14장 람다식

14.1 람다식이란?

- 람다식은 익명 함수를 생성하기 위한 식으로 객체 지향 언어보다는 함수지향 언어에 가깝다.
- 자바에서 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 시에는 익명 구현 객체를 생성한다. 즉, 람다식 -> 매개 변수를 가진 코드 블록 -> 익명 구현 객체

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

Runnable runnable = () -> { ... }; // 람다식, (매개변수) -> {실행코드}
```

14.2 람다식 기본 문법

- 함수적 스타일의 람다식 작성법
  - 기본형: (타입 매개변수, ..) -> {실행문, ...}
  - 매개 변수를 이용해서 중괄호{}를 실행한다는 뜻으로 해석하면 된다.
- 타입은 런타임시에 대입값 따라 자동 인식 - 생략 가능
- 하나의 매개변수만 있을 경우에는 괄호( ) 생략 가능
- 하나의 실행문만 있다면 중괄호 { } 생략 가능
- 매개변수 없다면 괄호 ( ) 생략 불가
- 리턴값이 있는 경우, return 문 사용
- 중괄호 { }에 return 문만 있을 경우, 중괄호 생략 가능

```
(a) -> { System.out.println(a); }
a -> System.out.println(a)
() -> System.out.println()
(x,y) -> { return x+y; }
(x,y) -> x+y
```

14.3 타겟 타입과 함수적 인터페이스

- 람다식은 인터페이스의 익명 구현 객체를 생성한다.
- 람다식이 대입될 인터페이스를 람다식의 타켓 타입(target type)이라고 한다.

```
인터페이스 변수 = 람다식;
```

14.3.1 함수적 인터페이스(@FunctionalInterface)

- 하나의 추상 메소드만 선언된 인터페이스가 타겟 타입
- @FunctionalInterface 어노테이션
  - 하나의 추상 메소드만을 가지는지 컴파일러가 체크
  - 두 개 이상의 추상 메소드가 선언되어 있으면 컴파일 오류를 발생시킨다.
  - 선택사항이다. 이 어노테이션이 없더라도 하나의 추상 메소드만 있다면 모두 함수적 인터페이스이다.

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

14.3.2 매개 변수와 리턴값이 없는 람다식

- 다음과 같이 매개 변수와 리턴값이 없는 추상 메소드를 가진 함수적 인터페이스가 있다고 가정해 보자.

[MyFunctionalInterface.java] 함수적 인터페이스

```
01   package sec03.exam01_no_arguments_no_return;
02
03   @FunctionalInterface
04   public interface MyFunctionalInterface {
05            public void method();
06   }
```

[MyMethodReferencesExample.java] 람다식

```
01   package sec03.exam01_no_arguments_no_return;
02
03   public class MyFunctionalInterfaceExample {
04            public static void main(String[] args) {
05                     MyFunctionalInterface fi;
06
07                     fi = () -> {
08                              String str = "method call1";
09                              System.out.println(str);
10                     };
11                     fi.method();
12
13                     fi = () -> {
14                              System.out.println("method call2");
15                     };
16                     fi.method();
17
18                     fi = () -> System.out.println("method call3");  //실행문이 하나라면 중괄호는
19   생략가능
20                     fi.method();
21            }
     }
```

14.3.3 매개 변수가 있는 람다식

■ 다음과 같이 매개 변수가 있고 리턴값이 없는 추상 메소드를 가진 함수적 인터페이스가 있다고 보자.

[MyFunctionalInterface.java] 함수적 인터페이스

```
01  package sec03.exam02_arguments;
02
03  @FunctionalInterface
04  public interface MyFunctionalInterface {
05          public void method(int x);
06  }
```

[MyMethodReferencesExample.java] 람다식

