

```
for object to mirror_mod.mirror_object
operation == "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False
operation == "MIRROR_Y":
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True
```

```
@selection at the end -add
mirror_ob.select= 1
modifier_ob.select=1
context.scene.objects.active
("Selected" + str(modifier_ob.name))
mirror_ob.select = 0
= bpy.context.selected_object
data.objects[one.name].select
print("please select exactly one mirror")
```

```
-- OPERATOR CLASSES -----
types.Operator):
X mirror to the selected
object.mirror_mirror_x"
mirror X"
```

Java 기초

람다식

람다식

- 람다식이란?
 - 객체 지향 프로그래밍과 함수적 프로그래밍을 혼합하는 방법이 대세가 되면서 떠오르기 시작.
 - 자바8 부터 람다식을 지원하기 시작.
 - 수학자 알론소 처치가 발표한 람다 계산법에서 사용된 식으로 존 메카시가 프로그래밍 언어에 도입
 - 자바에서 람다식을 수용한 이유는 자바 코드가 매우 간결해지고, 컬렉션의 요소를 필터링하거나 매핑해서 원하는 결과를 쉽게 집계할 수 있음

```
Runnable runnable = new Runnable() {    //익명 구현 객체
    public void run() { ... }
}
```



```
Runnable runnable = () -> { ... }    // 람다식
```

람다식

- 람다식이란?
 - 기본 문법

(타입 매개변수, ...) -> {실행문;...}

() -> {실행문;...}

- 매개 변수가 없다면 람다식에서 반드시 빈 괄호를 사용한다.

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

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

```
a -> System.out.println(a)    // 단일 매개 변수, 단일 실행문 시 {}, {} 제거
```

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

```
(x, y) -> x + y    // return 문만 있을 경우 return 문장 생략 가능
```

타겟 타입과 함수적 인터페이스

- 타겟 타입과 함수적 인터페이스

- 자바는 메소드를 단독으로 선언할 수 없고 항상 클래스의 구성 멤버로 선언하기 때문에 람다식은 단순히 메소드를 선언하는 것이 아니라 가지고 있는 객체를 생성해 낸다.

인터페이스 변수 = 람다식;

- 람다식은 대입될 인터페이스의 종류에 따라 작성방법이 달라지기 때문에 람다식이 대입될 인터페이스를 람다식의 타겟 타입(target type) 이라고 한다.
- 하나의 추상 메소드를 가진 인터페이스 만이 람다식의 타겟 타입이 될 수 있는데, 이러한 인터페이스를 함수적 인터페이스라고 한다.

```
@FunctionalInterface
public interface MyFunctionInterface {
    public void method();
    //public void otherMethod(); //메러
}
```

타겟 타입과 함수적 인터페이스

- 매개변수와 리턴 값이 없는 람다식

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

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

        fi = () -> {
            String str = "method call1";
            System.out.println(str);
        };
        fi.method();

        fi = () -> { System.out.println("method call2"); };
        fi.method();

        fi = () -> System.out.println("method call3");
        fi.method();
    }
}
```

타겟 타입과 함수적 인터페이스

- 매개변수가 있는 람다식

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

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

        fi = (x) -> {
            int result = x * 5;
            System.out.println(result);
        };
        fi.method(2);

        fi = (x) -> { System.out.println(x*5); };
        fi.method(2);

        fi = x -> System.out.println(x*5);
        fi.method(2);
    }
}
```

타겟 타입과 함수적 인터페이스

- 리턴값이 있는 람다식

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

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

        fi = (x, y) -> {
            int result = x + y;
            return result;
        };
        System.out.println(fi.method(2, 5));

        fi = (x, y) -> { return x + y; };
        System.out.println(fi.method(2, 5));

        fi = (x, y) -> x + y;
        System.out.println(fi.method(2, 5));

        fi = (x, y) -> sum(x, y);
        System.out.println(fi.method(2, 5));
    }

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

타겟 타입과 함수적 인터페이스

- 클래스의 멤버 사용

```
public interface MyFunctionalInterface {  
    public void method();  
}  
  
public class UsingThis {  
    public int outterField = 10;  
  
    class Inner {  
        int innerField = 20;  
  
        void method() {  
            //람다식  
            MyFunctionalInterface fi= () -> {  
                System.out.println("outterField: " + outterField);  
                System.out.println("outterField: " + UsingThis.this.outterField +  
  
                System.out.println("innerField: " + innerField);  
                System.out.println("innerField: " + this.innerField + "\n");  
            };  
            fi.method();  
        }  
    }  
}
```

