

Lambda Expression

Lambda Expression(람다식)

- 함수형 프로그래밍(Functional Programming)
- 함수를 변수처럼 사용할수 있는 개념
- Java 8 지원
- 익명 함수(Anonymous Function) / 익명 객체(Anonymous Object) 지칭
 - 이름이 없음, 파라미터는 () 괄호안에 지정
- 함수를 보다 단순하게 표현하는 법
- 함수형 인터페이스
 - 람다식을 선언하기 위한 인터페이스
 - 하나의 추상메서드만 선언된 인터페이스
 - @FunctionalInterface 어노테이션
 - 다중의 추상메서드 선언시 에러 발생
- 메서드와 함수 차이
 - 매개변수등을 받아서 수행하고, 결과를 반환하는 일련의 행위는 동일함
 - 메서드: 클래스에 종속(OOP 개념 용어)
 - 함수: 클래스에 독립(일반적 용어)

람다식 장/단점

■ 장점

- 코드 라인수 줄어듦
- 병렬 프로그래밍 가능
- 람다식으로 실행문을 전달 할 수 있음
- 가독성이 높음

■ 단점

- 재사용 불가: 일회용 함수
- 불필요하게 과용하게 되면 가독성이 떨어짐

람다식 구현하기

- 익명 함수 (Anonymous Function) 만들기
- 매개 변수와 매개 변수를 이용한 실행문 구현
- 람다식 형식:
 - (매개변수, 매개변수) -> { 함수 실행문; };
 - () -> 함수 실행문
- 예시:

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```



```
(a, b) -> a > b ? a : b
```

람다식 문법

메서드	함수
<pre>void doA(String str) { System.out.println(str); } void doA() { System.out.println("lambda"); }</pre>	<pre>str -> System.out.println(str) (String str) -> System.out.println(str.length()) () -> System.out.println("lambda")</pre> <p>매개변수 1개이면 () 생략 가능 단일 수행문이면 {} 생략 가능 매개변수에 타입 지정시에는 1개 이어도 (타입 매개변수명) 괄호 표기해야함 매개변수가 없는 경우에는 () 빈 괄호 표기해야함</p>
<pre>void doB(int a, int b) { System.out.println(a + b); }</pre>	<pre>(a, b) -> System.out.println(a + b);</pre> <p>매개변수 2개 이상이면 (a,b) 괄호 생략 불가 매개변수의 타입이 추론 가능하면 생략가능</p>
<pre>void doC(int a, int b) { return a + b; }</pre>	<pre>(a, b) -> { return a + b; }</pre> <p>단일 수행문이어도 return 구문을 명시하면 {} 생략 불가</p>
<pre>int doD(String str) { return str.length(); } int doE(int a, int b) { return a + b; }</pre>	<pre>str -> str.length(); (a, b) -> a + b;</pre> <p>수행문이 return 구문 하나라면 식 또는 값만 적고 return, {} 생략 가능</p>

람다식 사용 예시:

- 두 매개변수 중 더 큰 수를 반환하는 람다식
- 함수형 인터페이스 선언

```
6 @FunctionalInterface
7 interface LambdaFunctionMax {
8     int max(int a, int b);
9 }
```

- 람다식 구현 및 호출 사용

```
19 // --1. 함수형인터페이스
20 LambdaFunctionMax f = (a, b) -> a > b ? a : b;
21
22 int answer = f.max(5, 10);
23 System.out.println(answer);
```

실습: 람다식 작성하기

메서드	함수 (TODO)
<pre>int min(int a, in b) { return a < b ? a : b; }</pre>	
<pre>int printScore(String name, int score) { System.out.println(name + ":" + score); }</pre>	
<pre>int square(int x) { return x * x; }</pre>	
<pre>int lottoNo() { return Math.random(45)+1; }</pre>	

실습: 람다식 작성하기

메서드	함수
<pre>int min(int a, in b) { return a < b ? a : b; }</pre>	<pre>(a, b) -> a < b ? a : b</pre>
<pre>int printScore(String name, int score) { System.out.println(name + ":" + score); }</pre>	<pre>(name, score) -> System.out.println(name + ":" + score)</pre>
<pre>int square(int x) { return x * x; }</pre>	<pre>x -> x * x</pre>
<pre>int lottoNo() { return Math.random(45)+1; }</pre>	<pre>() -> Math.random(45)+1</pre>