```
01  package sec03.exam02_arguments;
02
03  public class MyMethodReferencesExample {
04          public static void main(String[] args) {
05                  MyFunctionalInterface fi;
06
07                  fi = (x) -> {
08                          int result = x * 5;
09                          System.out.println(result);
10                  };
11                  fi.method(2);
12
13                  fi = (x) -> {
14                          System.out.println(x * 5);
15                  };
16                  fi.method(2);
17
18                  fi = x -> System.out.println(x * 5); // 매개 변수가 하나일 경우에는 괄호()를 생략할
19  수 있다.
20                  fi.method(2);
21          }
22  }
```

14.3.4 리턴값이 있는 람다식

■ 다음은 매개 변수가 있고 리턴값이 있는 추상 메소드를 가진 함수의 인터페이스이다.

[MyFunctionalInterface.java] 함수적 인터페이스

```
01  package sec03.exam03_return;
02
03  @FunctionalInterface
04  public interface MyFunctionalInterface {
05          public int method(int x, int y);
06  }
```

[MyFunctionalInterfaceExample.java] 람다식

```java
package sec03.exam03_return;

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; // return문만 있을 경우 중괄호{}와 return문 생략 가능
		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);
	}
}
```

14.4 클래스 멤버와 로컬 변수 사용

14.4.1 클래스의 멤버 사용

- 람다식 실행 블록에는 클래스의 멤버인 필드와 메소드 제약 없이 사용
- 람다식 실행 블록 내에서 this는 람다식을 실행한 객체의 참조 -> 주의해서 사용해야 한다.

[MyFunctionalInterface.java] 함수적 인터페이스

```java
package sec04.exam01_field;

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

[UsingThis.java] this 사용

```java
package sec04.exam01_field;

public class UsingThis {
	public int outterField = 10;

	class Inner {
		int innerField = 20;

		void method() {
			// 람다식
			MyFunctionalInterface fi = () -> {
```

```
12                                          System.out.println("outterField: " + outterField);
13                                          System.out.println("outterField: " + UsingThis.this.outterField
14  + "\n");  // 바깥 객체의 참조를 얻기 위해서는 클래스명.this를 사용한다.
15
16                                          System.out.println("innerField: " + innerField);
17                                          System.out.println("innerField: " + this.innerField + "\n");  //
18  람다식 내부에서 this는 inner 객체를 참조한다.
19                                  };
20                                  fi.method();
21                      }
22              }
    }
```

[UsingThisExample.java] 실행 클래스

```
01  package sec04.exam01_field;
02
03  public class UsingThisExample {
04              public static void main(String... args) {
05                      UsingThis usingThis = new UsingThis();
06                      UsingThis.Inner inner = usingThis.new Inner();
07                      inner.method();
08              }
09  }
```

14.4.2 로컬 변수 사용

- 람다식은 함수적 인터페이스의 익명 구현 객체를 생성시킨다.
- 람다식에서 사용하는 외부 로컬 변수는 final 특성을 가져야 한다.

[MyFunctionalInterface.java] 함수적 인터페이스

```
01  package sec04.exam02_local_variable;
02
03  public interface MyFunctionalInterface {
04      public void method();
05  }
```

[UsingLocalVariable.java] final 특성을 가지는 로컬 변수

```
01  package sec04.exam02_local_variable;
02
03  public class UsingLocalVariable {
04              void method(int arg) { // arg는 final 특성을 가짐
05                      int localVar = 40; // localVar는 final 특성을 가짐
06
07                      // arg = 31; //final 특성 때문에 수정 불가
08                      // localVar = 41; //final 특성 때문에 수정 불가
09
10                      // 람다식
11                      MyFunctionalInterface fi = () -> {
12                              // 로컬변수 사용
13                              System.out.println("arg: " + arg);
14                              System.out.println("localVar: " + localVar + "\n");
15                      };
16                      fi.method();
17              }
18  }
```

[UsingLocalVariableExample.java] 실행 클래스

```
01   package sec04.exam02_local_variable;
02
03   public class UsingLocalVariableExample {
04            public static void main(String... args) {
05                     UsingLocalVariable ulv = new UsingLocalVariable();
06                     ulv.method(20);
07            }
08   }
```

14.5 표준 API의 함수적 인터페이스

- 한 개의 추상 메소드를 가지는 인터페이스들은 모두 람다식으로 사용 가능하다.
- 매개타입으로 사용되어 람다식을 매개값으로 대입할 수 있다.
- 예) Thread thread = new Thread(() -> { ... });
- 자바8부터는 빈번하게 사용되는 함수적 인터페이스는 java.util.function 표준 API 패키지로 제공한다.
- 인터페이스에 선언된 추상 메소드의 매개값과 리턴값의 유무 따라 구분된다.

