

14장. 랴다식 & 스트림



Lambda & Stream



람다식(lambda expression)

함수형 프로그래밍과 람다식

- 자바는 객체를 기반으로 프로그램을 구현하며, 어떤 기능이 필요하다면 클래스를 만들고 그 안에 기능을 구현한 메서드(함수)를 만든 후 사용한다.
- 그러므로, 클래스가 없다면 메서드를 사용할 수 없다.
- 자바 8이후부터 사용할 수 있는 람다식은 객체 없이 인터페이스의 구현만으로 메서드를 호출할 수 있다.

람다식 구현하기

- 함수 이름이 없는 **익명 함수**를 만드는 것으로 익명 객체 구현.
- 표현식 : **(매개변수) -> { 실행문; }**

```
int add(int x, int y){  
    return x + y;  
}
```

일반 메서드(함수)



```
(x, y) -> x + y
```

람다식



람다식(lambda expression)

람다식 문법 살펴보기

- 매개변수 자료형과 괄호 생략하기 : 매개변수가 하나인 경우 괄호 생략

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

- 중괄호 생략하기 : 중괄호 안의 구현 부분이 한 문장인 경우 중괄호 생략

```
str -> System.out.println(str);
```

- 매개변수가 없다면 괄호 생략할 수 없음

```
() -> { 실행문; }
```

- return 생략

```
(x, y) -> {return x + y};
```

```
(x, y) -> x + y;
```



람다식(lambda expression)

함수형 인터페이스

람다식을 구현하기 위해 **함수형 인터페이스**를 만들고, 인터페이스에 람다식으로 구현할 메서드를 선언한다. 람다식은 이름이 없는 익명 함수로 구현하기 때문에 메서드에 **오직 하나의 추상 메서드**만 선언할 수 있다. (여러 개는 구현이 모호해지므로)

- 객체지향언어는 객체를 기반으로 구현하는 방식
- 함수형 프로그램은 함수를 기반으로 하고 자료를 입력받아 구현하는 방식이다.



람다식(lambda expression)

매개 변수가 없는 람다식

```
package lambda.functionalinterface;

@FunctionalInterface
public interface MyFunctionalInterface {

    public void method();

}
```

애너테이션을 명시해서 실행
전에 오류 체크함

```
MyFunctionalInterface fi;
```

```
fi = () -> { //매개 변수가 없는 경우 ()-빈 괄호
    String str = "Hello~ lambda";
    System.out.println(str);
};
fi.method();
```

```
//{ } 생략 가능
fi = () -> System.out.println("Hello~ lambda");
```



람다식(lambda expression)

매개 변수가 1개 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface2 {
    //매개변수가 없고 반환값이 없는 메서드
    public void method(int x);
}
```

```
MyFuncInterface2 fi;
```

```
//덧하기 계산
```

```
fi = (x) -> {
    x = x + 10;
    System.out.println(x);
};
fi.method(10);
```

```
fi = (x) -> System.out.println(x + 10);
fi.method(10);
```

```
//제곱수 계산
```

```
fi = (n) -> System.out.println(n * n);
fi.method(4);
```



람다식(lambda expression)

매개변수가 2개 있고 return문이 있는 함수형 인터페이스

```
@FunctionalInterface
public interface MyNumber {
    int getMax(int n1, int n2);
    //int add(int a, int b);
}
```

1개의 추상메서드만 사용가능

```
MyNumber num;

num = (x, y) -> {
    return (x >= y) ? x : y;
};
System.out.println(num.getMax(10, 20));

//{}와 return 생략 가능
num = (x, y) -> (x >= y) ? x : y;
System.out.println("더 큰 수는 " + num.getMax(10, 20));
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The target type of this expression must be a functional interface

at lambda.functionalinterface.TestMyNumber.main([TestMyNumber.java:7](#))



실습 문제

정수형 계산기 구현

```
@FunctionalInterface
public interface Calculator {
    int calculate(int n1, int n2);
}
```

```
Calculator add, sub, mul, div;
add = (x, y) -> x + y;
System.out.println(add.calculate(10, 5));

sub = (x, y) -> x - y;
System.out.println(sub.calculate(10, 5));

mul = (x, y) -> x * y;
System.out.println(mul.calculate(10, 5));

div = (x, y) -> x / y;
System.out.println(div.calculate(10, 5));
```



람다식(lambda expression)

객체 지향 프로그래밍 방식과 람다식 비교

1. 객체 지향 프로그래밍 방식

```
package lambda.stringconcat;

@FunctionalInterface
public interface StringConcat {

    public void makeString(String s1, String s2);
}

public class StringConcatImpl implements StringConcat{

    @Override
    public void makeString(String s1, String s2) {
        System.out.println(s1 + ", " + s2);
    }
}

String str1 = "Hill";
String str2 = "State";

//일반 객체 지향 프로그래밍
StringConcatImpl concat1 = new StringConcatImpl();
concat1.makeString(str1, str2);
```



