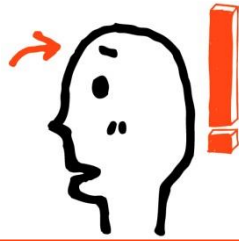




# Android Java

22장

람다와 함수형 인터페이스

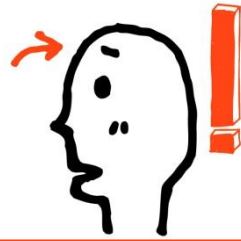


# 람다의 이해1

```
interface Printable {  
    void print(String s);  
}  
  
class Printer implements Printable {  
    public void print(String s) {  
        System.out.println(s);  
    }  
}  
  
class Lambda1 {  
    public static void main(String[] args) {  
        Printable prn = new Printer();  
        prn.print("What is Lambda?");  
    }  
}
```

```
interface Printable {  
    void print(String s);  
}  
  
class Lambda2 {  
    public static void main(String[] args) {  
        Printable prn = new Printable() { //익명 클래스  
            public void print(String s) {  
                System.out.println(s);  
            }  
        };  
  
        prn.print("What is Lambda?");  
    }  
}
```

아직 람다 등장 안 했음!



# 람다의 이해2

```
interface Printable {  
    void print(String s);  
}
```

```
class Lambda2 {  
    public static void main(String[] args) {  
        Printable prn = new Printer() {  
            public void print(String s) {  
                System.out.println(s);  
            }  
        };  
  
        prn.print("What is Lambda?");  
    }  
}
```

## 드디어 람다 등장

```
interface Printable { // 추상 메소드가 하나인 인터페이스  
    void print(String s);  
}  
  
class Lambda3 {  
    public static void main(String[] args) {  
        Printable prn = (s) -> { System.out.println(s); };  
        prn.print("What is Lambda?");  
    }  
}
```




# 람다의 이해3 : 생략 가능한 것을 지워보자.

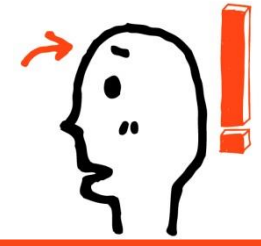
```
Printable prn = new Printable() {  
    public void print(String s) {  
        System.out.println(s);  
    }  
};
```

```
interface Printable {  
    void print(String s);  
}
```

prn이 Printable형 참조변수이니 = 의 왼편에는 new가 당연히 올 것이고,  
메소드 정의가 온 것을 보니, 익명 클래스를 기반으로 보건대 이는 인스턴스 생성이야!



```
Printable prn = new Printable() {  
    public void print(String s) {  
        System.out.println(s);  
    }  
};
```



# 람다의 이해4

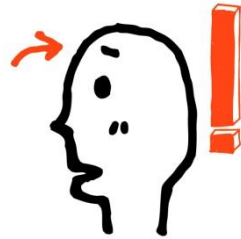
```
Printable prn =  
    public void print(String s) {  
        System.out.println(s);  
    } ;
```

```
interface Printable {  
    void print(String s);  
}
```

Printable 인터페이스에 있는 메소드 그거 public void print(String s)니 뻔하지 뭐!



```
Printable prn =  
    public void print(String s) {  
        System.out.println(s);  
    } ;
```



# 람다의 이해5

```
Printable prn = { System.out.println(s); };
```

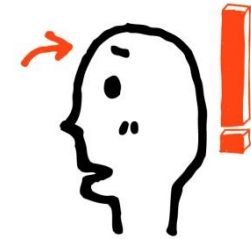
```
interface Printable {  
    void print(String s);  
}
```

컴파일러가 저 `s`가 매개변수라고 판단해 주길 바라는 것은 무리이니까!

```
Printable prn = (String s) -> { System.out.println(s); };   완성된 람다식!
```

`s`가 `String`형 임은 `Printable` 인터페이스 보면 알 수 있지 않아?

```
Printable prn = (s) -> { System.out.println(s); };   조금 더 줄이면!
```