바깥 객체의 참조를 얻기 위해서는 클래스명.this를 사용

람다식 내부에서 this는 Inner 객체를 참조

타겟 타입과 함수적 인터페이스

- 클래스의 멤버 사용

```
public class UsingThisExample {  
    public static void main(String... args) {  
        UsingThis usingThis = new UsingThis();  
        UsingThis.Inner inner = usingThis.new Inner();  
        inner.method();  
    }  
}
```

outterField: 10

outterField: 10

innerField: 20

innerField: 20

타겟 타입과 함수적 인터페이스

- 로컬 변수 사용

```
arg: 20  
localVar: 40
```

```
public interface MyFunctionalInterface {  
    public void method();  
}  
  
public class UsingLocalVariable {  
    void method(int arg) { //arg는 final 특성을 가짐  
        int localVar = 40; //localVar는 final 특성을 가짐  
  
        //arg = 31;          //final 특성 때문에 수정 불가  
        //localVar = 41;     //final 특성 때문에 수정 불가  
  
        //람다식  
        MyFunctionalInterface fi = () -> {  
            //로컬변수 사용  
            System.out.println("arg: " + arg);  
            System.out.println("localVar: " + localVar + "\n");  
        };  
        fi.method();  
    }  
}  
  
public class UsingLocalVariableExample {  
    public static void main(String... args) {  
        UsingLocalVariable ulv = new UsingLocalVariable();  
        ulv.method(20);  
    }  
}
```

표준 API의 함수적 인터페이스

- 표준 API 함수적 인터페이스

- 자바에서 제공되는 표준 API에서 한 개의 추상 메소드를 가지는 인터페이스들은 모두 람다식을 이용해서 익명 구현 객체로 표현이 가능하다.
- 대표적인 예제로는 Runnable과 Thread가 존재한다.
- 자바 8부터 빈번하게 사용되는 함수적 인터페이스는 java.util.function 표준 API 패키지로 제공한다.
- java.util.function 패키지의 함수적 인터페이스는 크게 Consumer, Supplier, Function, Operator, Predicate로 구분 된다. 구분 기준은 인터페이스에 선언된 추상 메소드의 매개 값과 리턴 값 유무이다.

종류	추상메소드 특징
Consumer	- 매개값은 있고, 리턴값은 없음
Supplier	- 매개값은 없고, 리턴값은 있음
Function	- 매개값도 있고, 리턴값도 있음 - 주로 매개값을 리턴값으로 매핑(타입변환)
Operator	- 매개값도 있고, 리턴값도 있음 - 주로 매개값을 연산하고 결과를 리턴
Predicate	- 매개값은 있고, 리턴타입은 boolean - 매개값을 조사해서 true/false를 리턴

표준 API의 함수적 인터페이스

- Consumer 함수는 리턴값이 없는 accept() 메소드를 가지고 있다.
- 매개변수의 타입과 수에 따라서 아래와 같은 Consumer들이 있다

인터페이스명	추상 메소드	설명
Consumer<T>	void accept(T t)	객체 T를 받아 소비
BiConsumer<T,U>	void accept(T t, U u)	객체 T와 U를 받아 소비
DoubleConsumer	void accept(double value)	double 값을 받아 소비
IntConsumer	void accept(int value)	int 값을 받아 소비
LongConsumer	void accept(long value)	long 값을 받아 소비
ObjDoubleConsumer<T>	void accept(T t, double value)	객체 T와 Double값을 받아 소비
ObjIntConsumer<T>	void accept(T t, int value)	객체 T와 Int 값을 받아 소비
ObjLongConsumer<T>	void accept(T t, long value)	객체 T와 Long 값을 받아 소비

표준 API의 함수적 인터페이스

- Consumer Example

```
public class ConsumerExample {  
    public static void main(String[] args) {  
        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);  
        objIntConsumer.accept("Java", 8);  
    }  
}
```

표준 API의 함수적 인터페이스

- Supplier 함수는 매개변수가 없고 리턴값이 있는 getXXX() 메소드를 가지고 있다.
- 이 메소드들은 실행 후 호출한 곳으로 데이터를 리턴하는 역할을 한다..

