모던 자바인 액션 3장

유성민

람다의 특징

- 익명 - 보통의 메서드와 달리 이름이 존재하지 않는다.
- 함수 - 특정 클래스에 종속되지 않으므로 함수이다.
- 전달 - 람다 표현식을 메서드 인수로 전달하거나 변수로 저장할 수 있다.
- 간결성 - 익명 클래스보다 훨씬 간결하다.

```
Comparator<Apple> byWeight = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

파라미터 리스트

람다 바디, 람다의 반환값에 해당하는 표현식

함수형 인터페이스 (Functional Interface)

- 추상 메서드를 딱 하나만 가지고 있는 인터페이스
- @FunctionalInterface 어노테이션을 가지고 있는 인터페이스
- 함수형 인터페이스도 여러개의 디폴트 메서드를 가질수 있다

함수 디스크립터 (Function Descriptor)

- 람다 표현식의 시그니처를 서술하는 메서드
- 메서드 시그니쳐: 메서드의 특성과 이름, 전달인자, 반환값의 데이터 타입을 표현한 형태

```
T → boolean // boolean method(T t);
(int, int) → void // void method(int n1, int n2)
```

함수형 인터페이스 - Predicate

- 논리 판단을 해주는 함수형 인터페이스
- 입력을 받아서 boolean 타입 출력을 반환한다
- 매개변수와 boolean 리턴값이 있는 test() 메서드를 가지고 있다

```
public class PredicateTest {
  public static void main(String[] args) {
    List<String> listOfStrings = new ArrayList<>();
    listOfStrings.add("hello");
    listOfStrings.add("bye");

    Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
    List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
    System.out.println("nonEmpty = " + nonEmpty);
  }

  public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for (T t : list) {
      if (p.test(t)) {
        results.add(t);
      }
    }
    return results;
  }
}
```

Consumer

- 파라미터 입력을 받아서 그것을 소비하는 함수형 인터페이스이다.
- 소비라는 것은 함수가 이용된다 라고 생각하면 된다. 리턴이 되지 않고 함수 내에서 사용이 되고 새로운 출력으로 되는게 아니고 없어진다.

```
public class ConsumerTest {
  public static void main(String[] args) {
    forEach(
      Arrays.asList(1, 2, 3, 4, 5),
      (Integer i) -> System.out.println(i)      //Consumer의 accept 메서드를 구현하는 람다
    );
  }

  public static <T> void forEach(List<T> list, Consumer<T> c) {
    for (T t : list) {
      c.accept(t);
    }
  }
}
```

Function

- 입력 -> 출력을 연결하는 함수형 인터페이스
- 입력 타입과 출력 타입은 다를 수 있다
- 매개값을 리턴값으로 매핑하는 역할을 한다
- apply() 메서드 사용

```
public class FunctionTest {
  public static void main(String[] args) {
    List<Integer> integerList = map(
      Arrays.asList("lambdas", "in", "action"),
      (String s) -> s.length()      //Function의 apply 메서드를 구현하는 람다
    );
    System.out.println("integerList = " + integerList);
  }

  public static <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for (T t : list) {
      result.add(f.apply(t));
    }
    return result;
  }
}
```

기본형 특화

자바에서 박싱, 언박싱, 오토박싱이 일어날때 변환 과정에서 당연히 비용이 소모된다. 이를 피할 수 있도록 자바 8에서는 기본형을 입출력으로 사용하는 특별한 버전의 함수형 인터페이스를 제공한다.

```
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);       //박싱 없음

Predicate<Integer> oddNumbers = (Integer i) -> i % 2 != 0;
oddNumbers.test(1000);        //박싱
```

일반적으로 위처럼 인터페이스 이름 앞에 IntPredicate, DoublePredicate 등 형식명이 앞에 붙는다.

형식 검사 (Type Checking)

```
@FunctionalInterface
public interface MyFunction {
    int apply(int x, int y);
}
```

이 함수형 인터페이스는 하나의 추상 메서드 apply 를 가지고 있다. 이 메서드는 두개의 int 를 인수로 받고 int 를 리턴한다. 이 함수형 인터페이스를 람다로 구현하면 아래의 코드가 완성된다.