# 람다의 이해6

## 함수적 스타일의 람다식 작성법 정리

```
(타입 매개변수, ...) -> { 실행문; ... }
```

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

매개 타입은 런타임시에 대입값 따라 자동 인식 → 생략 가능

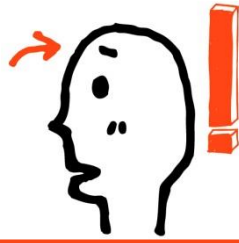
하나의 매개변수만 있을 경우에는 괄호( ) 생략 가능

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



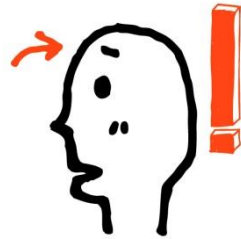
# 람다식의 인자 전달

```
interface Printable {  
    void print(String s);  
}
```

```
Printable prn = (s) -> { System.out.println(s); };
```

```
method((s) -> System.out.println(s));    // void method(Printable prn) {...}
```





# 인스턴스보다 기능 하나가 필요한 상황을 위한 람다

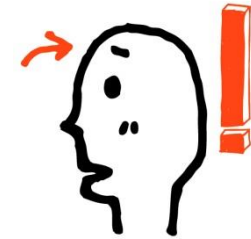
```
class SLenComp implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length()
    }
}

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:\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 둘인 람다식

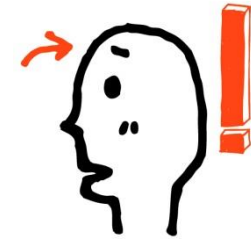
```
interface Calculate {
    void cal(int a, int b); // 매개변수 둘, 반환형 void
}

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
    }
}
```

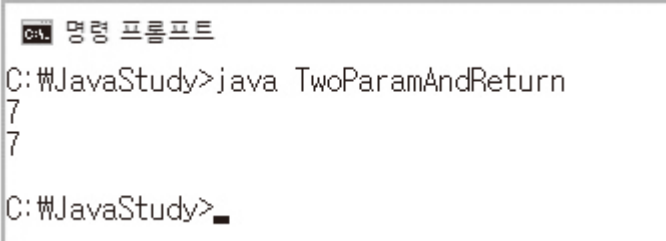
```
C:\> 명령 프롬프트
C:\JavaStudy>java TwoParamNoReturn
7
1
12
C:\JavaStudy>
```



# 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));
```



# 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:\> 명령 프롬프트
C:\JavaStudy>java OneParamAndReturn
13
C:\JavaStudy>_
```



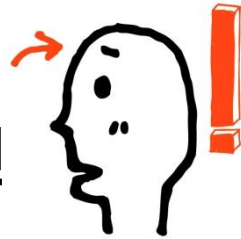
# 매개변수가 없는 람다식

```
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());
    }
}
```

```
cmd 명령 프롬프트
C:\JavaStudy>java NoParamAndReturn
49
C:\JavaStudy>
```



# 함수형 인터페이스(Functional Interfaces)와 어노테이션

함수형 인터페이스: 추상 메소드가 딱 하나만 존재하는 인터페이스

**@FunctionalInterface**

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

@FunctionalInterface

함수형 인터페이스의 조건을 갖추었는지에 대한 검사를 컴파일러에게 요청!

**@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);
}

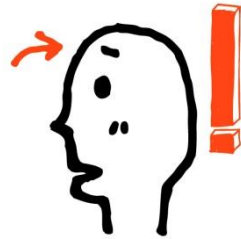
class LambdaGeneric {
    public static void main(String[] args) {
        Calculate<Integer> ci = (a, b) -> a + b;
        System.out.println(ci.cal(4, 3));

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

```
cmd 명령 프롬프트
C:\WJavaStudy>java LambdaGeneric
7
7.7700000000000005
C:\WJavaStudy>
```

인터페이스가 제네릭 기반이라 하여 특별히 신경 쓸 부분은 없다.  
타입 인자가 전달이 되면(결정이 되면) 추상 메소드의 T는 결정이 되므로!

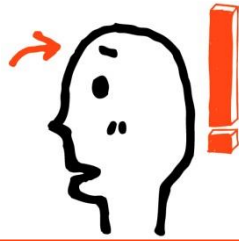




# 람다식 작성하기

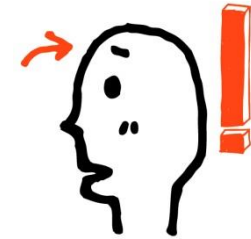
문제 1. 아래 코드에서 주석에 명시된 연산의 결과를 출력하기 위한 `calAndShow` 메소드의 호출문을 람다식을 기반으로 작성해보자.