인터페이스명	추상 메소드	설명
Supplier<T>	T get()	T 객체를 리턴
BooleanSupplier	boolean getAsBoolean()	boolean 값을 리턴
DoubleSupplier	double getAsDouble()	double 값을 리턴
IntSupplier	int getAsInt()	int 값을 리턴
LongSupplier	long getAsLong()	long 값을 리턴

표준 API의 함수적 인터페이스

- Supplier Examples

```
public class SupplierExample {  
    public static void main(String[] args) {  
        IntSupplier intSupplier = () -> {  
            int num = (int) (Math.random() * 6) + 1;  
            return num;  
        };  
  
        int num = intSupplier.getAsInt();  
        System.out.println("눈의 수: " + num);  
    }  
}
```

표준 API의 함수적 인터페이스

- Function 함수는 매개변수와 리턴값이 있는 getXXX() 메소드를 가지고 있다.
- 이 메소드들은 매개값을 리턴값으로 매핑하는 역할을 한다.

인터페이스명	추상 메소드	설명
Function<T,R>	R apply(T t)	객체 T를 객체 R로 매핑
BiFunction<T,U,R>	R apply<T t, U u>	객체 T와 U를 객체 R로 매핑
DoubleFunction<R>	R apply<double value>	double을 객체 R로 매핑
IntFunction<R>	R apply(Int value)	int를 객체 R로 매핑
IntToDoubleFunction	double applyAsDouble(int value)	int를 double로 매핑
IntToLongFunction	long applyAsLong(int long)	int를 long으로 매핑
LongToDoubleFunction	double applyAsDouble(long value)	long을 double로 매핑
LongToIntFunction	int applyAsInt(long value)	long을 int로 매핑
ToDoubleBiFunction<T,U>	double applyAsDouble(T t, U u)	객체 T와 U를 double로 매핑
ToDoubleFunction<T>	double applyAsDouble(T t)	객체 T를 double로 매핑

표준 API의 함수적 인터페이스

- Function 함수는 매개변수와 리턴값이 있는 getXXX() 메소드를 가지고 있다.
- 이 메소드들은 매개값을 리턴값으로 매핑하는 역할을 한다.

인터페이스명	추상 메소드	설명
ToIntBiFunction<T>	int applyAsInt(T t, U u)	객체 T와 U를 int로 매핑
ToIntFunction<T>	int applyAsInt(T t)	객체 T를 int로 매핑
ToLongBiFunction<T,U>	long applyAsLong(T t,U u)	객체 T와 U를 long으로 매핑
ToLongFunction<T>	long applyAsLong(T t)	객체 T를 long으로 매핑

표준 API의 함수적 인터페이스

- Function 예제 - 1

```
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; }  
}
```

```
public class FunctionExample1 {  
    private static List<Student> list = Arrays.asList(  
        new Student("홍길동", 90, 96),  
        new Student("신용권", 95, 93)  
    );  
  
    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() );  
    }  
}
```

표준 API의 함수적 인터페이스

- Function 예제 - 2

```
public class FunctionExample2 {  
    private static List<Student> list = Arrays.asList(  
        new Student("홍길동", 90, 96),  
        new Student("신용권", 95, 93)  
    );  
  
    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> FunctionExample2 [Java A

영어 평균 점수: 92.5

수학 평균 점수: 94.5

표준 API의 함수적 인터페이스

- Operator 함수적 인터페이스는 매개 변수와 리턴값이 있는 applyXXX() 메소드를 가지고 있다.
- 이 메소드들은 매개값을 리턴값으로 매핑하는 역할보다는 매개값을 이용해 연산을 수행한 후 동일한 타입으로 리턴값을 제공하는 역할을 한다.

인터페이스명	추상 메소드	설명
BinaryOperator<T>	BiFunction<T,R,U>	T와 U을 연산 후 R을 리턴
UnaryOperator<T>	Function<T,T>의 하위 인터페이스	T를 연산한 후 T리턴
DoubleBinaryOperator	double applyAsDouble(double, double)	두개의 double 연산
DoubleUnaryOperator	double applyAsDouble(double)	한 개의 double 연산
IntBinaryOperator	int applyAsInt(int, int)	두개의 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

표준 API의 함수적 인터페이스

- Operator Example

```
public class OperatorExample {  
    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);  
    }  
}
```

표준 API의 함수적 인터페이스

- Predicate 함수적 인터페이스는 매개 변수와 boolean 리턴값이 있는 testXXX() 메소드를 가지고 있다.
- 이 메소드들은 매개값을 조사해서 true 혹은 false를 리턴하는 역할을 한다.

인터페이스명	추상 메소드	설명
Predicate<T>	boolean test(T t)	객체 T를 조사
BiPredicate<T,U>	boolean test(T t, U u)	객체 T와 U를 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

표준 API의 함수적 인터페이스

- Predicate 예제

