

Effective Java

Item 42. 익명 클래스보다는 람다를 사용하라

목차

- 익명 클래스의 사용
- 람다식의 등장과 사용
- 람다식의 특징
- 요약 정리

익명 클래스의 사용

익명 클래스의 사용

함수형 언어 ✕

객체 지향 언어 ○

- 함수를 변수에 담을 수 없다.
- 함수를 다른 함수의 인자로 넘길 수 없다.
- 반환 값으로 줄 수 없다.



Java™

익명 클래스의 사용

문자열 정렬 기준을 바꾸고 싶어!

```
1 class LengthComparator implements Comparator<String> {
2     public int compare(String s1, String s2) {
3         return s1.length() - s2.length();
4     }
5 }
6
7 // 사용
8 Collections.sort(strings, new LengthComparator());
```

사용!

정렬 기준 함수를 클래스로 만들어서

📌 단점

- 너무 단순한 일인데 별도의 .java 파일 생성됨
- 재사용성이 떨어지는 일회성 동작인데 이름 붙인 클래스가 생성됨

➡ 번거롭고 비효율적!

익명 클래스의 사용

🤔 여전히 코드가 길고 가독성이 떨어지는데..?

익명 클래스의 인스턴스를 함수 객체로 사용

```
1 Collections.sort(words, new Comparator<String>() {  
2     public int compare(String s1, String s2) {  
3         return Integer.compare(s1.length(), s2.length());  
4     }  
5 });
```

정렬을 담당하는 추상 전략

```
new Comparator<String>()
```

➡ 두 객체를 비교하는 함수형 인터페이스

문자열을 정렬하는 구체적인 전략

```
public int compare(String s1, String s2) {  
    return Integer.compare(s1.length(), s2.length());  
}
```

➡ 익명 클래스

람다식의 등장과 사용

람다식이란?

- 메서드처럼 전달할 수 있는 **익명 함수**
- 함수형 인터페이스의 인스턴스를 람다식을 활용해 만들 수 있다.

람다식의 사용

익명 클래스의 인스턴스를 함수 객체로 사용



```
1 Collections.sort(words, new Comparator<String>() {  
2     public int compare(String s1, String s2) {  
3         return Integer.compare(s1.length(), s2.length());  
4     }  
5 });
```

람다식을 함수 객체로 사용



```
1 Collections.sort(words,  
2     (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

비교자 생성 메서드 적용



```
1 Collections.sort(words, comparingInt(String::length));
```

자바 8 sort 메서드 적용



```
1 words.sort(comparingInt(String::length));
```

상수별 클래스 몸체와 데이터 사용 열거 타입



```
1 public enum Operation {
2     PLUS("+") {
3         public double apply(double x, double y) { return x + y; }
4     },
5     MINUS("-") {
6         public double apply(double x, double y) { return x - y; }
7     },
8     TIMES("*") {
9         public double apply(double x, double y) { return x * y; }
10    },
11    DIVIDE("/") {
12        public double apply(double x, double y) { return x / y; }
13    };
14
15    private final String symbol;
16
17    Operation(String symbol) { this.symbol = symbol; }
18
19    @Override
20    public String toString() { return symbol; }
21    public abstract double apply(double x, double y);
22 }
```

함수 객체를 인스턴스 필드에 저장해 구현한 열거 타입



```
1 public enum Operation {
2     PLUS("+", (x,y) -> x + y),
3     MINUS("-", (x,y) -> x - y),
4     TIMES("*", (x,y) -> x * y),
5     DIVIDE("/", (x,y) -> x / y);
6
7     private final String symbol;
8     private final DoubleBinaryOperator op;
9
10    Operation(String symbol, DoubleBinaryOperator op) {
11        this.symbol = symbol;
12        this.op = op;
13    }
14
15    @Override
16    public String toString() { return symbol; }
17
18    public double apply(double x, double y) {
19        return op.applyAsDouble(x, y);
20    }
21 }
```

```
System.out.println(Operation.PLUS.apply(x, y));
```

람다식의 특징

람다식의 특징

- 1 함수형 인터페이스에서만 사용된다.

```
1 @FunctionalInterface
2 interface Printer {
3     void print(String message);
4 }
5
6 // 람다식 사용 가능
7 Printer printer = msg -> System.out.println("출력: " + msg);
8
9 printer.print("Hello!"); // 출력: 출력: Hello!
```

추상 메서드가 딱 하나만 있는 인터페이스

- 추상 클래스의 인스턴스를 만드는 경우
- 추상 메서드가 여러 개의 인터페이스의 인스턴스를 만드는 경우
- ➔ 익명 클래스를 사용해야 한다.

람다식의 특징

- **2** 람다는 자기 자신을 참조할 수 없다.
람다에서의 `this` 키워드는 바깥 인스턴스를 가리킨다.

```
1 public class LambdaThisExample {
2     private String name = "OuterClass";
3
4     public void run() {
5         Runnable r = () -> {
6             System.out.println("람다 this.name = " + this.name); // 바깥 클래스 참조
7         };
8         r.run();
9     }
10
11     public static void main(String[] args) {
12         new LambdaThisExample().run();
13     }
14 }
```



```
1 람다 this.name = OuterClass
```

요약 정리

요약 정리

- 랩다는 “**함수형 인터페이스**”에서만 사용해야 한다
- 익명 클래스는 “함수형 인터페이스가 아닌” 타입의 인스턴스를 만들 때 사용해야 한다.
- 랩다는 자기 자신을 참조할 수 없다. 랩다의 `this`는 바깥 인스턴스를 가르킨다.
- 코드가 짧고 간결할수록 랩다가 적합하다
 - 복잡한 로직, 상태 유지, 자기 참조가 필요한 경우 익명 클래스를 사용하는 것이 더 명확할 수 있다.

- The End -