프로그래밍 언어 응용

chapter13

내부클래스, 람다식, 스트림

제공된 자료는 훈련생의 수업을 돕기 위한 것으로, 타인과 공유하시면 안됩니다.

Contents

내부클래스

내부클래스란?

내부 클래스란?

- 내부클래스는 클래스 안에 선언된 클래스이다.
- 클래스간의 포함 관계를 표현할 수 있다.
- 다른 클래스에서 사용할 일이 없는 경우에 주로 사용한다.

날씨API의 응답데이터

내부클래스의 특징

내부 클래스의 장점

- 내부 클래스는 외부 클래스의 멤버를 자유롭게 사용할 수 있다.
- 외부에 불필요한 클래스를 감출 수 있다.

```
class A {
   int a;
}

class B {
   new A().a = 10;
}
```

```
class A {
   int a;

   class B {
      a = 10;
   }
}
```

내부클래스 종류

내부 클래스는 선언 위치에 따라 구분되며, 종류에 따라 유효 범위가 다르다.

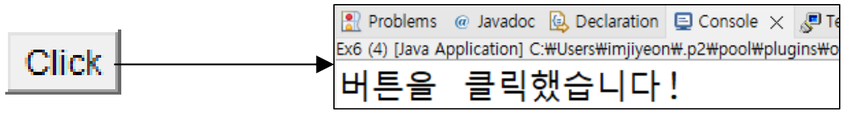
- 인스턴스 내부 클래스: 외부 클래스의 인스턴스가 생성된 후에, 사용할 수 있다.
- 정적 내부 클래스: 외부 클래스의 인스턴스를 생성하지 않아도, 사용할 수 있다.
- 지역 내부 클래스: 메소드 안에서만 사용 할 수 있다.

```
class Outer {

    class InstanceInner {          //인스턴스 클래스
    }
    static class StaticInner {     //정적 클래스
    }

    void method(){
        class LocalInner {}        //지역 클래스
    }
}
```

익명 클래스란?

익명 클래스란?

- 이름이 없는 클래스를 의미한다.
- 익명 클래스는 한번만 쓸 때 사용한다.



클래스 정의 + 객체 생성

```
class EventHandler implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("버튼을 클릭했습니다!");
    }
}

button.addActionListener(new EventHandler());
```

액션리스너의 구현클래스를 만들고, 객체를 생성한 후에 사용

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("버튼을 클릭했습니다!");
    }
});
```

익명 클래스를 사용

익명 클래스 만들기

익명 클래스의 특징

- 인터페이스와 추상클래스는 직접 사용할 수 없다.
- 그런데 익명 클래스를 사용하면, 클래스를 정의와 객체 생성을 한번에 할 수 있다.

익명 클래스 만드는 방법

1. 부모 인터페이스 타입으로 객체를 생성한다.
2. 바로 구현클래스를 만들면서, 부모한테 물려받은 추상 메소드를 구현한다.

익명 클래스

```
button.addActionListener( new ActionListener (){
    @Override
    public void actionPerformed(ActionEvent e){
        System.out.println("버튼을 클릭했습니다!");
    }
});
```

내부클래스 정리

내부 클래스	특징
인스턴스 내부 클래스	외부 클래스의 멤버로 사용된다. 외부 클래스의 객체를 생성한 후에 내부 클래스의 객체를 생성할 수 있다.
정적 내부 클래스	외부 클래스의 멤버로 사용된다. 외부 클래스의 객체 없이 사용할 수 있다.
지역 내부 클래스	메소드 안에 선언되어, 메소드 내부에서만 사용할 수 있다.
익명 내부 클래스	클래스 정의와 객체 생성을 동시에 하는 이름이 없는 일회용 클래스이다.

람다식

함수형 프로그래밍란?

자바는 객체지향 언어로, 기능이 필요하면 클래스 안에 메소드를 정의해야 한다.
그런데 자바 8부터 함수형 프로그래밍 방식을 지원하고 있다.
함수형 프로그래밍은 클래스 없이 함수만 정의하는 방식이다.
함수형 프로그래밍을 사용하면 함수를 더 쉽게 만들 수 있다.


```
class CalcImpl implements Calc {

    @Override
    public int hap(int x, int y) {
        return x + y;
    }

}

Calc calc = new CalcImpl();
calc.hap(3, 5);
```

객체지향 프로그래밍 방식

```
Calc calc = (x, y) -> x + y;
calc.hap(3, 5);
```

함수형 프로그래밍 방식

함수형 인터페이스와 람다식

함수형 프로그래밍는 함수형 인터페이스와 람다 표현식을 사용한다.

- **함수형 인터페이스**: 단 하나의 추상 메소드를 가지는 인터페이스이다.
- **람다식**: 함수를 간단하게 표현하는 방식이다.

함수형 인터페이스

```
@FunctionalInterface
interface MyNumber {
    int add(int num1, int num2);
}
```

람다식으로 인터페이스 구현하기

```
MyNumber number = (x, y) -> x+y ;

number.add(10, 20);
```

람다 표현식의 특징

- 함수를 람다식으로 표현하면 이름이 없기 때문에, 익명 함수가 된다.
- 함수의 이름, 반환타입, 매개변수타입를 생략할 수 있다.
- 코드가 한 줄인 경우에는 {} 중괄호와 return키워드를 생략할 수 있다.

