1. 람다식(Lambda Expressions)이란

01. 자바 8부터 함수적 프로그래밍 위해 람다식 지원

- 람다식(Lambda Expressions)을 언어 차원에서 제공
  - 람다 계산법에서 사용된 식을 프로그래밍 언어에 접목
  - 익명 함수(anonymouse function)을 생성하기 위한 식
- 자바에서 람다식을 수용한 이유
  - 코드가 매우 간결해진다
  - 컬렉션 요소(대용량 데이터)를 필터링 또는 매핑해 쉽게 집계
- 자바는 람다식을 함수적 인터페이스의 익명 구현 객체로 취급

```
람다식 -> 매개변수를 가진 코드 블록 -> 익명 구현 객체
```

  - 어떤 인터페이스를 구현할지는 대입되는 인터페이스 달려 있음

```
Runable runable = () -> { ... }
```

2. 람다식 기본 문법

01. 함수적 스타일의 람다식 작성법

```
(타입 매개변수, ...) -> { 실행문; ... }
(int a) -> { System.out.println(a); }
```

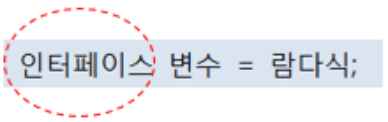
- 매개 타입은 런타임시에 대입값에 따라 자동 인식 -> 생략 가능
- 하나의 매개변수만 있을 경우에는 괄호() 생략 가능
- 하나의 실행문만 있다면 중괄호 {} 생략 가능
- 매개변수 없다면 괄호 () 생략 불가
- 리턴ㄱ밧이 있는 경우 return문 사용
- 중괄호 {}에 return문만 있을 경우, 중괄호 생략 가능

3. 타겟 타입과 함수적 인터페이스

01. 타겟 타입(target type)

- 람다식은 대입되는 인터페이스
- 익명 구현 객체를 만들 때 사용할 인터페이스

```
인터페이스 변수 = 람다식;
```



02. 함수적 인터페이스(functional interface)

- 하나의 추상 메소드만 선언된 인터페이스 타겟 타입
- @FunctionalInterface 어노테이션
  - 하나의 추상 메소드만을 가지는지 컴파일러가 체크
  - 두 개 이상의 추상 메소드가 선언되어 있으면 컴파일 오류 발생

03. 매개변수와 리턴값이 ㅇ벗는 람다식

- Method()가 매개 변수를 가지지 않는 경우

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

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

04. 매개변수가 있는 람다식

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

MyFunctionalInterface fi = (x) -> { ... }
or
MyFunctionalInterface fi = x -> { ... }
fi.method(5);
```

05. 리턴값이 있는 람다식

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

MyFunctionalInterface fi = (x, y) -> { ...; return 값; }
int result = fi.method(2, 5);

MyFunctionalInterface fi = (x, y) -> { return x+ y; }
or
MyFunctionalInterface fi = (x, y) -> return x+ y;

MyFunctionalInterface fi = (x, y) -> { return sum(x, y); }
or
MyFunctionalInterface fi = (x, y) -> sum(x, y);
```

4. 클래스 멤버와 로컬 변수 사용

01. 클래스의 멤버 사용

- 람다식 실행 블록에는 클래스의 멤버인 필드와 메소드 제약 없이 사용
- 람다식 실행 블록 내에서 this는 람다식을 실행한 객체의 참조
    - 주의해서 사용해야 할 필요성 가짐

```
public class ThisExaple {
   public int outterField = 10;

 class inner {
   int innerField = 20;

     void method() {
       // 람다식
         MyFunctionalInterface fi = 0 -> {
           System.out.println("outterFiled"+outterField);
               System.out.println("outterFiled"+ThisExaple.this.outterField+"\n");
               // 바깥 객체의 참조를 얻기 위해서는 클래스명.this를 사용


             System.out.println("innerField"+innerField);
               System.out.println("innerField"+this.innerField+"\n");
               // 람다식 내부에서 this는 innser 객체를 참조

         };
         fi.method();
  }
```