```
interface Calculate<T> {  
    T cal(T a, T b);  
}  
  
class CalculatorDemo {  
    public static <T> void calAndShow(Calculate<T> op, T n1, T n2) {  
        T r = op.cal(n1, n2);  
        System.out.println(r);  
    }  
  
    public static void main(String[] args) {  
        // 3 + 4  
        // 2.5 + 7.1  
        // 4 - 2  
        // 4.9 - 3.2  
    }  
}
```



# 람다식 작성하기

문제 2. 본 Chapter의 첫번째 예제인 `SLenComparator.java`를 람다식 기반으로 수정해보자. 수정결과에서는 클래스 `SLenComp`의 정의가 지워져야 한다.



# 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);
```

미리 정의해 두었으므로 Predicate라는 이름만으로 통한다!

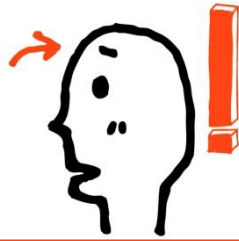
```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```



# 대표 선수들!!!

Predicate<T>	<code>boolean test(T t)</code> 전달 인자를 근거로 참 또는 거짓을 반환
Supplier<T>	<code>T get()</code> 메소드 호출 시 무엇인가를 제공함
Consumer<T>	<code>void accept(T t)</code> 무엇인가를 받아 들이기만 함
Function<T, R>	<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);  
}
```

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



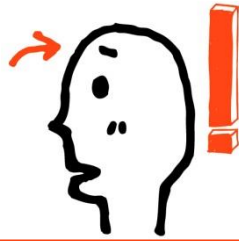
# 람다식 작성하기

문제 3. 아래의 코드에서 주석으로 표시된 내용의 출력을 보이도록 show 메소드의 몸체를 채워보자.

```
class PredicateShow {
    public static <T> void show(Predicate<T> p, List<T> lst) {
        //채워 넣을 부분
    }

    public static void main(String[] args) {
        List<Integer> lst1 = Arrays.asList(1, 3, 8, 10, 11);
        System.out.println("홀수 출력");
        show(n -> n%2 != 0, lst1);    // 홀수만 출력

        List<Double> lst2 = Arrays.asList(-1.2, 3.5, -2.4, 9.5);
        System.out.println("0.0 보다 큰 수 출력");    // 0.0 보다 큰 수 출력
        show(n -> n > 0.0, lst2);
    }
}
```



# Predicate<T>를 구체화하고 다양화 한 인터페이스들

<code>IntPredicate</code>	<code>boolean test(int value)</code>
<code>LongPredicate</code>	<code>boolean test(long value)</code>
<code>DoublePredicate</code>	<code>boolean test(double value)</code>
<code>BiPredicate&lt;T, U&gt;</code>	<code>boolean test(T t, U u)</code>

```
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);
}
```

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





# 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

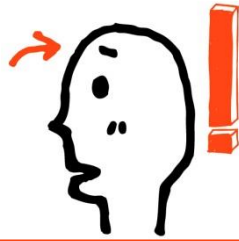


# Consumer<T>

```
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");  
    }  
}
```

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

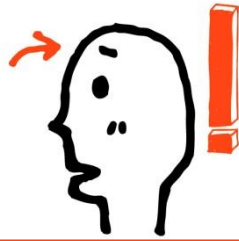


# Consumer<T>를 구체화하고 다양화 한 인터페이스들

<code>IntConsumer</code>	<code>void accept(int value)</code>
<code>ObjIntConsumer&lt;T&gt;</code>	<code>void accept(T t, int value)</code>
<code>LongConsumer</code>	<code>void accept(long value)</code>
<code>ObjLongConsumer&lt;T&gt;</code>	<code>void accept(T t, long value)</code>
<code>DoubleConsumer</code>	<code>void accept(double value)</code>
<code>ObjDoubleConsumer&lt;T&gt;</code>	<code>void accept(T t, double value)</code>
<code>BiConsumer&lt;T, U&gt;</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);
```

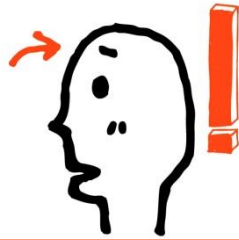


# Function<T, R>

```
R apply(T t);
```

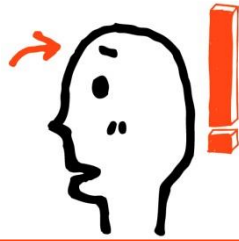
```
class FunctionDemo {  
    public static void main(String[] args) {  
        Function<String, Integer> f = s -> s.length();  
        System.out.println(f.apply("Robot"));  
        System.out.println(f.apply("System"));  
    }  
}
```

