void accept(T t, long value)	객체 T 와 long 값을 받아 사용

Consumer<T>인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. accept() 메소드는 매개값으로 T 객체 하나를 가지므로 람다식도 한 개의 매개변수를 사용한다.

```
Consumer<String> consumer = t -> { t 를 사용하는 실행문; }
```

BiConsumer<T, U>인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. accept() 메소드는 매개값으로 T 와 U 두 개의 객체를 가지므로 람다식도 두 개의 매개변수를 사용한다.

```
BiConsumer<String, String> consumer = (t, u) -> { t 와 u 를 사용하는 실행문; }
```

DoubleConsumer 인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. accept() 메소드는 매개값으로 double 하나를 가지므로 람다식도 한 개의 매개변수를 사용한다.

```
DoubleConsumer consumer = d -> { d 를 사용하는 실행문; }
```

ObjConsumer<T>인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. accept() 메소드는 매개값으로 T 객체와 int 값 두 개를 가지므로 람다식도 두 개의 매개변수를 사용한다.

```
ObjIntConsumer<String> consumer = (t, i) -> { t 와 i 를 사용하는 실행문; }
```

ConsumerEx.java

```
import java.util.function.BiConsumer;
import java.util.function.Consumer;
import java.util.function.DoubleConsumer;
import java.util.function.ObjIntConsumer;

public class ConsumerEx {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Consumer<String> consumer = t -> System.out.println(t + "8");
		consumer.accept("java");

		BiConsumer<String, String> bigConsumer = (t, u) -> System.out.println(t + u);
		bigConsumer.accept("Java", "8");

		DoubleConsumer doubleConsumer = d -> System.out.println("Java" + d);
		doubleConsumer.accept(8.0);

		ObjIntConsumer<String> objIntConsumer = (t, i) -> System.out.println(t + i);
```

```
<terminated>
java8
Java8
Java8.0
Java8
```

```
                objIntConsumer.accept("Java", 8);
        }
}
```

(2) Supplier 함수적 인터페이스

Supplier 함수적 인터페스의 특징은 매개변수가 없고 반환값이 있는 getXXX() 메소드를 가지고 있다. 이 메소드들은 실행 후 호출한 곳으로 데이터를 반환하는 역할을 한다.

반환값을 반환값으로 매핑(타입전환)하는 역할을 한다.

매개변수의 타입과 수에 따라서 다음과 같은 Suppier 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Supplier	T get()	T 객체를 반환
BooleanSupplier	boolean getAsBoolean()	Boolean 값을 반환
DoubleSupplier	double getAsDouble()	double 값을 반환
IntSupplier	int getAsInt()	int 값을 반환
LongSupplier	long getAsLong()	long 값을 반환

Supplier<T>인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. get() 메소드가 매개값을 가지지 않으므로 람다식도 ()를 사용한다. 람다식의 { }는 반드시 한 개의 T 객체를 반환해야 한다.

```
Supplier<String> supplier = ( ) -> { ... ; return "문자열"; }
```

IntSupplier 인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. getAsInt() 메소드가 매개값을 가지지 않으므로 람다식도 ()를 사용한다. 람다식의 { }는 반드시 int 값를 반환해야 한다.

```
IntSupplier<String> supplier = ( ) -> { ... ; return int 값; }
```

주사위의 숫자를 랜덤하게 공급하는 IntSupplier 인터페이스를 타겟 타입으로 하는 람다식

SupplierEx.java

```
import java.util.function.IntSupplier;

public class SupplierEx {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                IntSupplier intSupplier = () -> {
                        int num = (int) (Math.random() * 6) + 1;
```

```
                    return num;
                };
                int num = intSupplier.getAsInt();
                System.out.println("눈의 수: " + num);
        }
}
```

(3) Function 함수적 인터페이스

Function 함수적 인터페이스의 특징은 매개값과 반환값이 있는 applyXXX() 메소드를 가지고 있다. 이 메소드들은 매개값을 반환값으로 매핑(타입 변환)하는 역할을 한다.

