

람다식과 자바API의 함수적 인터페이스

람다식의 개념 및 기본 문법

람다식(Lambda Expression)

☞ 람다식이란

- 1 - 자바에서 **함수적 프로그래밍** 지원 기법
- 코드의 간결화 및 **병렬처리**에 강함 (**Collection API 성능 효과적 개선** (Stream))

TIP

4

- Java는 객체지향프로그램으로서 모든 함수는 클래스/인터페이스 **내부에만 존재** 가능
(클래스 내부의 함수 = 메서드)

☞ 람다식 이해를 위한 **기본 용어**의 정리

- 2 - **함수(function)** : 기능, 동작을 정의

```
void abc(){  
    //기능 및 동작  
}
```

- 3 - **메서드(method)** : 클래스 또는 인터페이스 **내부에서 정의된 함수**

```
class A{  
    void abc(){  
        //기능 및 동작  
    }  
}
```

- 5 - **함수형 인터페이스(functional interface)**

: 내부에 **단 1개의 추상메서드**만 존재하는 인터페이스

```
interface A{  
    public abstract void abc();  
}
```

람다식(Lambda Expression)

☞ 본래의미의 함수적 프로그래밍과 객체지향형의 개념적 비교

- 본래 의미의 함수적 프로그래밍에서의 함수 사용 (자주 사용하는 기능 구현)

1 STEP#1 함수정의/구현

함수 정의 및 구현

```
void abc(){  
    //메서드(함수) 정의 및 구현  
}
```

2 STEP#2 함수 호출

함수이름 바로 사용

```
abc()
```



자바는 새로운 함수 문법을 정의한 것이 아니라 이미 있는 인터페이스를 빌어 람다식을 표현

메서드의 정의 → 람다식 표현
메서드의 호출 → 참조변수.메서드이름 (객체지향과 동일)

- 객체지향형 프로그래밍에서의 메서드(함수=기능) 사용

4 STEP#1 함수정의/구현

인터페이스내 메서드(함수) 정의
클래스 내 메서드(함수) 구현

```
class A {  
    void abc(){  
        //메서드(함수) 정의 및 구현  
    }  
}
```

5 STEP#2 객체생성

타입 참조변수 = 객체 생성

```
A a = new A();
```

6 STEP#3 객체 내부의 메서드 호출

참조변수.메서드이름

```
a.abc();
```

람다식(Lambda Expression)

👉 객체지향형과 자바의 함수적 프로그래밍의 문법적 비교

- 객체지향형 프로그래밍 메서드(함수=기능) 사용

1 CASE#1

```
interface A {  
    void abc();  
}
```

```
class B implements A {  
    public void abc() {  
        //메서드 내용  
    }  
}
```

```
A a = new B();
```

```
a.abc();
```

2 CASE#2

```
interface A {  
    void abc();  
}
```

익명이너클래스

```
A a = new A() {  
    void abc() {  
        //메서드 내용  
    }  
};
```

```
a.abc();
```

메서드(함수)

- 자바의 함수적 프로그래밍(람다식) 함수(=기능) 사용

3 CASE#3

```
interface A {  
    void abc();  
}
```

```
A a = () -> {  
    //메서드 내용  
};
```

```
a.abc();
```

람다식

4

문법적인 의미만 고려하면 람다식은 익명이너클래스의 약식 표현


5

Quiz

모든 인터페이스의 구현 메서드는 람다식으로 변환할 수 있을까? **NO!**

함수적 인터페이스의 메서드만 람다식 표현 가능

람다식(Lambda Expression)

1  메서드(함수) → 람다식 변환 방법
인터페이스의 구현 메서드

메서드(함수)

```
리턴타입 메서드 이름(입력매개변수) {  
    //메서드 내용  
}
```

2

람다식

```
(입력매개변수) → {  
    //메서드 내용  
}
```

람다식 기호

메서드(함수)

3

```
void method1() {  
    System.out.println(3);  
}
```

4

람다식

```
() → {  
    System.out.println(3);  
}
```

```
() → {System.out.println(3);}
```

```
void method2(int a) {  
    System.out.println(a);  
}
```

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

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

```
int method3() {  
    return 5;  
}
```

```
() → {  
    return 5;  
}
```

```
() → {return 5;}
```

