

람다 표현식

27-1.

람다와 함수형 인터페이스

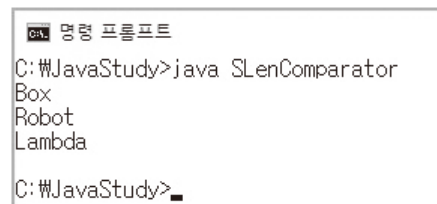
인스턴스보다 기능 하나가 필요한 상황을 위한 람다

```
class SLenComp implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length()
    }
}

class SLenComparator {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Robot");
        list.add("Lambda");
        list.add("Box");

        Collections.sort(list, new SLenComp()); // 정렬

        for(String s : list)
            System.out.println(s);
    }
}
```



```
C:\> 명령 프롬프트
C:\> cd C:\JavaStudy
C:\JavaStudy> java SLenComparator
Box
Robot
Lambda
C:\JavaStudy>
```

매개변수가 있고 반환하지 않는 람다식

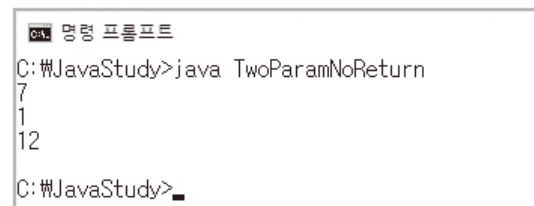
```
interface Printable {  
    void print(String s); // 매개변수 하나, 반환형 void  
}  
  
class OneParamNoReturn {  
    public static void main(String[] args) {  
        Printable p;  
        p = (String s) -> { System.out.println(s); }; // 줄임 없는 표현  
        p.print("Lambda exp one.");  
  
        p = (String s) -> System.out.println(s); // 중괄호 생략  
        p.print("Lambda exp two.");  
  
        p = (s) -> System.out.println(s); // 매개변수 형 생략  
        p.print("Lambda exp three.");  
  
        p = s -> System.out.println(s); // 매개변수 소괄호 생략  
        p.print("Lambda exp four.");  
    }  
}
```

메소드 몸체가 둘 이상의 문장으로 이뤄져 있거나, 매개변수의 수가 둘 이상인 경우에는 각각 중괄호와 소괄호의 생략이 불가능하다.

매개변수가 둘 인 람다식

```
interface Calculate {  
    void cal(int a, int b); // 매개변수 둘, 반환형 void  
}
```

```
class TwoParamNoReturn {  
    public static void main(String[] args) {  
        Calculate c;  
        c = (a, b) -> System.out.println(a + b);  
        c.cal(4, 3);    // 이번엔 덧셈이 진행  
  
        c = (a, b) -> System.out.println(a - b);  
        c.cal(4, 3);    // 이번엔 뺄셈이 진행  
  
        c = (a, b) -> System.out.println(a * b);  
        c.cal(4, 3);    // 이번엔 곱셈이 진행  
    }  
}
```

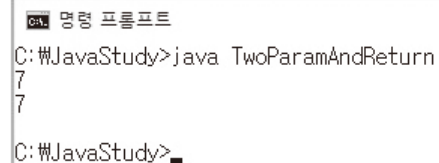


```
명령 프롬프트  
C:\JavaStudy>java TwoParamNoReturn  
7  
1  
12  
C:\JavaStudy>
```

매개변수가 있고 반환하는 람다식1

```
interface Calculate {  
    int cal(int a, int b); // 값을 반환하는 추상 메소드  
}
```

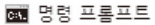
```
class TwoParamAndReturn {  
    public static void main(String[] args) {  
        Calculate c;  
        c = (a, b) -> { return a + b; }; return문의 중괄호는 생략 불가!  
        System.out.println(c.cal(4, 3));  
  
        c = (a, b) -> a + b; 연산 결과가 남으면, 별도로 명시하지 않아도 반환 대상이 됨!  
        System.out.println(c.cal(4, 3));  
    }  
}
```



```
명령 프롬프트  
C:\WJavaStudy>java TwoParamAndReturn  
7  
C:\WJavaStudy>
```

매개변수가 있고 반환하는 람다식2

```
interface HowLong {  
    int len(String s);    // 값을 반환하는 메소드  
}  
  
class OneParamAndReturn {  
    public static void main(String[] args) {  
        HowLong hl = s -> s.length();  
        System.out.println(hl.len("I am so happy"));  
    }  
}
```



```
C:\JavaStudy>java OneParamAndReturn  
13  
C:\JavaStudy>
```

매개변수가 없는 람다식

```
interface Generator {  
    int rand(); // 매개변수 없는 메소드  
}  
  
class NoParamAndReturn {  
    public static void main(String[] args) {  
        Generator gen = () -> {  
            Random rand = new Random();  
            return rand.nextInt(50);  
        };  
  
        System.out.println(gen.rand());  
    }  
}
```