02. 로컬 변수의 사용

- 람다식은 함수적 인터페이스의 익명 구현 객체 생성
- 람다식에서 사용하는 외부 로컬 변수는 final 특성

```
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 = () -> {
            // 로컬 변수 사용
            System.out.println("arg : "+arg);
            System.out.println("localVar : "+localVar);
        }
        fi.method();
    }
}
```

5. 표준 API의 함수적 인터페이스

01. 자바 8부터 표준 API로 제공되는 함수적 인터페이스

- java.util.function 패키지에 포함
- 매개타입으로 사용되어 람다식을 매개값으로 대입할 수 있도록 한 개의 추상 메소드를 가지는 인터페이스들은 모두 람다식 사용가능
- 인터페이스에 선언된 추상 메소드의 매개값과 리턴 유무 따라 구분

02. Consumer 함수적 인터페이스

- 매개값만 있고 리턴값이 없는 추상 메소드 가짐

```
매개값 -> Consumer
```

- 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 -> { d를 소비하는 실행문; }

ObjIntConsumer<String> consumer = (t, i) -> { t와 i를 소비하는 실행문; }
```

- ConsumerExample.java, Consumer함수적 인터페이스

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

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> consumer = t -> System.out.println(t + "8");
        consumer.accept("java");

        BiConsumer<String, String> biConsumer = (t, u) -> System.out.println(t + u);
        biConsumer.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);
    }
}
```

03. Supplier 함수적 인터페이스

- 매개값은 없고 리턴값만 있는 추상 메소드 가짐

```
Supplier -> 리턴값
```

- 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 int 값; }
```

- SupplierExample.java, Supplier 함수적 인터페이스

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

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);
    }
}
```

04. 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

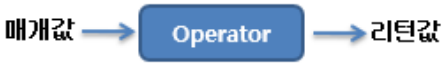
```
매개값 -> function -> 리턴값
```

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

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

05. Operator 함수적 인터페이스

- 매개값과 리턴값이 모두 있는 추상 메소드 가짐
- 주로 매개값을 연산하고 그 결과를 리턴할 경우에 사용
- 매개 변수의 타입과 수에 따라 분류



인터페이스명	추상 메소드	설명
BinaryOperator<T>	BiFunction<T,U,R>의 하위 인터페이스	T 와 U 를 연산한 후 R 리턴
UnaryOperator<T>	Function<T,R>의 하위 인터페이스	T 를 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