매개변수 타입과 반환 타입에 따라서 다음과 같은 Function 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Function<T, R>	R apply(T t)	객체 T 를 객체 R 로 매핑
BiFunction <T, U, R>	R apply(T t, U u)	객체 T 와 U 를 객체 R 로 매핑
DoubleFunction<R>	R apply(double value)	double 을 객체 R 로 매핑
IntFunction<R>	R apply(int value)	int 를 객체 R 로 매핑
IntToDoubleFunction	double applyAsDouble(int value)	int 를 double 로 매핑
IntToLongFunction	int applyAsLong(int value)	int 를 long 으로 매핑
LongToDoubleFunction	double applyAsDouble(long value)	long 를 double 로 매핑
LongToIntFunction	int applyAsInt(long value)	long 를 int 로 매핑
ToDoubleBiFunction<T, U>	double applyAsDouble(T t, U u)	객체 T 와 U 를 double 로 매핑
ToDoubleFunction<T>	double applyAsDouble(T value)	객체 T 를 double 로 매핑
ToIntBiFunction<T, U>	int applyAsInt(T t, U u)	객체 T 와 U 를 int 로 매핑
ToIntFunction<T>	int applyAsInt(T value)	객체 T 를 int 로 매핑
ToLongBiFunction<T, U>	long applyAsLong(T t, U u)	객체 T 와 U 를 long 으로 매핑
ToLongFunction<T>	long applyAsLong(T value)	객체 T 를 long 으로 매핑

Function<T, R> 인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. apply() 메소드가 매개값으로 T 객체 하나를 가지므로 람다식도 한 개의 매개변수를 사용한다. 그리고 apply()메소드의 반환타입이 R 이므로 람다식의 { }는 반환값은 R 객체가 된다.

```
Function<Student, String> function = t -> { return t.getNmae(); }
또는
Function<Student, String> function = t -> t.getNmae();
```

T 가 Student 타입이고 R 이 String 타입이므로 t 매개변수 타입은 Strudent 가 되고 람다식의 { }에서는 String 을 반환해야 한다.

ToIntFunction<T> 인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. applyAsInt() 메소드가 매개값으로 T 객체 하나를 가지므로 람다식도 한 개의 매개변수를 사용한다. 그리고 applyAsInt()메소드의 반환타입이 int 이므로 람다식의 { }는 반환값은 int 가 된다.

```
ToIntFunction<Student> function = t -> { return t.getScore(); }
또는
ToIntFunction<Student> function = t -> t.getScore();
```

T 가 Student 타입이므로 t 매개변수 타입은 Strudent 가 된다.

List 에 저장된 학생 객체를 하나씩 꺼내서 이름과 점수를 출력한다.

Student.java

```
public class Student {
	private String name;
	private int englishScore;
	private int mathScore;

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

	public String getName() { return name; }
	public int getEnglishScore() { return englishScore; }
	public int getMathScore() { return mathScore; }
}
```

FunctionEx1.java

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.function.ToIntFunction;

public class FunctionEx1 {
	private static List<Student> list = Arrays.asList(
				new Student("유성룡", 99, 95), new Student("이순신", 90, 91));

	public static void printString(Function<Student, String> function) {
		for (Student student : list) {
```

```
                        System.out.print(function.apply(student) + " ");
                }
                System.out.println();
        }

        public static void printInt(ToIntFunction<Student> function) {
                for (Student student : list) {
                        System.out.print(function.applyAsInt(student) + " ");
                }
                System.out.println();
        }

        public static void main(String[] args) {
                System.out.println("[학생 이름]");
                printString(t -> t.getName());

                System.out.println("[영어 점수]");
                printInt(t -> t.getEnglishScore());

                System.out.println("[수학 점수]");
                printInt(t -> t.getMathScore());
        }
}
```

```
<terminated> Fun
[학생 이름]
유성룡 이순신
[영어 점수]
99 90
[수학 점수]
95 91
```

FunctionEx2.java

```
import java.util.Arrays;
import java.util.List;
import java.util.function.ToIntFunction;

public class FunctionEx2 {
        private static List<Student> list = Arrays.asList(
                        new Student("유성룡", 99, 95), new Student("이순신", 90, 91));

        public static double avg(ToIntFunction<Student> function) {
                int sum = 0;
                for (Student student : list) {
                        sum += function.applyAsInt(student);
                }
                double avg = (double) sum / list.size();
```

```
                return avg;
        }

        public static void main(String[] args) {
                double englishAvg = avg(s -> s.getEnglishScore());
                System.out.println("영어 평균 점수: " + englishAvg);

                double mathAvg = avg(s -> s.getMathScore());
                System.out.println("수학 평균 점수: " + mathAvg);
        }
}
```