```
public class Student {  
    private String name;  
    private String sex;  
    private int score;  
  
    public Student(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; }  
}
```

```
public class PredicateExample {  
    private static List<Student> list = Arrays.asList(  
        new Student("홍길동", "남자", 90),  
        new Student("김순희", "여자", 90),  
        new Student("감자바", "남자", 95),  
        new Student("박한나", "여자", 92)  
    );  
  
    public static double avg(Predicate<Student> predicate) {  
        int count = 0, sum = 0;  
        for(Student 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);  
    }  
}
```

Predicate<Student> 인스턴스 대입

andThen(), compose()

- andThen(), compose()
 - Consumer, Function, Operator 종류의 함수적 인터페이스는 andThen()과 compose() 디폴트 메소드를 가지고 있다.
 - andThen()과 compose() 디폴트 메소드는 두 개의 함수적 인터페이스를 순차적으로 연결하고, 첫 번째 처리 결과를 두 번째 매개값으로 제공해서 최종 결과값을 얻을 때 사용한다.
 - andThen()과 compose()의 차이점은 어떤 함수적 인터페이스부터 먼저 처리하느냐에 차이가 있다.

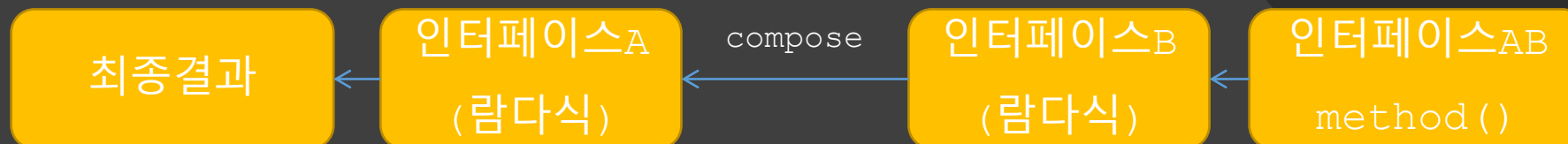
andThen(), compose()

- andThen(), compose()

```
인터페이스AB = 인터페이스A.andThen(인터페이스B);  
최종결과 = 인터페이스AB.method();
```



```
인터페이스AB = 인터페이스A.compose(인터페이스B);  
최종결과 = 인터페이스AB.method();
```



andThen(), compose()

- andThen(), compose()가 제공되는 메서드

종류	함수적 인터페이스	andThen()	compose()
Consumer	Consumer<T>	O	
	BiConsumer<T,U>	O	
	DoubleConsumer	O	
	IntConsumer	O	
	LongConsumer	O	
Function	Function<T,R>	O	O
	BiFunction<T,U,R>	O	
Operator	BinaryOperator<T>	O	
	DoubleUnaryOperator	O	O
	IntUnaryOperator	O	O
	LongUnaryOperator	O	O

andThen(), compose()

- consumer 예제

```
public class Member {  
    private String name;  
    private String id;  
    private Address address;
```

```
    public Member(String name, String id, Address address) {  
        this.name = name;  
        this.id = id;  
        this.address = address;  
    }
```

```
    public String getName() { return name; }  
    public String getId() { return id; }  
    public Address getAddress() { return address; }  
}
```

```
public class Address {  
    private String country;  
    private String city;  
  
    public Address(String country, String city) {  
        this.country = country;  
        this.city = city;  
    }  
  
    public String getCountry() { return country; }  
    public String getCity() { return city; }  
}
```

andThen(), compose()

- consumer 예제

```
public class ConsumerAndThenExample {  
    public static void main(String[] args) {  
        Consumer<Member> consumerA = (m) -> {  
            System.out.println("consumerA: " + m.getName());  
        };  
  
        Consumer<Member> consumerB = (m) -> {  
            System.out.println("consumerB: " + m.getId());  
        };  
  
        Consumer<Member> consumerAB = consumerA.andThen(consumerB);  
        consumerAB.accept(new Member("홍길동", "hong", null));  
    }  
}
```

andThen(), compose()

- Function 예제