종류	추상 메소드 특징	
Consumer	매개값은 있고 리턴값이 없음	매개값 -> Consumer
Supplier	매개값은 없고 리턴값은 있음	Supplier -> 리턴값
Function	매개값도 있고 리턴값도 있음 매개값을 리턴값으로 매핑(타입변환)	매개값 -> Function -> 리턴값
Operator	매개값도 있고 리턴값도 있음 매개값을 연산하고 결과를 리턴	매개값 -> Operator -> 리턴값
Predicate	매개값도 있고 리턴값도 있음 매개값을 조사해서 true/false를 리턴	매개값 -> Predicate -> 리턴값

[RunnableExample.java] 함수적 인터페이스의 람다식

```
01   package sec05.exam01_runnable;
02
03   public class RunnableExample {
04            public static void main(String[] args) {
05                     /*Runnable runnable = () -> {
06                              for(int i=0; i<10; i++) {
07                                       System.out.println(i);
08                              }
09                     };
10
11                     Thread thread = new Thread(runnable);
12                     thread.start();*/
13
14                     Thread thread = new Thread(() -> {
15                              for(int i=0; i<10; i++) {
16                                       System.out.println(i);
17                              }
18                     });
19                     thread.start();
20            }
21   }
```

14.5.1 Consumer 함수적 인터페이스

- 매개값만 있고 리턴값이 없는 추상 메소드 가짐

- 매개 변수의 타입과 수에 따라 분류

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 double 값을 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

```
Consumer<String> consumer = t -> { t를 소비하는 실행문; };
BiConsumer<String, String> consumer = (t,u) -> { t와 u를 소비하는 실행문; };
DoubleConsumer consumer = d -> { t를 소비하는 실행문; };
ObjIntConsumer<String> consumer = (t,i) -> { t와 i를 소비하는 실행문; };
```

[ConsumerExample.java] Consumer 함수적 인터페이스

```
01   package sec05.exam02_consumer;
02
03   import java.util.function.BiConsumer;
04   import java.util.function.Consumer;
05   import java.util.function.DoubleConsumer;
06   import java.util.function.ObjIntConsumer;
07
08   public class ConsumerExample {
09            public static void main(String[] args) {
10                     Consumer<String> consumer = t -> System.out.println(t + "8");
11                     consumer.accept("java");
12
13                     BiConsumer<String, String> bigConsumer = (t, u) -> System.out.println(t + u);
14                     bigConsumer.accept("Java", "8");
15
16                     DoubleConsumer doubleConsumer = d -> System.out.println("Java" + d);
17                     doubleConsumer.accept(8.0);
18
19                     ObjIntConsumer<String> objIntConsumer = (t, i) -> System.out.println(t + i);
20                     objIntConsumer.accept("Java", 8);
21            }
22   }
```

14.5.2 Supplier 함수적 인터페이스

- 매개값은 없고 리턴값만 있는 추상 메소드 가짐

- 리턴 타입 따라 분류

인터페이스명	추상 메소드	설명
Supplier<T>	T get()	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

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

[SupplierExample.java] Supplier 함수적 인터페이스

```
01  package sec05.exam03_supplier;
02
03  import java.util.function.IntSupplier;
04
05  public class SupplierExample {
06          public static void main(String[] args) {
07                  IntSupplier intSupplier = () -> {
08                          int num = (int) (Math.random() * 6) + 1;
09                          return num;
10                  };
11
12                  int num = intSupplier.getAsInt();
13                  System.out.println("눈의 수: " + num);
14          }
15  }
```