```
C:\ 명령 프롬프트  
C:\WJavaStudy>java FunctionDemo  
5  
6  
C:\WJavaStudy>_
```



# Function<T, R>을 구체화하고 다양화 한 인터페이스들

<code>IntToDoubleFunction</code>	<code>double applyAsDouble(int value)</code>
<code>DoubleToIntFunction</code>	<code>int applyAsInt(double value)</code>
<code>IntUnaryOperator</code>	<code>int applyAsInt(int operand)</code>
<code>DoubleUnaryOperator</code>	<code>double applyAsDouble(double operand)</code>
<code>BiFunction&lt;T, U, R&gt;</code>	<code>R apply(T t, U u)</code>
<code>IntFunction&lt;R&gt;</code>	<code>R apply(int value)</code>
<code>DoubleFunction&lt;R&gt;</code>	<code>R apply(double value)</code>
<code>ToIntFunction&lt;T&gt;</code>	<code>int applyAsInt(T value)</code>
<code>ToDoubleFunction&lt;T&gt;</code>	<code>double applyAsDouble(T value)</code>
<code>ToIntBiFunction&lt;T, U&gt;</code>	<code>int applyAsInt(T t, U u)</code>
<code>ToDoubleBiFunction&lt;T, U&gt;</code>	<code>double applyAsDouble(T t, U u)</code>



# 추가로!

Function<T, R>

R apply(T t)

BiFunction<T, U, R>

R apply(T t, U u)

앞서 소개한 인터페이스들

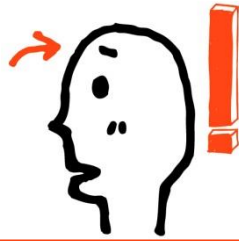
UnaryOperator<T>

T apply(T t)

BinaryOperator<T>

T apply(T t1, T t2)

T와 R을 일치시킨 인터페이스들



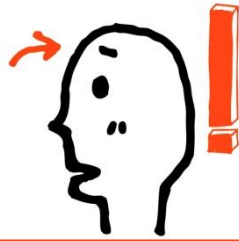
# removeIf 메소드를 사용해 보자1

Collection<E> 인터페이스의 디폴트 메소드

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

ArrayList<Integer> 인스턴스의 removeIf

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

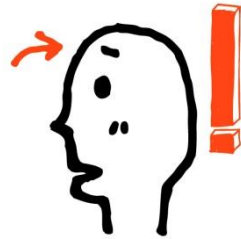


## removeIf 메소드를 사용해 보자2

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





# 메소드 참조의 4가지 유형과 메소드 참조의 장점

- static 메소드의 참조
- 참조변수를 통한 인스턴스 메소드 참조
- 클래스 이름을 통한 인스턴스 메소드 참조
- 생성자 참조

기본적으로 람다식보다 조금 더 코드를 단순하게 할 수 있다는 장점이 있다.  
일부 람다식을 메소드 참조로 대신하게 할 수 있다.

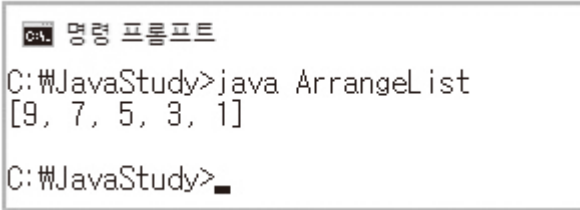


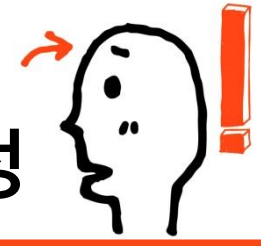
# static 메소드의 참조: 람다식 기반 예제

Collections 클래스의 reverse 메소드 기반 예제

```
public static void reverse(List<?> list) // 저장 순서를 뒤집는다.
```

```
class ArrangeList {  
    public static void main(String[] args) {  
        List<Integer> ls = Arrays.asList(1, 3, 5, 7, 9);  
        ls = new ArrayList<>(ls);  
  
        Consumer<List<Integer>> c = l -> Collections.reverse(l); // reverse 메소드 호출 중심의 람다식  
        c.accept(ls); // 순서 뒤집기 진행  
        System.out.println(ls); // 출력  
    }  
}
```





# static 메소드의 참조: 메소드 참조 기반으로 수정

```
Consumer<T>    void accept(T t)
```

```
Consumer<List<Integer>> c = l -> Collections.reverse(l);
```

```
→ Consumer<List<Integer>> c = Collections::reverse;
```

accept 메소드 호출 시 전달되는 인자를 reverse 메소드를 호출하면서 그대로 전달한다는  
약속에 근거한 수정