```
public class FunctionAndThenComposeExample {
    public static void main(String[] args) {
        Function<Member, Address> functionA;
        Function<Address, String> functionB;
        Function<Member, String> functionAB;
        String city;

        functionA = (m) -> m.getAddress();
        functionB = (a) -> a.getCity();

        functionAB = functionA.andThen(functionB);
        city = functionAB.apply(
            new Member("홍길동", "hong", new Address("한국", "서울"))
        );
        System.out.println("거주 도시: " + city);

        functionAB = functionB.compose(functionA);
        city = functionAB.apply(
            new Member("홍길동", "hong", new Address("한국", "서울"))
        );
        System.out.println("거주 도시: " + city);
    }
}
```

and(), or(), negate() 디폴트 메서드와 isEqual() 정적 메서드

- Predicate 종류의 함수적 인터페이스는 and(), or(), negate() 디폴트 메소드를 가지고 있다.
- and() = &&, or() = ||, negate() = !

종류	함수적 인터페이스	and()	or()	negate()
Predicate	Predicate<T>	O	O	O
	BiPredicate<T,U>	O	O	O
	DoublePredicate	O	O	O
	IntPredicate	O	O	O
	LongPredicate	O	O	O

and(), or(), negate() 디폴트 메서드와 isEqual() 정적 메서드

- and, or, negate 예제

```
public class PredicateAndOrNegateExample {  
    public static void main(String[] args) {  
        //2의 배수 검사  
        IntPredicate predicateA = a -> a%2 == 0;  
        //3의 배수 검사  
        IntPredicate predicateB = (a) -> a%3 == 0;  
        IntPredicate predicateAB;  
        boolean result;  
        //and()  
        predicateAB = predicateA.and(predicateB);  
        result = predicateAB.test(9);  
        System.out.println("9는 2와 3의 배수입니까? " + result);  
        //or()  
        predicateAB = predicateA.or(predicateB);  
        result = predicateAB.test(9);  
        System.out.println("9는 2 또는 3의 배수입니까? " + result);  
        //negate()  
        predicateAB = predicateA.negate();  
        result = predicateAB.test(9);  
        System.out.println("9는 홀수입니까? " + result);  
    }  
}
```

and(), or(), negate() 디폴트 메서드와 isEqual() 정적 메서드

- isEqual 예제

```
public class PredicateIsEqualExample {  
    public static void main(String[] args) {  
        Predicate<String> predicate;  
  
        predicate = Predicate.isEqual(null);  
        System.out.println("null, null: " + predicate.test(null));  
  
        predicate = Predicate.isEqual("Java8");  
        System.out.println("null, Java8: " + predicate.test(null));  
  
        predicate = Predicate.isEqual(null);  
        System.out.println("Java8, null: " + predicate.test("Java8"));  
  
        predicate = Predicate.isEqual("Java8");  
        System.out.println("Java8, Java8: " + predicate.test("Java8"));  
  
        predicate = Predicate.isEqual("Java8");  
        System.out.println("Java7, Java8: " + predicate.test("Java7"));  
    }  
}
```


minBy(), maxBy()

- BinaryOperator<T> 함수적 인터페이스는 minBy()와 maxBy() 정적 메소드를 제공한다.
- 이 두 메소드는 매개값으로 제공되는 Comparator를 이용해서 최대 T와 최소 T를 얻는 BinaryOperator<T>를 리턴한다.

리턴 타입	정적 메소드
BinaryOperator<T>	minBy(Comparator<? super T>comparator)
BinaryOperator<T>	maxBy(Comparator<? super T>comparator)

minBy(), maxBy()

- 아래에서 처럼 Comparator<T>는 o1과 o2를 비교해서 o1이 작으면 음수를, o1과 o2가 동일하면 0을, o1이 크면 양수를 리턴하는 compare() 메소드가 선언되어 있다.

```
@FunctionalInterface
public interface Comparator<T>{
    public int compare(T o1, T o2);
}
```

- 람다식은 아래와 같이 작성 가능하다.

```
(o1, o2) -> { ...; return int값; }
```

minBy(), maxBy()

- 람다식은 아래와 같이 작성 가능하다.

```
(o1, o2) -> { ...; return int값; }
```

- 만약 o1과 o2가 int 타입이면 다음과 같이 Integer.compare(int, int) 메소드를 이용할 수 있다.
- Integer.compare()는 첫 번째 매개값이 두 번째 매개값보다 작으면 음수, 같으면 0, 크면 양수를 리턴한다.