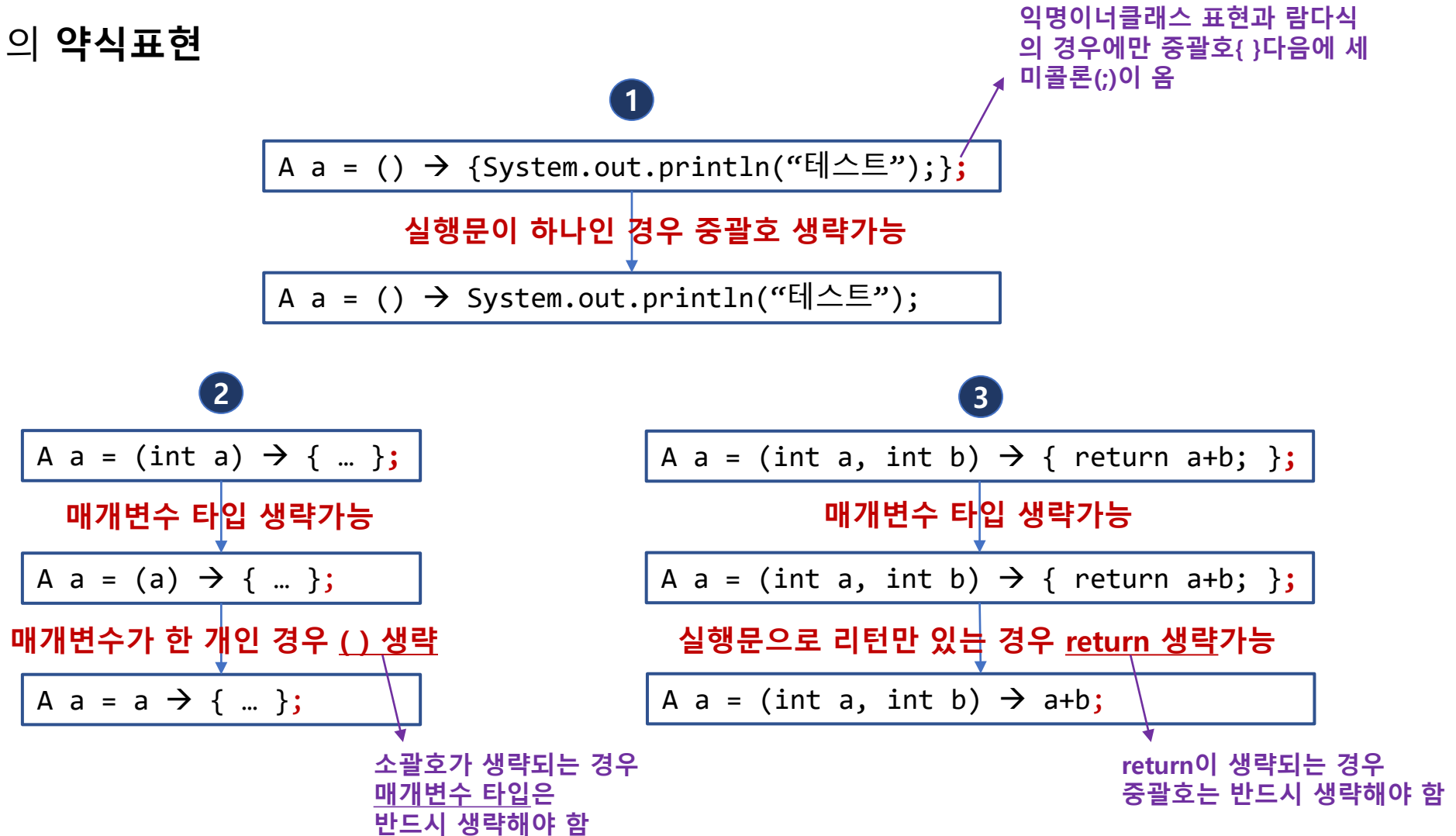
```
double method4(int a, double b) {  
    return a+b;  
}
```

```
(int a, double b) → {  
    return a+b;  
}
```

```
(int a, double b) → {return a+b;}
```

람다식(Lambda Expression)

☞ 람다식의 약식표현



The End

람다식의 세 가지 활용

람다식(Lambda Expression)

☞ 람다식의 활용

내부에 한 개의 메서드만 포함한 인터페이스



- 1 - **활용#1.** 익명이너클래스 내 구현 메서드의 **약식(람다식) 표현** (함수형 인터페이스만 가능)
- 2 - **활용#2.** 메서드 참조 (인스턴스 메서드 참조 Type1, 정적 메서드 참조, 인스턴스 메서드 참조 Type2)
- 3 - **활용#3.** 생성자 참조 (배열 생성자 참조, 클래스 생성자 참조)

람다식(Lambda Expression)

☞ 람다식의 활용

- ① **활용#1** 익명이너클래스 내 구현 메서드의 약식(람다식) 표현 (함수형 인터페이스만 가능)

② 함수형 인터페이스



```
interface A {  
    void method1();  
}
```

③ 익명 이너클래스 활용



```
A a = new A(){  
    public void method1(){  
        ...  
    }  
}
```

④ 람다식 활용



```
A a = () -> { ... };
```

EX 1

EX 2

```
interface A {  
    void method2(int a);  
}
```

```
A a = new A(){  
    public void method2(int a){  
        ...  
    }  
}
```

```
A a = (int a) -> { ... };
```

람다식(Lambda Expression)

☞ 람다식의 활용

활용#1 익명이너클래스 내 구현 메서드의 약식(람다식) 표현 (함수형 인터페이스만 가능)

EX 3

1 함수형
인터페이스

```
interface A {  
    int method3();  
}
```

2 익명
이너클래스 활용

```
A a = new A(){  
    public int method3(){  
        return ...  
    }  
}
```

3 람다식 활용

```
A a = () -> { return ... };
```

EX 4

```
interface A {  
    double method4(int a, double b);  
}
```

```
A a = new A(){  
    public double method4(int a, double b){  
        return ...  
    }  
}
```

```
A a = (int a, double b) -> { return ... };
```

람다식(Lambda Expression)

☞ 람다식의 활용

활용#1 함수의 약식(람다식) 표현
(함수형 인터페이스만 가능)

```
1 interface A{ //입력x, 출력x
    void method1();
}
interface B{ //입력0, 출력x
    void method2(int a);
}
interface C{ //입력x, 출력0
    int method3();
}
interface D{ //입력0, 출력0
    double method4(int a, double b);
}
```

```
2 //#1. 입력x, 출력x의 함수
//@1-1. 익명이너클래스 표현
A a1 = new A() {
    @Override
    public void method1() {
        System.out.println("입력x, 출력x의 함수");
    }
};

//@1-2. 람다식 표현
A a2 = ()->{System.out.println("입력x, 출력x의 함수");};
A a3 = ()->System.out.println("입력x, 출력x의 함수");

3 //#2. //입력0, 출력x의 함수
//@2-1. 익명이너클래스 표현
B b1 = new B() {
    public void method2(int a) {
        System.out.println(a);
    }
};

//@2-2. 람다식 표현
B b2 = (int a)->{System.out.println(a);};
B b3 = (a)->{System.out.println(a);};
B b4 = (a)->System.out.println(a);
B b5 = a->System.out.println(a);
```

람다식(Lambda Expression)