익명 객체를 생성하는 람다식

- 객체지향 언어이므로 객체를 생성해야 메서드가 호출됨
- 람다식 메서드 구현하고 호출시 내부에서 익명 클래스가 생성됨
- 익명객체

//--2. 익명객체

```
List<String> list = Arrays.asList("김혜진", "서은지", "한진성", "강석민", "남현우");  
System.out.println(list);
```

```
Collections.sort(list, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.compareTo(s2);  
    }  
});
```

```
System.out.println(list);
```

[김혜진, 서은지, 한진성, 강석민, 남현우]
[강석민, 김혜진, 남현우, 서은지, 한진성]

람다식

//--3. 람다식

```
list = Arrays.asList("김혜진", "서은지", "한진성", "강석민", "남현우");  
System.out.println(list);
```

```
Collections.sort(list, (s1, s2) -> s1.compareTo(s2));  
System.out.println(list);
```

함수를 변수처럼 사용하는 람다식

■ 인터페이스형 변수에 람다식 대입

```
@FunctionalInterface
interface LambdaFunction2Print {
    void printString(String str);
}
```

//--4-2. 인터페이스형 변수에 람다식 대입:

```
LambdaFunction2Print f2 = str -> System.out.println(str);  
f2.printString("f2: A1");
```

함수를 변수처럼 사용하는 람다식

■ 매개변수로 전달하는 람다식

```
@FunctionalInterface
interface LambdaFunction2Print {
    void printString(String str);
}
```

//--4-3. 메서드의 매개변수로 함수형 인터페이스 타입의 매개변수 전달

```
LambdaFunction2Print f3 = str -> System.out.println(str + "***");
methodA(f3, "f3: B1");
```

//--4-1. 함수형 인터페이스 타입으로 매개변수 받음

```
public static void methodA(LambdaFunction2Print f, String str) {
    f.printString(str); // 람다식 호출
}
```

함수를 변수처럼 사용하는 람다식

■ 반환 값으로 사용하는 람다식

```
@FunctionalInterface
interface LambdaFunction2Print {
    void printString(String str);
}
```

//--5-1. 함수형 인터페이스 타입의 반환타입



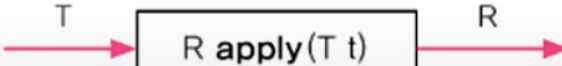
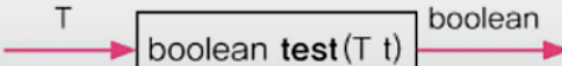
```
public static LambdaFunction2Print methodB() {
    //LambdaFunction2Print f2 = str -> System.out.println(str);
    //return f2;
    return str -> System.out.println(str + "===");
}
```

//--5-2. 반환값으로 사용하는 람다식

```
LambdaFunction2Print f4 = methodB();
f4.printString("f4: C1");
```

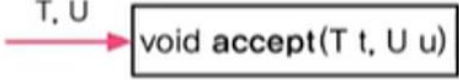
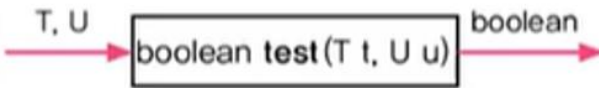
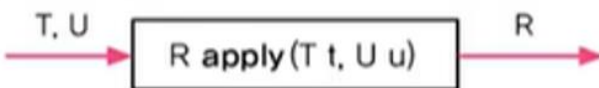
java.util.function

■ 자주 사용되는 함수형 인터페이스 제공

함수형 인터페이스	메서드	참고
java.lang. Runnable	<div>void run()</div>	매개변수도 없고, 반환값도 없음.
Supplier<T>	<div>T get()</div> 	매개변수는 없고, 반환값만 있음.
Consumer<T>	 <div>void accept(T t)</div>	Supplier와 반대로 매개변수만 있고, 반환값이 없음
Function<T,R>	 <div>R apply(T t)</div>	일반적인 함수. 하나의 매개변수를 받아서 결과를 반환
Predicate<T>	 <div>boolean test(T t)</div>	조건식을 표현하는데 사용됨. 매개변수는 하나, 반환 타입은 boolean

```
Predicate<String> isEmptyStr = s -> s.length()==0;  
String s = "";  
  
if(isEmptyStr.test(s)) // if(s.length()==0)  
    System.out.println("This is an empty String.");
```

■ 매개변수가 2개인 함수형 인터페이스

함수형 인터페이스	메서드	참고
BiConsumer<T,U>	 void accept (T t, U u)	두개의 매개변수만 있고, 반환값이 없음
BiPredicate<T,U>	 boolean test (T t, U u)	조건식을 표현하는데 사용됨. 매개변수는 둘, 반환값은 boolean
BiFunction<T,U,R>	 R apply (T t, U u)	두 개의 매개변수를 받아서 하나의 결과를 반환