명령 프롬프트

C:\JavaStudy>java NoParamAndReturn
49

C:\JavaStudy>

함수형 인터페이스(Functional Interfaces)와 어노테이션

함수형 인터페이스: 추상 메소드가 딱 하나만 존재하는 인터페이스

`@FunctionalInterface`

`@FunctionalInterface`

함수형 인터페이스의 조건을 갖추었는지에 대한 검사를 컴파일러에게 요청!

```
interface Calculate {  
    int cal(int a, int b);  
}
```

`@FunctionalInterface`

```
interface Calculate {  
    int cal(int a, int b);  
    default int add(int a, int b) { return a + b; }  
    static int sub(int a, int b) { return a - b; }  
}
```

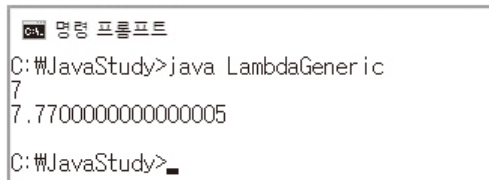
추상 메소드가 하나이니, 함수형 인터페이스 조건에 부합!

람다식과 제네릭

```
@FunctionalInterface
interface Calculate <T> {    // 제네릭 기반의 함수형 인터페이스
    T cal(T a, T b);
}

class LambdaGeneric {
    public static void main(String[] args) {
        Calculate<Integer> ci = (a, b) -> a + b;
        System.out.println(ci.cal(4, 3));

        Calculate<Double> cd = (a, b) -> a + b;
        System.out.println(cd.cal(4.32, 3.45));
    }
}
```



```
명령 프롬프트
C:\JavaStudy>java LambdaGeneric
7
7.7700000000000005
C:\JavaStudy>
```

인터페이스가 제네릭 기반이라 하여 특별히 신경 쓸 부분은 없다.

타입 인자가 전달이 되면(결정이 되면) 추상 메소드의 T는 결정이 되므로!

27-2. 정의되어 있는 함수형 인터페이스

미리 정의되어 있는 함수형 인터페이스

```
default boolean removeIf(Predicate<? super E> filter)
```

→ Collection<E> 인터페이스에 정의되어 있는 디폴트 메소드

Predicate 인터페이스의 추상 메소드는 다음과 같이 정의해 두었다.

```
boolean test(T t);
```

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

미리 정의해 두었으므로 Predicate라는 이름만으로 통한다!

대표 선수들!!!

<code>Predicate<T></code>	<code>boolean test(T t)</code>	전달 인자를 근거로 참 또는 거짓을 반환
<code>Supplier<T></code>	<code>T get()</code>	메소드 호출 시 무엇인가를 제공함
<code>Consumer<T></code>	<code>void accept(T t)</code>	무엇인가를 받아 들이기만 함
<code>Function<T, R></code>	<code>R apply(T t)</code>	입출력 출력이 있음(수학적으로는 함수)

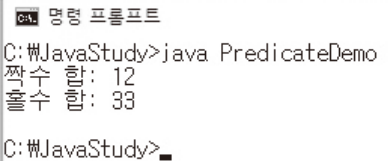
`java.util.function` 패키지로 묶여 있음!

Predicate<T>

```
boolean test(T t);
```

```
public static int sum(Predicate<Integer> p, List<Integer> lst) {  
    int s = 0;  
    for(int n : lst) {  
        if(p.test(n))  
            s += n;  
    }  
    return s;  
}
```

```
public static void main(String[] args) {  
    List<Integer> list = Arrays.asList(1, 5, 7, 9, 11, 12);  
    int s;  
  
    s = sum(n -> n%2 == 0, list);  
    System.out.println("짝수 합: " + s);  
  
    s = sum(n -> n%2 != 0, list);  
    System.out.println("홀수 합: " + s);  
}
```



```
명령 프롬프트  
C:\JavaStudy>java PredicateDemo  
짝수 합: 12  
홀수 합: 33  
C:\JavaStudy>
```

Predicate<T>를 구체화하고 다양화 한 인터페이스들

`IntPredicate`

`boolean test(int value)`

`LongPredicate`

`boolean test(long value)`

`DoublePredicate`

`boolean test(double value)`

`BiPredicate<T, U>`

`boolean test(T t, U u)`

```
public static int sum(Predicate<Integer> p, List<Integer> lst) { . . . }
```



```
public static int sum(IntPredicate p, List<Integer> lst) { . . . }
```

대체 가능! 그리고 박싱, 언박싱 과정이 필요 없어짐

Supplier<T>

T get();

```
public static List<Integer> makeIntList(Supplier<Integer> s, int n) {  
    List<Integer> list = new ArrayList<>();  
    for(int i = 0; i < n; i++)  
        list.add(s.get()); // 난수를 생성해 담는다.  
    return list;  
}
```

```
public static void main(String[] args) {  
    Supplier<Integer> spr = () -> {  
        Random rand = new Random();  
        return rand.nextInt(50);  
    };  
}
```

```
List<Integer> list = makeIntList(spr, 5);  
System.out.println(list);
```

```
list = makeIntList(spr, 10);  
System.out.println(list);
```

```
}
```

cmd 명령 프롬프트

```
C:\JavaStudy>java SupplierDemo  
[19, 31, 12, 40, 15]  
[47, 25, 20, 35, 37, 5, 11, 35, 47, 27]  
C:\JavaStudy>
```