06. Predicate 함수적 인터페이스

- 매개값 조사해 true 또는 false를 리턴할 때 사용

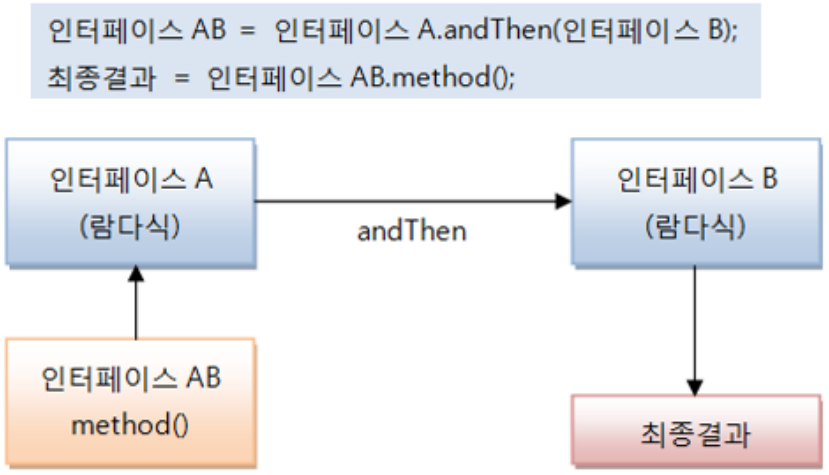
- 매개변수 타입과 수에 따라 분류

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

07. andThen()과 compose() 디폴트 메소드

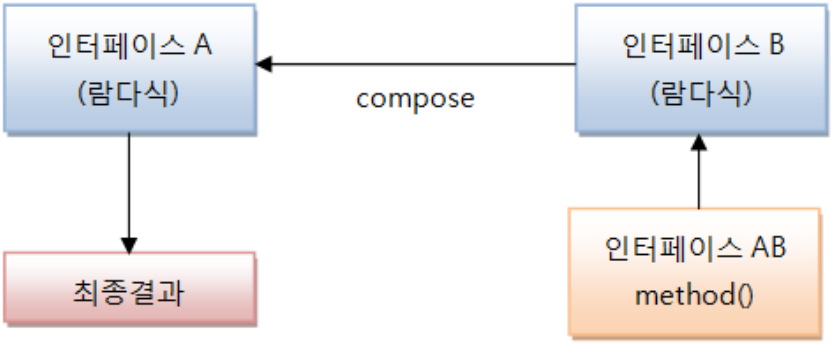
- 함수적 인터페이스가 가지고 있는 디폴트 메소드
- 두 개의 함수적 인터페이스를 순차적으로 연결해 실행
- 첫 번째 리턴값을 두 번째 매개값으로 제공해 최종 결과값 리턴
- andThen()과 compose()의 차이점
  - 어떤 함수적 인터페이스부터 처리하느냐

08. andThen() 디폴트 메소드



09. compose() 디폴트 메소드

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



10. Consumer의 순차적 연결

- 처리 결과 리턴하지 않음
- andThen()과 compose() 디폴트 메소드의 경우
  - 함수적 인터페이스의 호출 순서만 정할 것

11. Operator와 Function 함수 인터페이스 순차적 연결

- 먼저 실행한 함수적 인터페이스의 결과를 다음 함수적 인터페이스의 매개값으로 넘겨주고, 최종 처리결과 리턴(p.705 ~ 707)

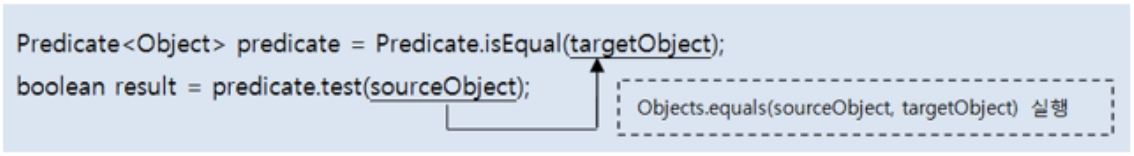
12. and(), or(), negate() 디폴트 메소드

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

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

13. isEqual() 정적 메소드

- Predicate의 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. minBy(), maxBy() 정적 메소드

- BinaryOperator함수적 인터페이스의 정적 메소드
- Comparator를 이용해 최대 T와 최소 T를 얻는 BinaryOperator 리턴

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

Comparator<T>는 다음과 같이 선언된 함수적 인터페이스이다. o1 과 o2 를 비교해서 o1 이 작으면 음수를, o1 과 o2 가 동일하면 0, o1 이 크면 양수를 리턴해야하는 compare() 메소드가 선언되어 있다.

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

Comparator<T>를 타겟 타입으로하는 람다식은 다음과 같이 작성할 수 있다.

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

만약 o1 과 o2 가 int 타입이라면 다음과 같이 Integer.compare(int, int) 메소드를 이용할 수 있다.

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

6. 메소드 참조(Method references)

01. 메소드 참조(Method references)

- 메소드 참조해 매개변수의 정보 및 리턴 타입 알아냄
  - 람다식에서 불필요한 매개변수를 제거하는 것이 목적
  - 종종 람다식은 기존 메소드를 단순하게 호출만 하는 경우로 존재
- 메소드 참조도 인터페이스의 익명 구현 객체로 생성

- 타겟 타입에서 추상 메소드의 매개변수 및 리턴 타입 따라 메소드 참조도 달라짐
- Ex) IntBinaryOperator 인터페이스
  - 두 개의 int 매개값을 받아 int 값 리턴
  - 동일한 매개값과 리턴 타입 갖는 Math 클래스의 max() 참조

02. 정적 메소드와 인스턴스 메소드 참조

- 정적 메소드 참조

```
클래스 :: 메소드
```

- 인스턴스 메소드 참조

```
참조변수 :: 메소드
```

03. 매개변수의 메소드 참조

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

04. 생성자 참조

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