14.5.3 Function 함수적 인터페이스

- 매개값과 리턴값이 모두 있는 추상 메소드 가짐

- 주로 매개값을 리턴값으로 매핑(타입 변환)할 경우 사용
- 매개 변수 타입과 리턴 타입 따라 분류

인터페이스명	추상 메소드	설명
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	long applyAsLong(int value)	int 를 long 으로 매핑
LongToDoubleFunction	double applyAsDouble(long value)	long 을 double 로 매핑
LongToIntFunction	int applyAsInt(long value)	long 을 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)	객체 T 와 U 를 long 으로 매핑
ToLongFunction<T>	long applyAsLong(T value)	객체 T 를 long 으로 매핑

```
//Student 객체를 학생 이름(String)으로 매핑
//Function<Student, String> function = t -> { return t.getName(); }
```

```
Function<Student, String> function = t -> t.getName();

//Student 객체를 학생 점수(int)로 매핑
ToIntFunction<Student> function = t -> t.getScore();
```

[FunctionExample1.java] Function 함수적 인터페이스

```
01   package sec05.exam04_function;
02
03   import java.util.Arrays;
04   import java.util.List;
05   import java.util.function.Function;
06   import java.util.function.ToIntFunction;
07
08   public class FunctionExample1 {
09            private static List<Student> list = Arrays.asList(new Student("홍길동", 90, 96), new
10   Student("신용권", 95, 93));
11
12            public static void printString(Function<Student, String> function) {
13                     for (Student student : list) {
14                              System.out.print(function.apply(student) + " ");
15                     }
16                     System.out.println();
17            }
18
19            public static void printInt(ToIntFunction<Student> function) {
20                     for (Student student : list) {
21                              System.out.print(function.applyAsInt(student) + " ");
22                     }
23                     System.out.println();
24            }
25
26            public static void main(String[] args) {
27                     System.out.println("[학생 이름]");
28                     printString(t -> t.getName());
29
30                     System.out.println("[영어 점수]");
31                     printInt(t -> t.getEnglishScore());
32
33                     System.out.println("[수학 점수]");
34                     printInt(t -> t.getMathScore());
35            }
36   }
```

[Student.java] Student 클래스

```
01   package sec05.exam04_function;
02
03   public class Student {
04            private String name;
05            private int englishScore;
06            private int mathScore;
07
08            public Student(String name, int englishScore, int mathScore) {
09                     this.name = name;
10                     this.englishScore = englishScore;
11                     this.mathScore = mathScore;
12            }
13
14            public String getName() {
15                     return name;
16            }
```

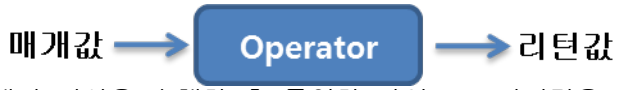
```
17
18              public int getEnglishScore() {
19                      return englishScore;
20              }
21
22              public int getMathScore() {
23                      return mathScore;
24              }
25      }
```

[FunctionExample2.java] Function 함수적 인터페이스

```
01      package sec05.exam04_function;
02
03      import java.util.Arrays;
04      import java.util.List;
05      import java.util.function.ToIntFunction;
06
07      public class FunctionExample2 {
08              private static List<Student> list = Arrays.asList(new Student("홍길동", 90, 96), new Student("
09      신용권", 95, 93));
10
11              public static double avg(ToIntFunction<Student> function) {
12                      int sum = 0;
13                      for (Student student : list) {
14                              sum += function.applyAsInt(student);
15                      }
16                      double avg = (double) sum / list.size();
17                      return avg;
18              }
19
20              public static void main(String[] args) {
21                      double englishAvg = avg(s -> s.getEnglishScore());
22                      System.out.println("영어 평균 점수: " + englishAvg); // 영어 평균 점수: 92.5
23
24                      double mathAvg = avg(s -> s.getMathScore());
25                      System.out.println("수학 평균 점수: " + mathAvg); // 수학 평균 점수: 94.5
26              }
27      }
```

14.5.4. Operator 함수적 인터페이스

- 매개값과 리턴값이 모두 있는 추상 메소드 가짐



- 매개값을 이용해서 연산을 수행한 후 동일한 타입으로 리턴값을 제공하는 역할을 한다.
- 매개 변수의 타입과 수에 따라 분류

인터페이스명	추상 메소드	설명
BinaryOperator<T>	BiFunction<T,U,R>의 하위 인터페이스	T와 U를 연산한 후 R리턴
UnaryOperator<T>	Function<T,R>의 하위 인터페이스	T를 연산한 후 R 리턴
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

```
//두 개의 int를 연산해서 결과값으로 int를 리턴한다.
IntBinaryOperator operator = (a,b) -> { ...; return int값; }
```