```
int method (int x) {
    print(x);
}
```

➡

```
x -> { print(x); }
```

```
void method () {
    print("안녕하세요")
}
```

➡

```
( ) -> print("안녕하세요");
```

```
int method (int x, int y) {
    print(x+y);
}
```

➡

```
( x,  y ) -> { print(x); }
```

```
int method () {
    return x + y;
}
```

➡

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

스트림이란?

스트림이란?

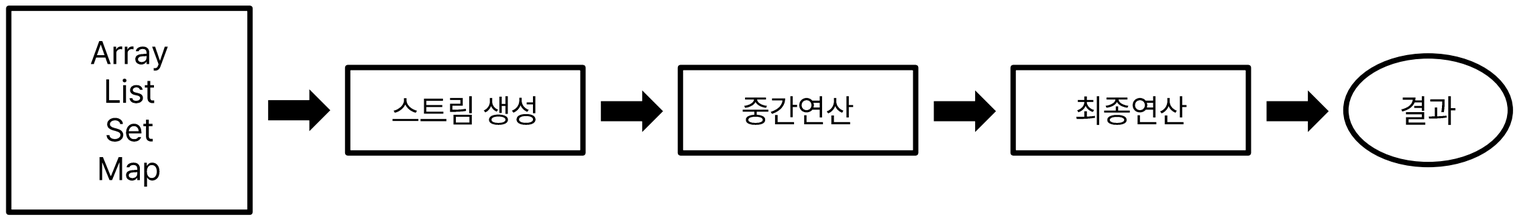
- 다양한 자료구조에 담긴 데이터를 일관된 방법으로 처리하는 데 사용된다.
- 다양한 자료구조를 기반으로 스트림을 생성한다.
- 스트림은 중간연산과 최종연산을 사용하여 데이터를 조작하고 최종결과를 얻는다.

예시

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
list.stream().forEach(n -> System.out.print(n));
```

```
String[] strArr = { "a", "b", "c" };
Arrays.stream(strArr).forEach(n -> System.out.print(n));
```

스트림 생성 과정



스트림 만들기

리스트로부터 스트림 생성하기

- Stream<T> list.stream ()

→ List<Integer> list = new ArrayList( );
list.stream ( );

객체배열로부터 스트림 생성하기

- Stream<T> Stream.of (T[] array)

→ Stream.of ({1, 2, 3});

기초배열로부터 스트림 생성하기

- IntStream Arrays.stream (int [ ])

→ int[ ] arr = { 1, 2, 3 };
Arrays.stream (arr);

스트림의 특징

- 한번 생성하고 사용한 스트림은 재사용할 수 없다.
- 스트림의 연산은 원본 데이터를 변경하지 않는다.
- 최종연산은 마지막에 한번만 적용된다.

```
int sum = stream.mapToInt(n -> n.intValue()).sum();
int count = stream.count(); // 에러남. 스트림이 이미 닫혔음
```

```
List<Integer> list = Arrays.asList(5, 1, 2, 4, 3);
List<Integer> sortedList = list.stream().sorted().collect(Collecto:

System.out.println(list); // [5, 1, 2, 4, 3]
System.out.println(sortedList); // [1, 2, 3, 4, 5]
```

```
list.stream().distinct().sorted().forEach(n->System.out.print(n)); //12345
```

스트림의 연산

중간연산

- 필터링, 변환, 정렬, 그룹화 등 다양한 작업을 수행할 수 있다.
- 중간연산이 끝나면 새로운 스트림을 반환하기 때문에, 반복적으로 사용할 수 있다.

최종연산

- 최종연산은 스트림의 각 요소를 꺼내어 반복작업을 수행하고, 결과값을 반환한다.
- 최종연산이 끝나면 스트림이 닫히고, 더 이상 스트림을 사용할 수 없다.

```
list.stream().filter(n -> n >= 3).sorted().forEach(n -> System.out.print(n));
```

스트림 생성 | 중간연산 | 중간연산 | 최종연산

연산의 종류

중간 연산	내용
Stream<T> distinct( )	중복되는 요소를 제거한다
Stream<T> filter( )	조건에 안 맞는 요소를 걸러낸다
Stream<T> limit( )	스트림의 일부를 잘라낸다
Stream<T> skip( )	스트림의 일부를 건너뛴다
Stream<T> peek( )	특정 작업을 수행한다
Stream<T> sorted( )	스트림의 요소를 정렬한다
Stream<T> map( )	스트림의 요소를 변환한다
DoubleStream mapToDouble( ) IntStream mapToInt( ) LongStream mapToLong( )	스트림을 기본형 스트림을 변환한다

연산의 종류

최종 연산	내용
void forEach( )	각 요소에 지정된 작업을 수행한다
long count( )	스트림 요소 개수를 반환한다
Optional max( )	스트림 요소 중 최대값을 반환한다
Optional min( )	스트림 요소 중 최소값을 반환한다
int sum( )	스트림 요소를 모두 더한 총 합을 반환한다
T reduce( )	기본값을 사용하여 반복작업을 수행한다
R collect( )	스트림을 다른 자료형으로 변환한다