☞ 람다식의 활용

활용#1 함수의 약식(람다식) 표현
(함수형 인터페이스만 가능)

```
interface A{ //입력x, 출력x
    void method1();
}
interface B{ //입력0, 출력x
    void method2(int a);
}
interface C{ //입력x, 출력0
    int method3();
}
interface D{ //입력0, 출력0
    double method4(int a, double b);
}
```

//#3. //입력x, 출력0의 함수
//@3-1. 익명이너클래스 표현

1

```
C c1 = new C() {
    @Override
    public int method3() {
        return 4;
    }
};
```

//@3-2. 람다식 표현

```
C c2 = ()-> {return 4;};
C c3 = ()->4;
```

//#4. //입력x, 출력0의 함수
//@4-1. 익명이너클래스 표현

2

```
D d1 = new D() {
    @Override
    public double method4(int a, double b) {
        return a+b;
    }
};
```

//@4-2. 람다식 표현

```
D d2 = (int a, double b) -> {return a+b;};
D d3 = (a, b) -> {return a+b;};
D d4 = (a, b) -> a+b;
```

람다식(Lambda Expression)

6 인스턴스메서드 참조 1

클래스**객체**::인스턴스메서드이름

☞ 람다식의 활용

1

활용#2-1. 인스턴스 메서드 참조 Type1 → 이미 정의된 인스턴스 메서드 참조

EX 1

2

```
interface A {  
    void abc();  
}
```

3

```
class B {  
    void bcd(){  
        System.out.println("메서드");  
    }  
}
```

4

```
A a = new A(){  
    public void abc(){  
        B b = new B();  
        b.bcd();  
    }  
};
```

5

```
A a = () -> {  
    B b = new B();  
    b.bcd();  
};
```

abc() 메서드 = B객체의 bcd() 메서드

7

```
B b = new B();  
A a = b::bcd;
```

TIP

8

- 인스턴스 메서드 참조 Type1을 위해서는 리턴타입과 매개변수가 동일하여야 함

람다식(Lambda Expression)

인스턴스메서드 참조 1

클래스객체::인스턴스메서드이름

☞ 람다식의 활용

활용#2-1. 인스턴스 메서드 참조 Type1 → 이미 정의된 인스턴스 메서드 참조

EX 2

1

```
interface A {  
    void abc(int k);  
}
```

2

```
A a = new A(){  
    public void abc(int k){  
        System.out.println(k);  
    }  
};
```

3

```
A a = (k)→{  
    System.out.println(k);  
};
```

abc(...) 메서드 = System.out.println(...) 메서드

4

```
A a = System.out::println;
```

TIP

- 인스턴스 메서드 참조 Type1을 위해서는 리턴타입과 매개변수가 동일하여야 함

람다식(Lambda Expression)

2

정적메서드 참조

클래스**이름**::정적메서드이름

☞ 람다식의 활용

1

활용#2-2. 정적 메서드 참조 → 이미 정의된 정적 메서드 참조

EX 1

3

```
interface A {  
    void abc();  
}
```

4

```
class B {  
    static void bcd(){  
        System.out.println("메서드");  
    }  
}
```

5

```
A a = new A(){  
    public void abc(){  
        B.bcd();  
    }  
};
```

6

```
A a = () -> {  
    B.bcd();  
};
```

abc() 메서드 = B.bcd() 정적메서드

7

```
A a = B::bcd;
```

TIP

8

- 정적 메서드 참조를 위해서는 리턴타입과 매개변수가 동일하여야 함

람다식(Lambda Expression)

5 인스턴스메서드 참조 2

클래스이름::인스턴스메서드이름

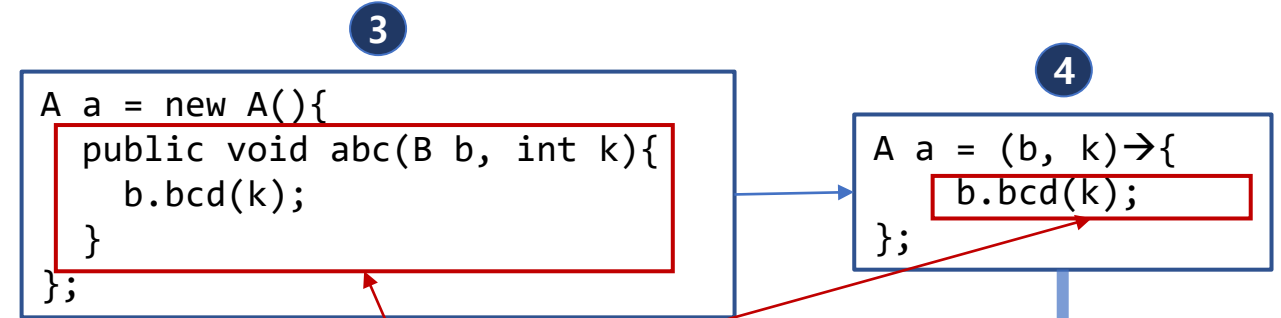
☞ 람다식의 활용

1
활용#2-3. 인스턴스 메서드 참조 Type2
→ 첫번째 매개변수로 전달된 객체의 메서드를 참조하는 경우

EX 1

2

```
interface A {  
    void abc(B b, int k);  
}  
  
class B {  
    void bcd(int k){  
        System.out.println(k);  
    }  
}
```



abc(B b, int k) 메서드 = b.bcd(k) 메서드

7 TIP
- 인스턴스메서드 참조 Type2를 위해서는 인터페이스 메서드매개변수가 참조 메서드 매개 변수보다 **하나 많음**

6

```
A a = B::bcd;
```

람다식(Lambda Expression)