Supplier<T>를 구체화 한 인터페이스들

IntSupplier

int getAsInt()

LongSupplier

long getAsLong()

DoubleSupplier

double getAsDouble()

BooleanSupplier

boolean getAsBoolean()

```
public static List<Integer> makeIntList(Supplier<Integer> s, int n) { . . . }
```



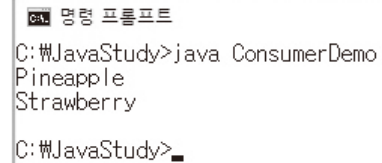
```
public static List<Integer> makeIntList(IntSupplier s, int n) { . . . }
```

대체 가능! 그리고 박싱, 언박싱 과정이 필요 없어짐

Consumer<T>

```
void accept(T t);
```

```
class ConsumerDemo {  
    public static void main(String[] args) {  
        Consumer<String> c = s -> System.out.println(s);  
        c.accept("Pineapple");      // 출력이라는 결과를 보임  
        c.accept("Strawberry");  
    }  
}
```



cmd 명령 프롬프트
C:\JavaStudy>java ConsumerDemo
Pineapple
Strawberry
C:\JavaStudy>

Consumer<T>를 구체화하고 다양화 한 인터페이스들

<code>IntConsumer</code>	<code>void accept(int value)</code>
<code>ObjIntConsumer<T></code>	<code>void accept(T t, int value)</code>
<code>LongConsumer</code>	<code>void accept(long value)</code>
<code>ObjLongConsumer<T></code>	<code>void accept(T t, long value)</code>
<code>DoubleConsumer</code>	<code>void accept(double value)</code>
<code>ObjDoubleConsumer<T></code>	<code>void accept(T t, double value)</code>
<code>BiConsumer<T, U></code>	<code>void accept(T t, U u)</code>

```
Consumer<String> c = s -> System.out.println(s);
```

```
ObjIntConsumer<String> c = (s, i) -> System.out.println(i + ". " + s);
```

Function<T, R>

R apply(T t);

```
class FunctionDemo {  
    public static void main(String[] args) {  
        Function<String, Integer> f = s -> s.length();  
        System.out.println(f.apply("Robot"));  
        System.out.println(f.apply("System"));  
    }  
}
```

명령 프롬프트

C:\JavaStudy>java FunctionDemo

5

6

C:\JavaStudy>

Function<T, R>을 구체화하고 다양화 한 인터페이스들

IntToDoubleFunction

DoubleToIntFunction

IntUnaryOperator

DoubleUnaryOperator

BiFunction<T, U, R>

IntFunction<R>

DoubleFunction<R>

ToIntFunction<T>

ToDoubleFunction<T>

ToIntBiFunction<T, U>

ToDoubleBiFunction<T, U>

double applyAsDouble(int value)

int applyAsInt(double value)

int applyAsInt(int operand)

double applyAsDouble(double operand)

R apply(T t, U u)

R apply(int value)

R apply(double value)

int applyAsInt(T value)

double applyAsDouble(T value)

int applyAsInt(T t, U u)

double applyAsDouble(T t, U u)

추가로!

Function<T, R>

R apply(T t)

BiFunction<T, U, R>

R apply(T t, U u)

앞서 소개한 인터페이스들

UnaryOperator<T>

T apply(T t)

BinaryOperator<T>

T apply(T t1, T t2)

T와 R을 일치시킨 인터페이스들

removeIf 메소드들 사용해 보자1

Collection<E> 인터페이스의 디폴트 메소드

```
default boolean removeIf(Predicate<? super E> filter)
```

ArrayList<Integer> 인스턴스의 removeIf

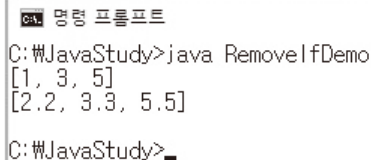
```
public boolean removeIf(Predicate<? super Integer> filter)
```

removeIf 메소드의 기능

“Removes all of the elements of this collection that satisfy the given predicate”

removeIf 메소드를 사용해 보자2

```
public static void main(String[] args) {  
    List<Integer> ls1 = Arrays.asList(1, -2, 3, -4, 5);  
    ls1 = new ArrayList<>(ls1);  
  
    List<Double> ls2 = Arrays.asList(-1.1, 2.2, 3.3, -4.4, 5.5);  
    ls2 = new ArrayList<>(ls2);  
  
    Predicate<Number> p = n -> n.doubleValue() < 0.0;    // 삭제의 조건  
    ls1.removeIf(p);    // List<Integer> 인스턴스에 전달  
    ls2.removeIf(p);    // List<Double> 인스턴스에 전달  
  
    System.out.println(ls1);  
    System.out.println(ls2);  
}
```



```
C:\JavaStudy>java RemoveIfDemo  
[1, 3, 5]  
[2.2, 3.3, 5.5]  
C:\JavaStudy>
```