```
void accept(T t)
void reverse(List<?> list)
accept로 전달된 것 reverse로 그대로. . .
```

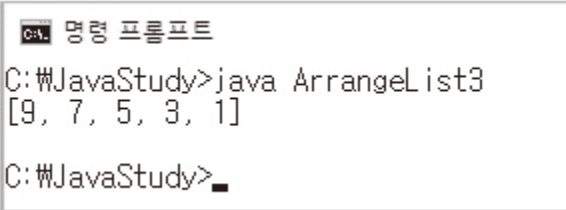


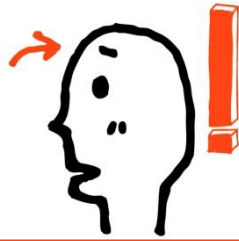
# 인스턴스 메소드 참조1 : effectively final

```
class JustSort {
    public void sort(List<?> lst) { // 인스턴스 메소드
        Collections.reverse(lst);
    }
}

class ArrangeList3 {
    public static void main(String[] args) {
        List<Integer> ls = Arrays.asList(1, 3, 5, 7, 9);
        ls = new ArrayList<>(ls);
        JustSort js = new JustSort(); // js는 effectively final

        Consumer<List<Integer>> c = e -> js.sort(e); // 람다식 기반
        c.accept(ls);
        System.out.println(ls);
    }
}
```





# 인스턴스 메소드 참조1 : 인스턴스 존재 상황에서 참조

```
class JustSort {
    public void sort(List<?> lst) { // 인스턴스 메소드
        Collections.reverse(lst);
    }
}
```

`Consumer<T>`      `void accept(T t)`

`void accept(T t)`

`void sort(List<?> list)`

```
class ArrangeList3 {
    public static void main(String[] args) {
        List<Integer> ls = Arrays.asList(1, 3, 5, 7, 9);
        ls = new ArrayList<>(ls);
        JustSort js = new JustSort(); // js는 effective final
```

`accept`로 전달된 것 `sort`로 그대로. . .

```
Consumer<List<Integer>> c = e -> js.sort(e); // 람다식 기반
```

```
→ Consumer<List<Integer>> c = js::sort;
```

```
. . .
```

```
}
```

```
}
```

명령 프롬프트

```
C:\JavaStudy>java ArrangeList3
[9, 7, 5, 3, 1]
```

```
C:\JavaStudy>_
```

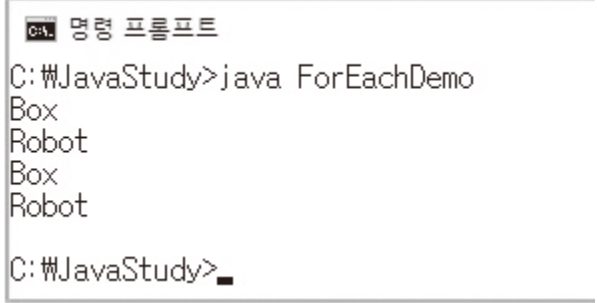


# 인스턴스 메소드 참조1 : forEach 메소드

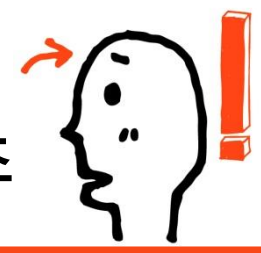
```
class ForEachDemo {
    public static void main(String[] args) {
        List<String> ls = Arrays.asList("Box", "Robot");
        ls.forEach(s -> System.out.println(s));    // 람다식 기반
        ls.forEach(System.out::println);    // 메소드 참조 기반
    }
}
```

Consumer<T>      void accept(T t)

accept로 전달된 것 그대로 println으로. . .