☞ 람다식의 활용

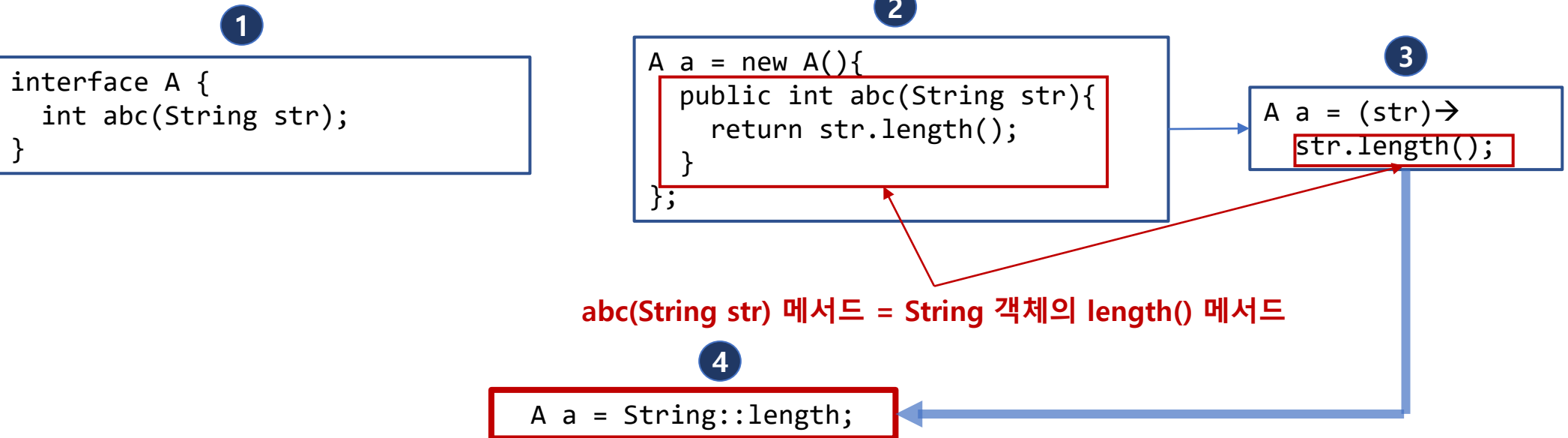
활용#2-3. 인스턴스 메서드 참조 Type2

→ 첫번째 매개변수로 전달된 객체의 메서드를 참조하는 경우

인스턴스메서드 참조 2

클래스이름::인스턴스메서드이름

EX 2



TIP

- 인스턴스메서드 참조 Type2를 위해서는 인터페이스 메서드매개변수가 참조 메서드 매개 변수보다 하나 많음

람다식(Lambda Expression)

☞ 람다식의 활용

① **활용#3-1. 배열 생성자 참조** → 배열의 **new** 생성자를 참조하는 경우
→ interface 메서드의 **리턴타입=배열객체**

⑤ 배열 생성자 참조

배열타입::new

EX 1

②
interface A {
 int[] abc(int **len**);
}

③
A a = new A(){
 public **int[]** abc(int len){
 return **new** int[len];
 }
};

④
A a = (len)→
 new int[len];

abc(int len) 메서드 = new int[len]

TIP

⑦

- 배열 생성자 참조를 위해서는 인터페이스 메서드의 매개변수로 배열의 길이를 전달

⑥
A a = int[]::new;

람다식(Lambda Expression)

☞ 람다식의 활용

활용#3-2. 클래스 생성자 참조 → 클래스의 **new** 생성자를 참조하는 경우
→ interface 메서드의 **리턴타입=클래스 객체**

EX 1

2
interface A {
 B abc();
}

```
class B {  
    B(){//첫번째 생성자}  
    B(int k){//두번째 생성자}  
}
```

3
A a = new A(){
 public B abc(){
 return **new** B();
 }
};

4
A a = () →
 new B();

abc() 메서드 = new B()

6
A a = B::new;

첫번째 생성자를 이용한 객체 생성

TIP

7

- 클래스 생성자 참조를 위해서는 인터페이스 메서드의 **매개변수에 따라 생성자 선택**

클래스 생성자 참조

5

클래스**이름**::new

람다식(Lambda Expression)

☞ 람다식의 활용

활용#3-2. 클래스 생성자 참조 → 클래스의 **new** 생성자를 참조하는 경우
→ interface 메서드의 **리턴타입=클래스 객체**

클래스 생성자 참조

클래스이름::new

EX 2

1
interface A {
 B abc(int k);
}

class B {
 B(){//첫번째 생성자}
 B(int k){//두번째 생성자}
}

2
A a = new A(){
 B abc(int k){
 return new B(k);
 }
};

3
A a = (k) →
new B(k);

abc(int k) 메서드 = new B(k)

4
A a = B::new;

두번째 생성자를 이용한 객체 생성

TIP

5

- 클래스 생성자 참조를 위해서는 인터페이스 메서드의 매개변수에 따라 생성자 선택

The End

자바 API의 함수형 인터페이스

자바API의 함수적 인터페이스

1

개념적 의미:

자주 사용하는 기능을 정의해 놓는 함수 제공 (X)

자주 사용하는 기능을 정의할 수 있는 함수 제공 (O)