람다식(lambda expression)

- 객체 지향 프로그래밍 방식과 람다식 비교

- 2. 람다 프로그래밍 방식

```
package lambda.stringconcat;
```

```
@FunctionalInterface
```

```
public interface StringConcat {
```

```
    public void makeString(String s1, String s2);
```

```
}
```

```
String s1 = "Hill";
```

```
String s2 = "State";
```

```
//람다 프로그래밍
```

```
StringConcat concat2;
```

```
concat2 = (s, v) -> System.out.println(s + ", " + v);
```

```
concat2.makeString(s1, s2);
```



람다식(lambda expression)

매개변수가 없는 람다식

```
@FunctionalInterface
public interface Workable {
    void work();
}
```

```
public class Person {
    public void action(Workable workable) {
        workable.work();
    }
}
```



람다식(lambda expression)

매개변수가 없는 람다식

```
public class LambdaPersonTest {  
  
    public static void main(String[] args) {  
        Person person = new Person();  
  
        person.action(() -> {  
            System.out.println("출근을 합니다");  
            System.out.println("프로그래밍을 합니다.");  
        });  
  
        person.action(() -> System.out.println("퇴근을 합니다.));  
    }  
}
```



람다식(lambda expression)

함수형 인터페이스를 변수 및 반환 자료형으로 사용

```
@FunctionalInterface
interface PrintString{

    void showString(String str);
}

public class LambdaTest {

    public static void main(String[] args) {
        //인터페이스형 객체 선언 - 람다식 구현
        PrintString lambdaPrint;
        lambdaPrint = (str) -> System.out.println(str);
        lambdaPrint.showString("Good Luck!!");

        //void형 메서드 호출
        writeString(lambdaPrint);
    }
}
```



람다식(lambda expression)

```
//return이 있는 메서드 호출
PrintString prStr = returnPrint();
prStr.showString("Good Luck!!");
}

//함수형 인터페이스를 매개변수로 전달
public static void writeString(PrintString prStr) {
    prStr.showString("Good Luck!!");
}

//함수형 인터페이스를 반환 자료형으로 사용
public static PrintString returnPrint() {
    return str -> System.out.println(str);
}
```


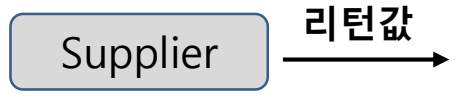
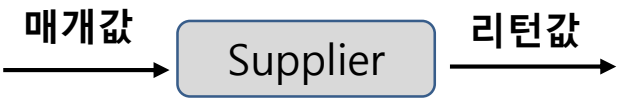
```
Good Luck!!
Good Luck!!
Good Luck!!
```



표준 API의 함수적 인터페이스

자바8부터는 빈번하게 사용되는 함수적 인터페이스(functional interface)는 `java.util.function` 표준 API로 패키지로 제공한다.

이 패키지에서 제공하는 함수적 인터페이스의 목적은 메소드 또는 생성자의 매개 타입으로 사용되어 람다식을 대입할 수 있도록 하기 위해서이다.

종 류	추상 메서드의 특징	설명
Consumer	- 매개값은 있고, 리턴값은 없음	
Supplier	- 매개값은 없고, 리턴값은 있음	
Function	- 매개값도 있고, 리턴값도 있음	



표준 API의 함수적 인터페이스

Consumer 함수적 인터페이스의 특징은 리턴값이 없는 **accept()** 메서드를 가지고 있다. accept()는 단지 매개값을 소비하는 역할만 한다. 소비한다는 리턴값이 없다는 뜻이다.

Consumer<T>

```
Consumer<String> consumer = t -> {t를 소비하는 실행문}
```

BiConsumer<T>

```
BiConsumer<String, String> consumer = (t, u) -> {t와 u를 소비하는 실행문}
```

DoubleConsumer<T>

```
DoubleConsumer consumer = d -> {d를 소비하는 실행문}
```

```
Module java.base
Package java.util.function
Interface Consumer<T>

Type Parameters:
T - the type of the input to the operation

All Known Subinterfaces:
Stream.Builder<T>

Functional Interface:
This is a functional interface and can therefore be

@FunctionalInterface
public interface Consumer<T>
```



표준 API의 함수적 인터페이스

Consumer<T> 사용 예제

```
package lambda.consumer;

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

public class ConsumerExample {
```