```
default void forEach(Consumer<? super T> action) {    // Iterable<T>의 디폴트 메소드
    for (T t : this)    // this는 이 메소드가 속한 컬렉션 인스턴스를 의미함
        action.accept(t);    // 모든 저장된 데이터들에 대해 이 문장 반복
}
```



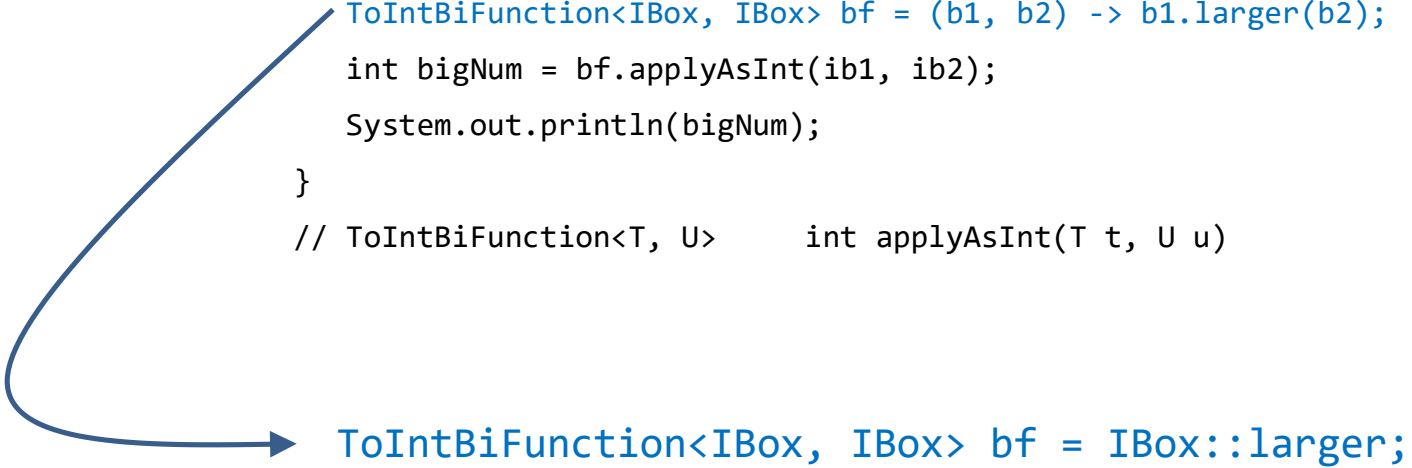
# 인스턴스 메소드 참조2 : 인스턴스 없이 인스턴스 메소드 참조

```
class IBox {
    private int n;
    public IBox(int i) { n = i; }
    public int larger(IBox b) {
        if(n > b.n)
            return n;
        else
            return b.n;
    }
}

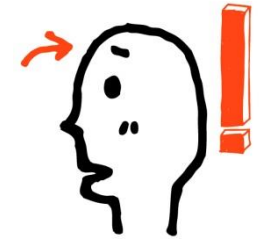
ToIntBiFunction<T,U>
public static void main(String[] args) {
    IBox ib1 = new IBox(5);
    IBox ib2 = new IBox(7);

    // 두 상자에 저장된 값 비교하여 더 큰 값 반환
    ToIntBiFunction<IBox, IBox> bf = (b1, b2) -> b1.larger(b2);
    int bigNum = bf.applyAsInt(ib1, ib2);
    System.out.println(bigNum);
}

// ToIntBiFunction<T, U>      int applyAsInt(T t, U u)
```



약속에 근거한 줄인 표현



# 메소드 참조 작성하기

- 다음 예제를 메소드 참조 방식으로 수정해보자.

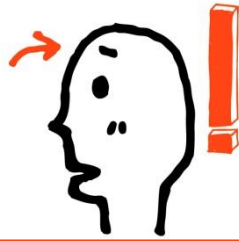
```
class StrIgnoreCaseComp {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("robot");
        list.add("Lambda");
        list.add("box");

        Collections.sort(list, (s1, s2) -> s1.compareToIgnoreCase(s2));
        // Collections.sort(list, String::compareToIgnoreCase);
        System.out.println(list);
    }
}
```

참고로 Collections.sort 메소드가 다음과 같으니  
Public static <T> void sort(List<T> list, Comparaoatr<? super T> c)

다음 문장을 메소드 참조 기반으로 수정한다고 생각하면 편하다.  
Comparator<? super ?> c = (s1, s2) -> -> s1.compareToIgnoreCase(s2))





# 생성자 참조

```
class StringMaker {  
    public static void main(String[] args) {  
        Function<char[], String> f = ar -> {  
            return new String(ar);  
        };  
        char[] src = {'R', 'o', 'b', 'o', 't'};  
        String str = f.apply(src);  
        System.out.println(str);  
    }  
}  
// Function<T, R>    R apply(T t)
```

```
Function<char[], String> f = ar -> {  
    return new String(ar);  
};
```



```
Function<char[], String> f = ar -> new String(ar);
```



```
Function<char[], String> f = String::new;
```

# THANK YOU

---

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술