☞ 메서드의 매개변수에 사용되는 함수적 인터페이스

2

표준형 함수적 인터페이스

Consumer<T>

Supplier<T>

Predicate<T>

Function<T, R>

3

확장형 함수적 인터페이스

BooleanSupplier

IntConsumer

IntSupplier

IntPredicate

IntFunction<R>

LongConsumer

LongSupplier

LongPredicate

LongFunction<R>

DoubleConsumer

DoubleSupplier

DoublePredicate

DoubleFunction<R>

BiConsumer<T,U>

BiPredicate<T,U>

BiFunction<T,U,R>

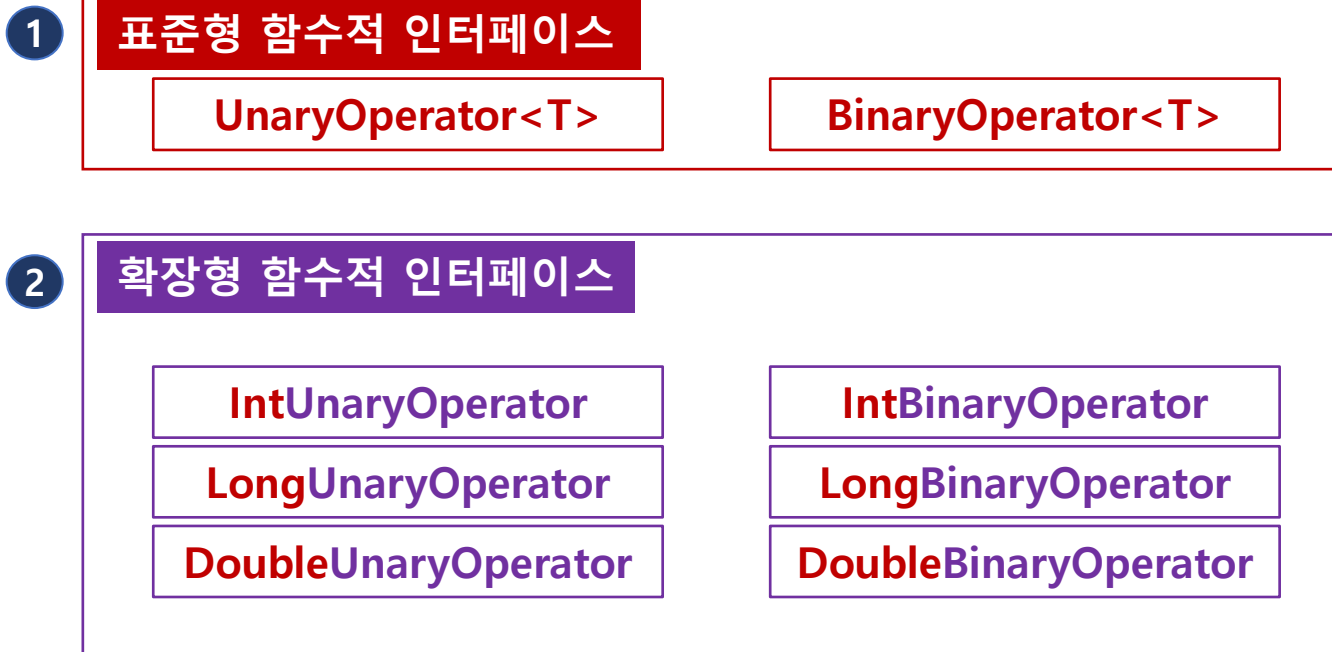
ToIntFunction<T>

ToLongFunction<T>

ToDoubleFunction<T>

자바API의 함수적 인터페이스

☞ 메서드의 매개변수에 사용되는 함수적 인터페이스



TIP

- 3
- XXXOperator는 입력 매개변수의 연산 결과는 동일한 타입으로 리턴하는 기능임
UnaryOperator : (입력 T → 출력 T)
BinaryOperator : (입력 (T,T) → 출력 T)

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    @FunctionalInterface
    public interface BinaryOperator<T> extends BiFunction<T,T,T>
```

자바API의 함수적 인터페이스

👉 표준형 Consumer<T> 함수적 인터페이스

5 `forEach(Consumer<? super T> action)`
Performs an action for each element of the

2

```
interface Consumer<T> {  
    public abstract void accept(T t);  
}
```

1

Consumer<T>



입력타입(T)



리턴없음(void)

EX

익명이너클래스

3

```
Consumer<String> c = new Consumer<String>() {  
    @Override  
    public void accept(String t) {  
        System.out.println(t);  
    }  
};  
  
c.accept("Consumer<T> 함수형 인터페이스");
```



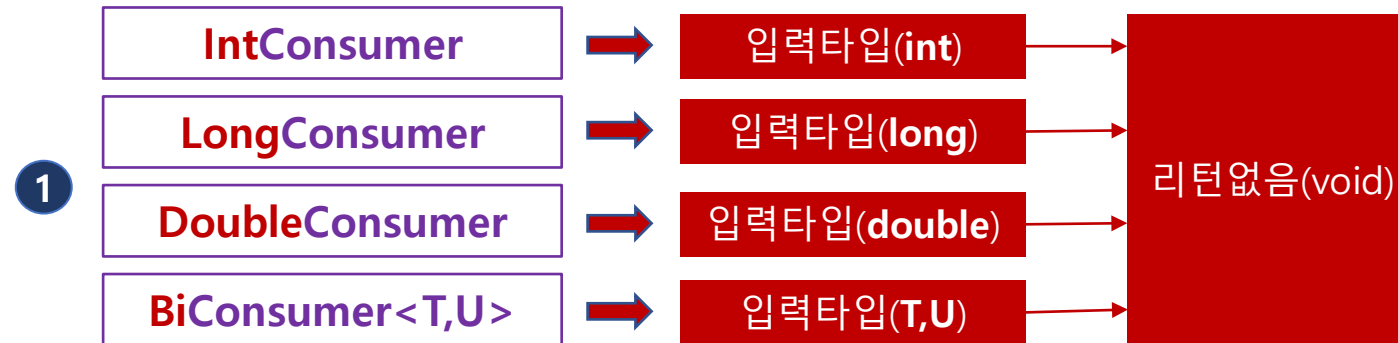
표준형 람다식

4

```
Consumer<String> c1;  
c1 = (str)→System.out.println(str);  
  
c1.accept("Consumer<T> 함수형 인터페이스");
```

자바API의 함수적 인터페이스

☞ 확장형 Consumer<T> 함수적 인터페이스



TIP

- 2
- 리턴타입이 기본자료형이 아니거나 확장형 인터페이스의 리턴타입이 모두 동일한 경우 인터페이스 메서드명은 동일함
(확장형 Consumer<T>는 모두 accept(..))

EX

확장형 람다식

3