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}
```

```
    public static void main(String[] args) {
        //t를 소비하는 실행문
        Consumer<String> consumer = t -> System.out.println(t + "8");
        consumer.accept("Java");

        //t, u를 소비하는 실행문 -> 2개
        BiConsumer<String, String> biConsumer = (t, u) -> System.out.println(t + u);
        biConsumer.accept("Java", "8");

        //d를 소비하는 실행문(실수형)
        DoubleConsumer doubleConsumer = d -> System.out.println("Java" +
        doubleConsumer.accept(8.0);
    }
```

매개값

```
Java8
Java8
Java8.0
```



표준 API의 함수적 인터페이스

IntSupplier 인터페이스를 타겟 타입으로 하는 람다식은 **getAsInt()** 메서드가 매개값을 가지지 않으므로 람다식도 ()를 사용한다.

```
package lambda.supplier;

import java.util.function.IntSupplier;

public class SupplierExample {

    public static void main(String[] args) {
        //IntSupplier매개값이 없고 리턴값이 있음
        IntSupplier intSupplier = () -> {
            int num = (int) (Math.random()*6) + 1;
            return num;
        };
        int num = intSupplier.getAsInt();
        System.out.println("주사위 눈 : " + num);
    }
}
```



스트림(Stream)

■ 스트림(Stream)

- 자료가 모여있는 배열이나 컬렉션에서 처리에 대한 기능을 구현해 놓은 클래스
- 스트림 클래스는 람다식으로 처리하는 반복자이다.

```
public class StreamTest {  
  
    public static void main(String[] args) {  
        /*ArrayList<String> companyList = new ArrayList<>();  
        companyList.add("LG");  
        companyList.add("Samsung");  
        companyList.add("Hyundai");*/  
  
        //Arrays 클래스 사용  
        List<String> companyList = Arrays.asList("LG", "Samsung", "현대");  
  
        for(String company : companyList)  
            System.out.println(company);  
  
        //Stream 클래스 - 람다식으로 구현  
        Stream<String> stream = companyList.stream();  
        stream.forEach(company -> System.out.println(company));  
    }  
}
```



스트림(Stream)

▪ Stream 클래스

Module `java.base`

Package `java.util.stream`

Classes to support functional-style operations on streams of elements,

```
int sum = widgets.stream()
    .filter(b -> b.getColor() == RED)
    .mapToInt(b -> b.getWeight())
    .sum();
```

▪ Arrays 클래스 – `asList()`, `stream()` 메서드

asList

```
@SafeVarargs public static <T> List<T> asList(T... a)
```

Returns a fixed-size list backed by the specified array. (Changes to the returned combination with `Collection.toArray()`. The returned list is serializable and implements `RandomAccess`.

This method also provides a convenient way to create a fixed-size list initialized with elements from an array:

```
List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");
```

stream

```
public static IntStream stream(int[] array)
```

Returns a sequential `IntStream` with the specified array.

Parameters:

array - the array, assumed to be unmodified during use



스트림(Stream)

- Arrays 클래스

```
public class ArraysTest {  
  
    public static void main(String[] args) {  
        int[] num1 = {3, 1, 2, 4};  
  
        System.out.println(Arrays.toString(num1));  
  
        //num1의 요소 중 2개 복사  
        int[] num2 = Arrays.copyOf(num1, 2);  
        System.out.println(Arrays.toString(num2));  
  
        //num1 오름차순 정렬하기  
        Arrays.sort(num1);  
        System.out.println(Arrays.toString(num1));  
    }  
}
```



스트림(Stream)

- 배열로부터
스트림 얻기

```
//문자형 배열
String[] fruit = {"apple", "banana", "grape"};

for(int i = 0; i < fruit.length; i++) {
    System.out.println(fruit[i]);
}

//stream의 forEach()와 람다식으로 구현
Stream<String> strStream = Arrays.stream(fruit);
strStream.forEach(str -> System.out.println(str));

//정수형 배열
int[] num = {1, 2, 3, 4};

for(int n : num)
    System.out.println(n);

//스트림을 얻어서 출력
Arrays.stream(num).forEach(n -> System.out.println(n));

//스트림을 얻어서 합계 구하기
int sumVal = Arrays.stream(num).sum();
int count = (int) Arrays.stream(num).count(); //count() long형 반환
double avg = (double)sumVal / count;