[OperatorExample.java] Operator 함수적 인터페이스

```
01  package sec05.exam05_operator;
02
03  import java.util.function.IntBinaryOperator;
04
05  public class OperatorExample {
06          private static int[] scores = { 92, 95, 87 };
07
08          public static int maxOrMin(IntBinaryOperator operator) {
09                  int result = scores[0];
10                  for (int score : scores) {
11                          result = operator.applyAsInt(result, score); // 람다식{} 실행
12                  }
13                  return result;
14          }
15
16          public static void main(String[] args) {
17                  // 최대값 얻기
18                  int max = maxOrMin((a, b) -> {
19                          if (a >= b)
20                                  return a;
21                          else
22                                  return b;
23                  });
24                  System.out.println("최대값: " + max);
25
26                  // 최소값 얻기
27                  int min = maxOrMin((a, b) -> {
28                          if (a <= b)
29                                  return a;
30                          else
31                                  return b;
32                  });
33                  System.out.println("최소값: " + min);
34          }
35  }
```

14.5.5. Predicate 함수적 인터페이스

- 매개값 조사해 true 또는 false를 리턴할 때 사용

- 매개변수 타입과 수에 따라 분류

인터페이스명	추상 메소드	설명
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

```
//String의 equals() 메소드를 이용해서 남학생만 true를 리턴한다.
//Predicate<Student> predicate = t -> { return t.getSex().equals("남자"); }
Predicate<Student> predicate = t -> t.getSex().equals("남자");
```

[PredicateExample.java] Predicate 함수적 인터페이스

```
01  package sec05.exam06_predicate;
02
03  import java.util.Arrays;
04  import java.util.List;
05  import java.util.function.Predicate;
06
07  public class PredicateExample {
08          private static List<Student> list = Arrays.asList(
09                          new Student("홍길동", "남자", 90),
10                          new Student("김순희", "여자", 90),
11                          new Student("감자바", "남자",  95),
12                          new Student("박한나", "여자", 92)
13          );
14
15          public static double avg(Predicate<Student> predicate) {
16                  int count = 0, sum = 0;
17                  for(Student student : list) {
18                          if(predicate.test(student)) {
19                                  count++;
20                                  sum += student.getScore();
21                          }
22                  }
23                  return (double) sum / count;
24          }
25
26          public static void main(String[] args) {
27                  double maleAvg = avg( t ->  t.getSex().equals("남자") );
28                  System.out.println("남자 평균 점수: " + maleAvg);
29
30                  double femaleAvg = avg( t ->  t.getSex().equals("여자") );
31                  System.out.println("여자 평균 점수: " + femaleAvg);
32          }
33  }
```

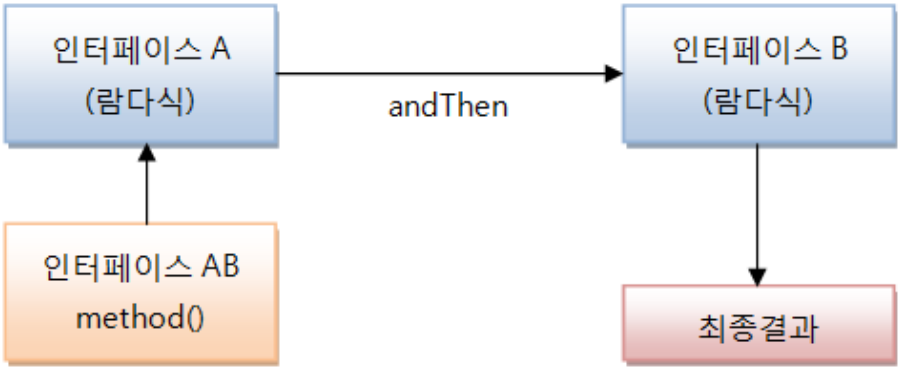
[Student.java] Student 클래스

```
01  package sec05.exam06_predicate;
02
03  public class Student {
04          private String name;
05          private String sex;
06          private int score;
07
08          public Student(String name, String sex, int score) {
09                  this.name = name;
10                  this.sex = sex;
11                  this.score = score;
12          }
13
14          public String getSex() { return sex; }
15          public int getScore() { return score; }
16  }
```