```
IntConsumer c2 = (num)->System.out.println("int 값="+num);
LongConsumer c3 = (num)->System.out.println("long 값="+num);
DoubleConsumer c4 = (num)->System.out.println("double 값="+num);
BiConsumer<String, Integer> c5 = (name, age)->System.out.println("이름:"+name + " 나이:"+age);

c2.accept(3);
c3.accept(5L);
c4.accept(7.8);
c5.accept("홍길동", 16);
```

자바API의 함수적 인터페이스

표준형 Supplier<T> 함수적 인터페이스



```
interface Supplier<T> {  
    public abstract T get();  
}
```

3 **generate(Supplier<T> s)**
Returns an infinite sequence of elements provided by the provided Supplier.

2

EX

익명이너클래스

4

```
Supplier<String> s = new Supplier<String>() {  
    @Override  
    public String get() {  
        return "Supplier<T> 함수형 인터페이스";  
    }  
};  
  
System.out.println(s.get());  
//Supplier<T> 함수형 인터페이스
```

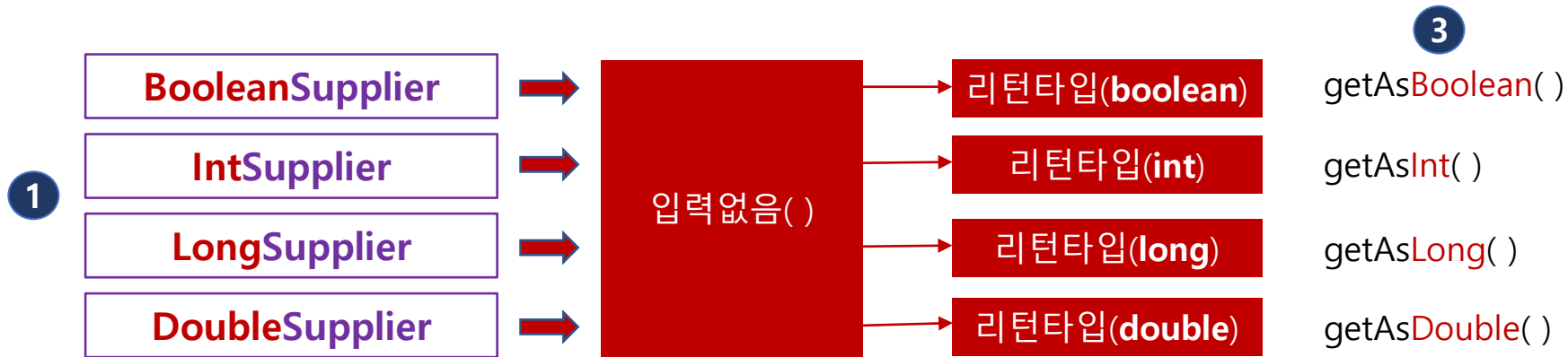
표준형 람다식

5

```
Supplier<String> s1;  
s1 = ()->"Supplier<T> 함수형 인터페이스";  
  
System.out.println(s1.get());  
//Supplier<T> 함수형 인터페이스
```

자바API의 함수적 인터페이스

☞ 확장형 Supplier <T> 함수적 인터페이스



TIP

- 2
- 리턴타입이 기본자료형이면서 확장형 인터페이스의 리턴타입이 다른 경우 인터페이스 메서드명은 서로 다름 (기본메서드명As리턴타입명)

EX

확장형 람다식

4

```
BooleanSupplier s2 = ()->false;
IntSupplier s3 = ()->2+3;
LongSupplier s4 = ()->10L;
DoubleSupplier s5 = ()->5.8;

System.out.println(s2.getAsBoolean()); //false
System.out.println(s3.getAsInt());      //5
System.out.println(s4.getAsLong());     //10
System.out.println(s5.getAsDouble());   //5.8
```

자바API의 함수적 인터페이스

👉 **표준형** Predicate<T> 함수적 인터페이스

1

Predicate<T>



입력타입(T)

리턴타입(boolean)

3

```
noneMatch(Predicate<? super T> predicate)  
Returns whether no elements of this stream match the given predicate.
```

2

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

EX

익명이너클래스

4

```
Predicate<String> p = new Predicate<String>() {  
    @Override  
    public boolean test(String t) {  
        return (t.length()>1)? true:false;  
    }  
};  
  
System.out.println(p.test("안녕")); //true
```



표준형 람다식

5

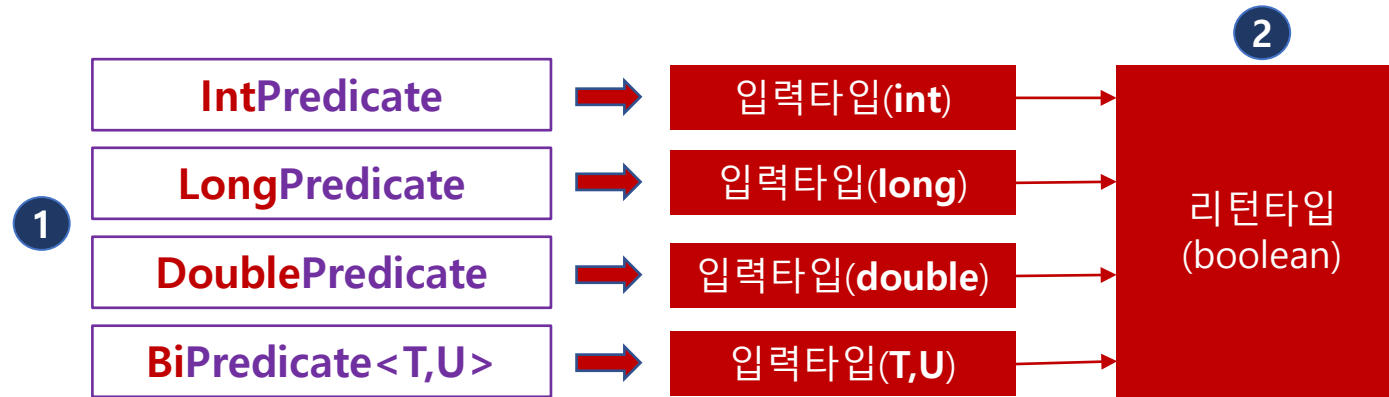
```
Predicate<String> p1;  
p1 = (str)->(str.length()>1)? true:false;  
  