```
(o1, o2) -> Integer.compare(o1, o2);
```

minBy(), maxBy()

- 두 파일의 값을 비교해서 낮거나 높은 파일을 얻어내는 예제

```
public class OperatorMinByMaxByExample {  
    public static void main(String[] args) {  
        BinaryOperator<Fruit> binaryOperator;  
        Fruit fruit;  
  
        binaryOperator = BinaryOperator.minBy((f1,f2)->Integer.compare(f1.price, f2.price));  
        fruit = binaryOperator.apply(new Fruit("딸기", 6000), new Fruit("수박", 10000));  
        System.out.println(fruit.name);  
  
        binaryOperator = BinaryOperator.maxBy((f1,f2)->Integer.compare(f1.price, f2.price));  
        fruit = binaryOperator.apply(new Fruit("딸기", 6000), new Fruit("수박", 10000));  
        System.out.println(fruit.name);  
    }  
}
```

```
public class Fruit {  
    public String name;  
    public int price;  
  
    public Fruit(String name, int price) {  
        this.name = name;  
        this.price = price;  
    }  
}
```

딸기
수박

메소드 참조

- 메소드 참조(Method References)는 메소드를 참조해서 매개변수의 정보 및 리턴 타입을 알아 내어, 람다식에서 불필요한 매개 변수를 제거하는 것이 목적이다.
- 정적 메소드를 참조할 경우 클래스 이름 뒤에 :: 기호를 붙이고 정적 메소드 이름을 기술하며 인스턴스 메소드일 경우에는 객체 생성 후 참조 변수 뒤에 :: 기호를 붙이고 인스턴스 메소드 이름을 기술하면 된다.

클래스 :: 메소드

참조변수 :: 메소드

메소드 참조

- 정적 메소드와 인스턴스 메소드 예제.

```
public class Calculator {  
    public static int staticMethod(int x, int y) {  
        return x + y;  
    }  
  
    public int instanceMethod(int x, int y) {  
        return x + y;  
    }  
}
```

```
public class MethodReferencesExample {  
    public static void main(String[] args) {  
        IntBinaryOperator operator;  
  
        //정적 메소드 참조 -----  
        operator = (x, y) -> Calculator.staticMethod(x, y);  
        System.out.println("결과1: " + operator.applyAsInt(1, 2));  
  
        operator = Calculator :: staticMethod;  
        System.out.println("결과2: " + operator.applyAsInt(3, 4));  
  
        //인스턴스 메소드 참조 -----  
        Calculator obj = new Calculator();  
        operator = (x, y) -> obj.instanceMethod(x, y);  
        System.out.println("결과3: " + operator.applyAsInt(5, 6));  
  
        operator = obj :: instanceMethod;  
        System.out.println("결과4: " + operator.applyAsInt(7, 8));  
    }  
}
```

메소드 참조

- 매개 변수의 메소드 참조.

```
(a, b) -> { a.instanceMethod(b); }
```




```
a 클래스 :: 인스턴스 메소드
```

메소드 참조

- 매개 변수의 메소드 참조.

```
public class ArgumentMethodReferencesExample {  
    public static void main(String[] args) {  
        ToIntBiFunction<String,String> function;  
  
        function = (a, b) -> a.compareToIgnoreCase(b);  
        print(function.applyAsInt("Java8", "JAVA8"));  
  
        function = String :: compareToIgnoreCase;  
        print(function.applyAsInt("Java8", "JAVA8"));  
    }  
  
    public static void print(int order) {  
        if(order<0) { System.out.println("사전순으로 먼저 옵니다."); }  
        else if(order == 0) { System.out.println("동일한 문자열입니다."); }  
        else { System.out.println("사전순으로 나중에 옵니다."); }  
    }  
}
```

동일한 문자열입니다.
동일한 문자열입니다.



메소드 참조

- 생성자 참조.

```
(a, b) -> { return new 클래스(a, b); }
```



```
클래스 :: new
```

메소드 참조

- 생성자 참조.

```
public class Member {  
    private String name;  
    private String id;  
  
    public Member() {  
        System.out.println("Member() 실행");  
    }  
    public Member(String id) {  
        System.out.println("Member(String id) 실행");  
        this.id = id;  
    }  
    public Member(String name, String id) {  
        System.out.println("Member(String name, String id)");  
        this.name = name;  
        this.id = id;  
    }  
  
    public String getId() { return id; }  
}
```

```
public class ConstructorReferencesExample {  
    public static void main(String[] args) {  
        Function<String, Member> function1 = Member :: new;  
        Member member1 = function1.apply("angel");  
  
        BiFunction<String, String, Member> function2 = Member :: new;  
        Member member2 = function2.apply("신천사", "angel");  
    }  
}
```