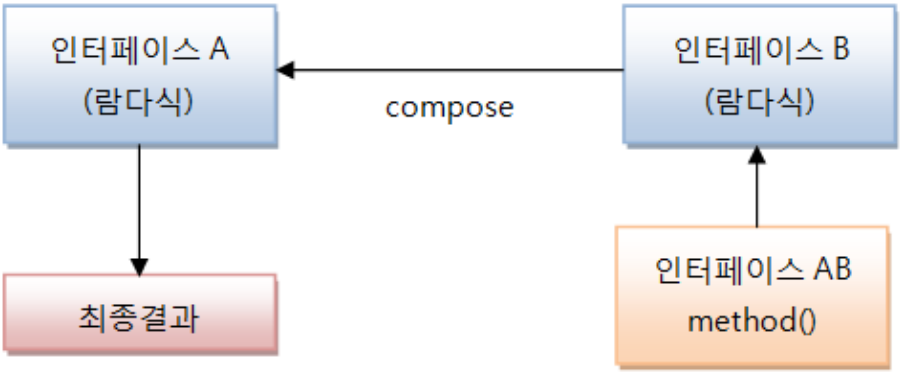
14.5.6. andThen()과 compose() 디폴트 메소드

- 함수적 인터페이스가 가지고 있는 디폴트 메소드
- 두 개의 함수적 인터페이스를 순차적으로 연결해 실행
- 첫 번째 리턴값을 두 번째 매개값으로 제공해 최종 결과값 리턴
- andThen()과 compose()의 차이점
  - 어떤 함수적 인터페이스부터 처리하느냐

```
인터페이스 AB = 인터페이스 A.andThen(인터페이스 B);
최종결과 = 인터페이스 AB.method();
```




```
인터페이스 AB = 인터페이스 A.compose(인터페이스 B);
최종결과 = 인터페이스 AB.method();
```



(1) Consumer의 순차적 연결

- Consumer 종류의 함수적 인터페이스는 처리 결과를 리턴하지 않기 때문에 andThen() 디폴트 메소드는 함수적 인터페이스의 호출 순서만 정한다.

(2) Function의 순차적 연결

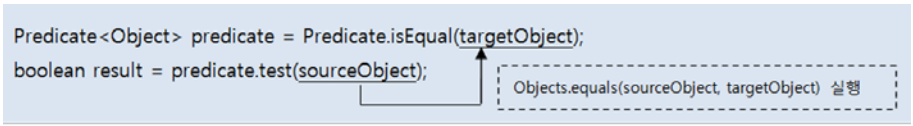
- Function과 Operator 종류의 함수적 인터페이스는 먼저 실행한 함수적 인터페이스의 결과를 다음 함수적 인터페이스의 매개값으로 넘겨주고, 최종 처리 결과를 리턴한다.

14.5.7. and(), or(), negate() 디폴트 메소드와 isEqual() 정적 메소드

- Predicate 함수적 인터페이스의 디폴트 메소드
- and() - &&와 대응
  - 두 Predicate가 모두 true를 리턴 → 최종적으로 true 리턴
- or() - || 와 대응
  - 두 Predicate 중 하나만 true를 리턴 → 최종적으로 true 리턴
- negate() - ! 와 대응
  - Predicate의 결과가 true이면 false, false이면 true 리턴
- Predicate 함수적 인터페이스

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

- isEqual() 정적 메소드



Objects.equals(sourceObject, targetObject)는 다음과 같은 리턴값을 제공한다.

sourceObject	targetObject	리턴값
null	null	true
not null	null	false
null	not null	false
not null	not null	sourceObject.equals(targetObject)의 리턴값

14.5.8. minBy(), maxBy() 정적 메소드

- BinaryOperator<T> 함수적 인터페이스의 정적 메소드
- Comparator를 이용해 최대 T와 최소 T를 얻는 BinaryOperator<T> 리턴