System.out.println("합계 : " + sumVal);
System.out.println("개수 : " + count);
System.out.println("평균 : " + avg);
```



스트림(Stream)

■ 스트림(Stream) 연산

중간 연산 – 자료를 거르거나 변경하여 또 다른 자료를 내부적으로 생성함

최종 연산 – 생성된 내부 자료를 소모해 가면서 연산을 수행함

중간연산 – filter(), map()

- 문자열 배열이 있을 때 문자열의 길이가 5 이상인 경우 출력
`sList.stream().filter(s->s.length() >=5).forEach(s->System.out.println(s));`
- 고객 클래스에서 고객 이름만 가져와 출력
`customerList.stream().map(c->c.getName()).forEach(s->System.out.println(s));`

최종 연산 – forEach(), count(), sum(), reduce()

- List컬렉션에서 정렬후 출력
`sList.stream().sorted().forEach(str -> System.out.print(str+ " "));`



스트림(Stream)

■ 컬렉션으로 스트림 얻기

```
package stream.kind;

public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public String getName() {
        return name;
    }

    public int getScore() {
        return score;
    }
}
```



스트림(Stream)

```
public class StudentStreamTest {  
  
    public static void main(String[] args) {  
        List<Student> list = Arrays.asList(  
            new Student("콩쥐", 90),  
            new Student("팥쥐", 70),  
            new Student("심청", 80)  
        );  
  
        //Student로 부터 스트림 얻어 출력하기  
        Stream<Student> stdStream = list.stream();  
        stdStream.forEach(std -> {  
            //System.out.println(std.getName() + " : " + std.getScore());  
            String name = std.getName();  
            int score = std.getScore();  
            System.out.println(name + " : " + score);  
        });  
    }  
}
```



스트림(Stream)

```
//stream()은 한 번 사용하면 소모되므로 다시 값을 저장함
//map() 함수 - 이름, 점수를 각각 출력
stdStream = list.stream();
stdStream.map(std -> std.getName())
          .forEach(s -> System.out.println(s));

stdStream = list.stream();
stdStream.mapToInt(std -> std.getScore())
          .forEach(s -> System.out.println(s));

//filter() 함수 - 점수가 90 이상인 학생의 이름 출력
list.stream().filter(std -> std.getScore() >= 90)
          .map(std -> std.getName())
          .forEach(s -> System.out.println(s));
}
```



스트림(Stream)

■ 스트림을 활용하여 여행객의 여행 비용 계산하기

CustomerLee	CustomerKang	CustomerHong
- 이름 : 이순신 - 나이 : 40 - 비용 : 100만원	- 이름 : 강감찬 - 나이 : 30 - 비용 : 100만원	- 이름 : 홍길동 - 나이 : 14 - 비용 : 50만원

▶ 예제 시나리오

1. 고객의 명단을 출력합니다.
2. 여행의 총 비용을 계산합니다.
3. 고객 중 20세 이상인 사람의 이름을 정렬하여 출력합니다.



스트림(Stream)

■ 스트림을 활용하여 여행객의 여행 비용 계산하기

```
public class Customer {  
    private String name;  
    private int age;  
    private int price;  
  
    public Customer(String name, int age, int price) {  
        this.name = name;  
        this.age = age;  
        this.price = price;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
}
```



스트림(Stream)

■ 스트림을 활용하여 여행객의 여행 비용 계산하기

```
List<Customer> customerList = new ArrayList<>();

Customer lee = new Customer("이순신", 40, 100);
Customer kang = new Customer("강감찬", 10, 100);
Customer hong = new Customer("홍길동", 15, 50);

customerList.add(lee);
customerList.add(kang);
customerList.add(hong);

System.out.println("===고객명단 추가된 순서대로 출력 ===");
customerList.stream().map(c -> c.getName()).forEach(s -> System.out.println(s));

int total = customerList.stream().mapToInt(c->c.getPrice()).sum();
System.out.println("총 여행 비용은 : " + total + "입니다.");

System.out.println("===20세 이상 고객 명단 정렬하여 출력===");
customerList.stream().filter(c->c.getAge() >= 20).map(c->c.getName())
    .sorted().forEach(s->System.out.println(s));
```

===고객명단 추가된 순서대로 출력 ===

이순신

강감찬

홍길동

총 여행 비용은 : 250입니다.

===20세 이상 고객 명단 정렬하여 출력===

이순신



스트림(Stream)

■ 실습 예제

스트림을 활용하여 다음처럼 책 목록을 출력해 보세요.

- 모든 책의 가격의 합
- 책의 가격이 20,000원 이상인 책의 이름을 정렬하여 출력

```
class Book{
    private String name;
    private int price;

    public Book(String name, int price)
        this.name = name;
        this.price = price;
    }
}
```

```
public class LibraryTest {
    public static void main(String[] args) {
        List<Book> bookList = new ArrayList<>();

        bookList.add(new Book("자바", 25000));
        bookList.add(new Book("파이썬", 19000));
        bookList.add(new Book("안드로이드", 30000));

        //스트림 생성하고 출력하기
    }
}
```