```
<terminated> FunctionEx2
영어 평균 점수: 94.5
수학 평균 점수: 93.0
```

(4) Operator 함수적 인터페이스

Operator 함수적 인터페이스는 Function 과 동일하게 매개변수와 반환값이 있는 applyXXX() 메소드를 가지고 있다. 하지만 이 메소드들은 매개값을 반환값으로 매핑(타입 변환)하는 역할보다는 매개값을 이용해서 연산을 수행한 후 동일한 타입으로 반환값을 제공하는 역할을 한다.

매개변수의 타입과 수에 따라서 다음과 같은 Operator 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
BinaryOperator<T>	BiFunction <T, U, R>의 하위 인터페이스	T 와 U 를 연산한 후 R 반환
UnaryOperator<T>	Function <T, R>의 하위 인터페이스	T 를 연산한 후 R 반환
DoubleBinaryOperator	double applyAsDouble(double, double)	두 개의 double 연산
DoubleUnaryOperator	double applyAsDouble(double)	한 개의 double 연산
IntBinaryOperator	int applyAsInt(int, int)	두 개의 int 연산
IntUnaryOperator	int applyAsInt(int)	한 개의 int 연산
LongBinaryOperator	long applyAsLong(long, long)	두 개의 long 연산
LongUnaryOperator	long applyAsLong(long)	한 개의 long 연산

IntBinaryOperator 인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. applyAsInt() 메소드가 매개값으로 두 개의 int 를 가지므로 람다식도 두 개의 int 매개변수 a 와 b 를 사용한다. 그리고 applyAsInt() 메소드의 반환 타입이 int 이므로 람다식의 { }는 반환값은 int 가 된다.

```
IntBinaryOperator operator= (a, b) -> { ...; return int 값; }
```

int[ ] 배열에서 최대값과 최소값을 얻는다.

OperatorEx.java

```
import java.util.function.IntBinaryOperator;
```

```
public class OperatorEx {
	private static int[] scores = { 92, 95, 87 };

	public static int maxOrMin(IntBinaryOperator operator) {
		int result = scores[0];
		for (int score : scores) {
			result = operator.applyAsInt(result, score);
		}
		return result;
	}

	public static void main(String[] args) {
		// 최대값 얻기
		int max = maxOrMin((a, b) -> {
			if (a >= b)
				return a;
			else
				return b;
		});
		System.out.println("최대값: " + max);

		// 최소값 얻기
		int min = maxOrMin((a, b) -> {
			if (a <= b)
				return a;
			else
				return b;
		});
		System.out.println("최소값: " + min);
	}
}
```

```
<terminated> Op
최대값: 95
최소값: 87
```

(5) Predicate 함수적 인터페이스

Predicate 함수적 인터페이스는 매개변수와 boolean 반환값이 있는 testXXX() 메소드를 가지고 있다. 이 메소드들은 매개값을 조사해서 true 또는 false 를 반환하는 역할을 한다.

매개변수의 타입과 수에 따라서 다음과 같은 Predicate 함수적 인터페이스들이 있다.

인터페이스명	추상 메소드	설 명
Predicate<T>	boolean test(T t)	객체 T 를 조사
BiPredicate<T, U>	boolean test(T t, U u)	객체 T 와 U 를 비교 조사
DoublePredicate	boolean test(double value)	double 값을 조사
IntPredicate	boolean test(int value)	int 값을 조사
LongPredicate	boolean test(long value)	long 값을 조사

Predicate<T> 인터페이스를 타겟 타입으로 하는 람다식은 다음과 같이 작성할 수 있다. test() 메소드가 매개값으로 T 객체 하나를 가지므로 람다식도 한 개의 매개변수를 사용한다. 그리고 test() 메소드의 반환 타입이 boolean 이므로 람다식의 { }는 반환값은 boolean 이 된다.

```
Predicate<Student1> predicate = t -> { return t.getSex().equal("남자"); }
또는
Predicate<Student1> predicate = t -> t.getSex().equal("남자");
```

T 가 Student1 타입이므로 t 매개변수 타입은 Strudent1 가 된다.

List 에 저장된 남자 또는 여자 학생들의 평균 점수를 출력한다.

Student1.java

```
public class Student1 {
	private String name;
	private String sex;
	private int score;

	public Student1(String name, String sex, int score) {
		this.name = name;
		this.sex = sex;
		this.score = score;
	}

	public String getSex() { return sex; }
	public int getScore() { return score; }
}
```

PredicateEx.java

```
import java.util.Arrays;
```

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

public class PredicateEx {
	private static List<Student1> list = Arrays.asList(
			new Student1("유성룡", "남자", 90),
			new Student1("이순신", "여자", 90),
			new Student1("감자바", "남자", 95),
			new Student1("박한나", "여자", 92)
	);

	public static double avg(Predicate<Student1> predicate) {
		int count = 0, sum = 0;
		for(Student1 student : list) {
			if(predicate.test(student)) {
				count++;
				sum += student.getScore();
			}
		}
		return (double) sum / count;
	}

	public static void main(String[] args) {
		double maleAvg = avg( t -> t.getSex().equals("남자") );
		System.out.println("남자 평균 점수: " + maleAvg);

		double femaleAvg = avg( t -> t.getSex().equals("여자") );
		System.out.println("여자 평균 점수: " + femaleAvg);
	}
}
```

```
<terminated> PredicateEx |
남자 평균 점수: 92.5
여자 평균 점수: 91.0
```