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

14.6. 메소드 참조

- 메소드 참조는 람다식에서 불필요한 매개 변수를 제거하는 것이 목적이다.
- 메소드 참조도 인터페이스의 익명 구현 객체로 생성된다.

```
(left, right) -> Math.max(left, right);
Math::max; // 메소드 참조

IntBinaryOperator operator = Math::max;
```

14.6.1. 정적 메소드와 인스턴스 메소드 참조

```
// 정적 메소드 참조
클래스::메소드

// 인스턴스 메소드 참조
참조변수::메소드
```

[Calculator.java] 정적 및 인스턴스 메소드

```
package sec06.exam01_method_references;

public class Calculator {
	public static int staticMethod(int x, int y) { // 정적 메소드
		return x + y;
	}

	public int instanceMethod(int x, int y) { // 인스턴스 메소드
		return x + y;
	}
}
```

[MethodReferencesExample.java] 정적 및 인스턴스 메소드 참조

```
package sec06.exam01_method_references;

import java.util.function.IntBinaryOperator;

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));
	}
}
```

14.6.2. 매개 변수의 메소드 참조

- 람다식에서 제공되는 a 매개변수의 메소드를 호출해서 b 매개변수를 매개값으로 사용하는 경우도 있다.
- 작성 방법은 정적 메소드 참조와 동일하지만 a 매개변수의 인스턴스 메소드가 참조되므

로 전혀 다른 코드가 실행된다.

```
// 매개변수의 메소드 사용
(a, b) -> { a.instanceMethod(b); }

// a의 인스턴스 메소드 참조
a의 클래스 :: instanceMethod
```

[ArgumentMethodReferencesExample.java] 매개 변수의 메소드 참조

```
package sec06.exam02_argument_method_references;

import java.util.function.ToIntBiFunction;

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.");
		}
	}
}
```

14.6.3. 생성자 참조

- 생성자를 참조한다는 것은 객체 생성을 의미한다.
- 생성자가 오버로딩되어 여러 개가 있을 경우, 컴파일러는 함수적 인터페이스의 추상메소드와 동일한 매개변수 타입과 개수를 가지고 있는 생성자를 찾아 실행한다.

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

[ConstructorReferencesExample.java] 생성자 참조

```
package sec06.exam03_constructor_references;

import java.util.function.BiFunction;
import java.util.function.Function;

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");
          }
}
```

[Member.java] 생성자 오버로딩

```
package sec06.exam03_constructor_references;

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;
          }
}
```

[과제] 확인문제

1. 람다식에 대한 설명으로 틀린 것은 무엇입니까?

(1) 람다식은 함수적 인터페이스의 익명 구현 객체를 생성한다.
(2) 매개 변수가 없을 경우 () -> {...} 형태로 작성한다.
(3) (x,y) -> { return x+y; }는 (x,y) -> x+y로 바꿀 수 있다.
(4) @FunctionalInterface가 기술된 인터페이스만 람다식으로 표현이 가능하다.

2. 메소드 참조에 대한 설명으로 틀린 것은 무엇입니까?

(1) 메소드 참조는 함수적 인터페이스의 익명 구현 객체를 생성한다.
(2) 인스턴스 메소드는 "참조변수::메소드"로 기술한다.
(3) 정적 메소드는 "클래스:::메소드"로 기술한다.
(4) 생성자 참조인 "클래스::new"는 매개 변수가 없는 디폴트 생성자만 호출한다.

3. 잘못 작성된 람다식은 무엇입니까?

(1) a -> a+3
(2) a,b -> a*b

(3) x -> System.out.println(x/5)
(4) (x,y) -> Math.max(x,y)

4. 다음 코드는 컴파일 에러가 발생합니다. 그 이유가 무엇입니까?