System.out.println(p1.test("안녕")); //true
```

자바API의 함수적 인터페이스

☞ 확장형 Predicate<T> 함수적 인터페이스

TIP

- 리턴타입이 기본자료형이 아니거나 확장형 인터페이스의 리턴타입이 모두 동일한 경우 인터페이스 메서드명은 동일함 (확장형 Predicate<T>는 모두 **test(..)**)



EX

확장형 람다식

4

```
IntPredicate p2 = (num)->(num%2==0)? true:false;
LongPredicate p3 = (num)->(num>100)? true:false;
DoublePredicate p4 = (num)->(num>0)? true:false;
BiPredicate<String, String> p5 = (str1, str2)->str1.equals(str2);

System.out.println(p2.test(2)); //true
System.out.println(p3.test(85L)); //false
System.out.println(p4.test(-5.8)); //false
System.out.println(p5.test("안녕", "안녕")); //true
```


자바API의 함수적 인터페이스

👉 **표준형** `Function<T, R>` 함수적 인터페이스



3

`flatMap(Function<? super T,? ex`
Returns a stream consisting of the re
contents of a mapped stream produc
element.

2

```
interface Function<T,R> {  
    public abstract R apply (T t);  
}
```

EX

익명이너클래스

4

```
Function<String, Integer> f;  
f = new Function<String, Integer>() {  
    @Override  
    public Integer apply(String t) {  
        return t.length();  
    }  
};  
  
System.out.println(f.apply("안녕")); //2
```



표준형 람다식

5

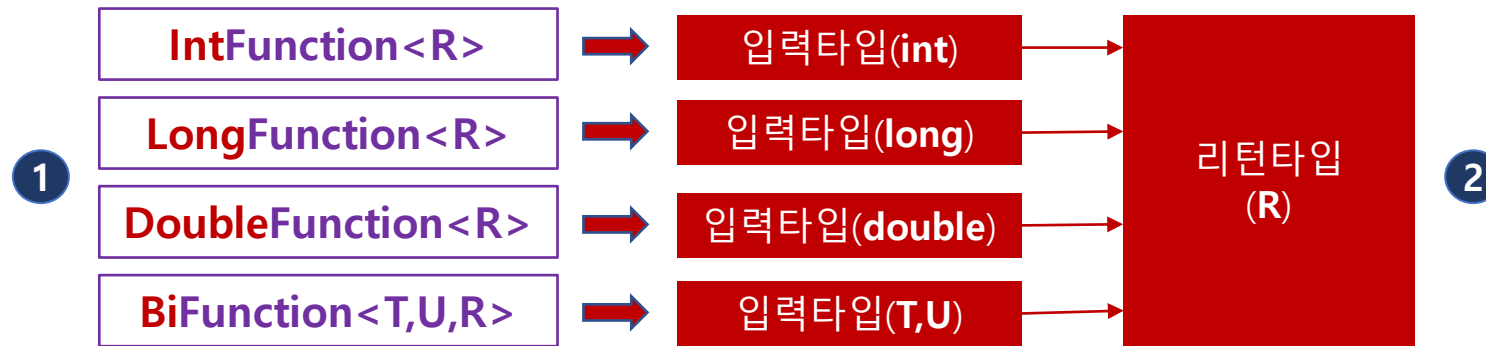
```
Function<String, Integer> f1;  
f1 = str->str.length();  
  
System.out.println(f1.apply("안녕")); //2
```

자바API의 함수적 인터페이스

☞ **확장형** `Function<T,R>` 함수적 인터페이스

TIP

- 리턴타입이 **기본자료형이 아니거나 확장형 인터페이스의 리턴타입이 모두 동일한 경우** 인터페이스 메서드명은 동일함 (이 경우는 모두 **apply(..)**)



EX

4

확장형 람다식

```
IntFunction<Double> f2 = (num)->(double)num; //int->double
LongFunction<String> f3 = (num)->String.valueOf(num); //long->문자열
DoubleFunction<Integer> f4 = (num)->(int)num; //double->int
BiFunction<String, Integer, String> f5 = (name, age)->"이름: "+name+", 나이 : "+age;

System.out.println(f2.apply(10)); //10.0
System.out.println(f3.apply(20L)); //20
System.out.println(f4.apply(30.5)); //30
System.out.println(f5.apply("홍길동", 16)); //이름: 홍길동, 나이 : 16
```

자바API의 함수적 인터페이스

☞ **확장형** `Function<T,R>` 함수적 인터페이스

TIP

- 리턴타입이 기본자료형이면서 확장형 인터페이스의 리턴타입이 다른 경우 인터페이스 메서드명은 서로 다름
(기본메서드명As리턴타입명)



EX

확장형 람다식

4

```
ToIntFunction<String> f6 = (str)->str.length(); //str->int
ToLongFunction<Double> f7 = (num)->num.longValue(); //double->long
ToDoubleFunction<Integer> f8 = (num)->num/100.0; //int->double

System.out.println(f6.applyAsInt("반갑습니다.")); //6
System.out.println(f7.applyAsLong(58.254)); //58
System.out.println(f8.applyAsDouble(250)); //2.5
```

자바API의 함수적 인터페이스

3 `iterate(T seed, UnaryOperator<T> f)`
Returns an infinite sequential ordered Stream of elements generated by repeatedly applying f to an initial element seed, producing a sequence of elements, etc.

👉 **표준형** `UnaryOperator<T>`, `BinaryOperator<T>` 함수적 인터페이스

2