[LambdaExample.java]

```
01    package verity.exam04;
02
03    import java.util.function.IntSupplier;
04
05    public class LambdaExample {
06              public static int method(int x, int y) {
07                        IntSupplier supplier = () -> {
08                                  x *= 10;
09                                  int result = x + y;
10                                  return result;
11                        };
12                        int result = supplier.getAsInt();
13                        return result;
14              }
15
16              public static void main(String[] args) {
17                        System.out.println(method(3,5));
18              }
19    }
20
```

5. 다음은 배열 항목 중에 최대값 또는 최소값을 찾는 코드입니다. maxOrMin() 메소드의 매개값을 람다식으로 기술해보세요.

[LambdaExample.java]

```
01    package verity.exam05;
02
03    import java.util.function.IntBinaryOperator;
04
05    public class LambdaExample {
06              private static int[] scores = { 10, 50, 3 };
07
08              public static int maxOrMin(IntBinaryOperator operator) {
09                        int result = scores[0];
10                        for(int score : scores) {
11                                  result = operator.applyAsInt(result, score);
12                        }
13                        return result;
14              }
15
16              public static void main(String[] args) {
17                        //최대값 얻기
18                        int max = maxOrMin(     #1
19
20
21
22
23                        );
24                        System.out.println("최대값: " + max);
25
26                        //최소값 얻기
```

```
27                      int min = maxOrMin(     #2
28
29
30
31
32                      );
33                      System.out.println("최소값: " + min);
34              }
35      }
36
37      // 실행 결과
38      // 최대값: 50
39      // 최소값: 3
```

6. 다음은 학생의 영어 평균 점수와 수학 평균 점수를 계산하는 코드입니다. avg() 메소드를 선언해보세요.

[LambdaExample.java]

```
01      package verity.exam06;
02
03      import java.util.function.ToIntFunction;
04
05      public class LambdaExample {
06              private static Student[] students = {
07                      new Student("홍길동", 90, 96),
08                      new Student("신용권", 95, 93)
09              };
10
11
12              // avg() 메소드 작성 위치, #1
13
14
15
16
17
18
19
20              public static void main(String[] args) {
21                      double englishAvg = avg( s -> s.getEnglishScore() );
22                      System.out.println("영어 평균 점수: " + englishAvg);
23
24                      double mathAvg = avg( s -> s.getMathScore() );
25                      System.out.println("수학 평균 점수: " + mathAvg);
26              }
27
28              public static class Student {
29                      private String name;
30                      private int englishScore;
31                      private int mathScore;
32
33                      public Student(String name, int englishScore, int mathScore) {
34                              this.name = name;
35                              this.englishScore = englishScore;
36                              this.mathScore = mathScore;
37                      }
38
39                      public String getName() { return name; }
40                      public int getEnglishScore() { return englishScore; }
41                      public int getMathScore() { return mathScore; }
42              }
```

```
43    }
44
45    // 실행 결과
46    // 영어 평균 점수: 92.5
47    // 수학 평균 점수: 94.5
```

7. 6번의 main() 메소드에서 avg()를 호출할 때 매개값으로 준 람다식을 메소드 참조로 변경해보세요.

[LambdaExample.java]

```
01    package verity.exam07;
02
03    import java.util.function.ToIntFunction;
04
05    public class LambdaExample {
06              private static Student[] students = {
07                        new Student("홍길동", 90, 96),
08                        new Student("신용권", 95, 93)
09              };
10
11              public static double avg(ToIntFunction<Student> function) {
12                        int sum = 0;
13                        for(Student student : students) {
14                                  sum += function.applyAsInt(student);
15                        }
16                        double avg = (double) sum / students.length;
17                        return avg;
18              }
19
20              public static void main(String[] args) {
21                        double englishAvg = avg(    #1     );
22                        System.out.println("영어 평균 점수: " + englishAvg);
23
24                        double mathAvg = avg(     #2     );
25                        System.out.println("수학 평균 점수: " + mathAvg);
26              }
27
28              public static class Student {
29                        private String name;
30                        private int englishScore;
31                        private int mathScore;
32
33                        public Student(String name, int englishScore, int mathScore) {
34                                  this.name = name;
35                                  this.englishScore = englishScore;
36                                  this.mathScore = mathScore;
37                        }
38
39                        public String getName() { return name; }
40                        public int getEnglishScore() { return englishScore; }
41                        public int getMathScore() { return mathScore; }
42              }
43    }
```