```
interface UnaryOperator<T> {  
    public abstract T apply (T t);  
}
```

1 `UnaryOperator<T>` ➡ 입력타입(T) ➡ 리턴타입(T)

EX

익명이너클래스

4

```
UnaryOperator<Integer> uo;  
uo = new UnaryOperator<Integer>() {  
    @Override  
    public Integer apply(Integer t) {  
        return t*2;  
    }  
};  
System.out.println(uo.apply(5)); //10
```

표준형 람다식

5

```
UnaryOperator<Integer> uo1 = value->value*2;  
System.out.println(uo1.apply(5)); //10
```

자바API의 함수적 인터페이스

3 `reduce(BinaryOperator<T> accumulator)`
Performs a **reduction** on the elements of the function, and returns an Optional describing

👉 **표준형** UnaryOperator<T>, BinaryOperator<T> 함수적 인터페이스

1 **BinaryOperator<T>** ➡ **입력타입(T, T)** ➡ **리턴타입(T)**

2

```
interface BinaryOperator<T> {  
    public abstract T apply (T t1, T t2);  
}
```

EX

익명이너클래스

4

```
BinaryOperator<String> bo;  
bo = new BinaryOperator<String>() {  
    @Override  
    public String apply(String t, String u) {  
        return t+u;  
    }  
};  
System.out.println(bo.apply("안녕", "하세요"));  
//안녕하세요
```



표준형 람다식

5

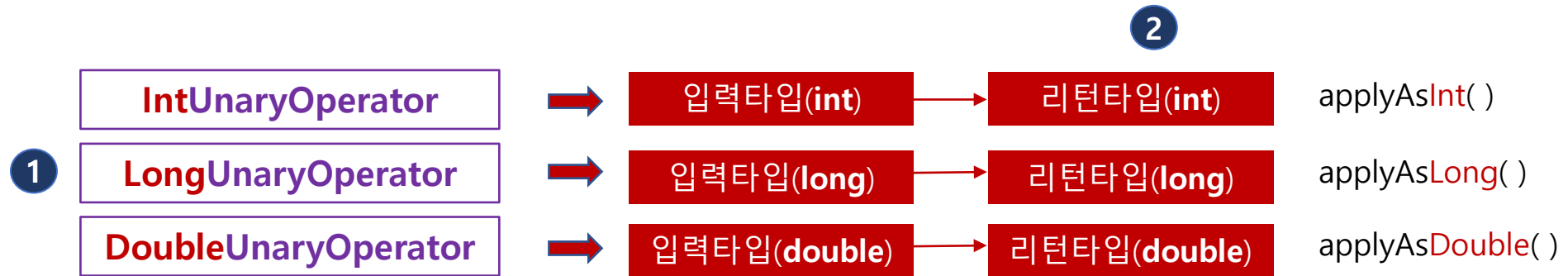
```
BinaryOperator<String> bo1 =  
    (value1, value2)->value1+value2;  
System.out.println(bo1.apply("안녕", "하세요"));  
//안녕하세요
```

자바API의 함수적 인터페이스

☞ 확장형 UnaryOperator<T> 함수적 인터페이스

TIP

- 리턴타입이 기본자료형이면서 확장형 인터페이스의 리턴타입이 다른 경우 인터페이스 메서드명은 서로 다름
(기본메서드명As리턴타입명)



EX

확장형 람다식

4

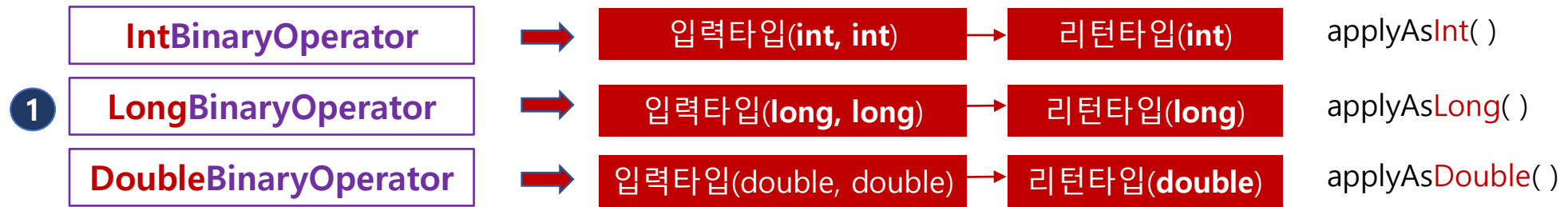
```
IntUnaryOperator uo2 = (num)->num*10; //int->int
LongUnaryOperator uo3 = (num)->num+20L; //long->long
DoubleUnaryOperator uo4 = (num)->num*10.0; //double->double
System.out.println(uo2.applyAsInt(10)); //100
System.out.println(uo3.applyAsLong(20L)); //40
System.out.println(uo4.applyAsDouble(30.5)); //305.0
```

자바API의 함수적 인터페이스

☞ **확장형** `BinaryOperator<T>` 함수적 인터페이스

TIP

- 리턴타입이 기본자료형이면서 확장형 인터페이스의 리턴타입이 다른 경우 인터페이스 메서드명은 서로 다름
(기본메서드명As리턴타입명)



EX

확장형 람다식

4

```
IntBinaryOperator bo2 = (num1,num2)->num1+num2; //int->int
LongBinaryOperator bo3 = (num1,num2)->num1*num2; //long->long
DoubleBinaryOperator bo4 = (num1,num2)->num1/num2; //double->double
System.out.println(bo2.applyAsInt(10,20)); //30
System.out.println(bo3.applyAsLong(20L, 10L)); //200
System.out.println(bo4.applyAsDouble(42.0, 12.0)); //3.5
```

The End