```
MyFunction add = (x, y) -> x + y;
```

그리고 이 람다 표현식은 아래의 익명 클래스와 같다.

```
MyFunction add = new MyFunction() {
    public int apply(int x, int y) {
        return x + y;
    }
};
```

우리는 이 람다가 사용되는 컨텍스트에서 람다의 형식을 추론할 수 있다. 여기서 람다 표현식의 타입은 'MyFunction' 이다.

형식 추론 (Type Inferring)

자바 컴파일러는 람다 표현식이 사용된 콘텍스트(대상 형식)을 이용해서 람다 표현식과 관련된 함수형 인터페이스를 추론한다. 대상 형식을 이용해서 함수 디스크립터를 알 수 있으므로 컴파일러는 람다의 시그니처도 추론할 수 있다.

```
Comparator<Apple> c = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());   //형식을 추론하지 않음

Comparator<Apple> c = (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());   //형식을 추론함
```

메서드 참조

메서드 참조를 이용하면 기존의 메서드 정의를 재활용해서 람다처럼 전달할 수 있다. 때로는 이 방법이 코드의 가독성이 더 좋고 자연스러울 수 있다.

```
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));

inventory.sort(comparing(Apple::getWeight));     //메서드 참조
```

또한, 특정 인스턴스의 메소드를 참조할 때에도 참조 변수의 이름을 통해 메소드 참조를 사용할 수 있다.

```
MyClass obj = new MyClass;
Function<String, Boolean> func = (a) -> obj.equals(a); // 람다 표현식
Function<String, Boolean> func = obj::equals(a);       // 메소드 참조
```

생성자 참조

단순히 객체를 생성하고 반환하는 람다 표현식은 생성자 참조로 변환할 수 있다.

객체를 생성하고 반환하는 람다 표현식:

```
(a) -> { return new Object(a); }
```

생성자 참조를 사용하여 표현:

```
Object::new;
```

람다 표현식을 조합할 수 있는 유용한 메서드

Comparator 조합

역정렬 사과의 무게를 내림차순으로 정렬하고 싶다면?

```
inventory.sort(comparing(Apple::getWeight).reversed());
```

만약에 무게가 같은 두 사과가 존재할 때 다른 정렬 조건을 추가하고 싶다면?

```
inventory.sort(comparing(Apple:getWeight)
        .reversed()
        .thenComparing(Apple::getCountry));      //두 사과의 무게가 같으면 국가별로 정렬
```

이렇게 두 사과의 무게가 같을때는 국가별로 정렬하도록 **thenComparing** 을 활용할 수 있다!

람다 표현식을 조합할 수 있는 유용한 메서드

Predicate 조합

Predicate 인터페이스는 복잡한 프레디케이트를 만들 수 있도록 negate, and, or 세 가지 메서드를 제공한다.

빨각색이 아닌 사과?

```
Predicate<Apple> notRedApple = redApple.negate();
```

빨간색이면서 무거운 사과?

```
Predicate<Apple> redAndHeavyApple = redApple.and(apple -> apple.getWeight() > 150);
```

빨간색이면서 무거운 사과 또는 그냥 녹색 사과?

```
Predicate<Apple> redAndHeavyAppleOrGreen = redApple.and(apple -> apple.getWeight() > 150)
                                                   .or(apple -> GREEN.equals(a.getColor()));
```

람다 표현식을 조합할 수 있는 유용한 메서드

Function 조합

Function 인터페이스는 Function 인스턴스를 반환하는 andThen, compose 두 가지 디폴트 메서드를 제공한다.

andThen 메서드는 주어진 함수를 먼저 적용한 결과를 다른 함수의 입력으로 전달하는 함수를 반환한다. 아래 코드를 살펴보자:

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g);

int result = h.apply(1);
```

위 코드에서 result의 값은 4 이다. 수학적으로 위 코드를 살펴보면 g(f(x)) 로 설명할 수 있다.

compose 메서드는 인수로 주어진 함수를 먼저 실행한 다음에 그 결과를 외부 함수의 인수로 제공한다. 즉, 위 코드에서 f.andThen(g) 대신 f.compose(g)를 사용하면 g(f(x))가 아니라 f(g(x)) 라는 수식이 된다.

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);

int result = h.apply(1);
```

위 코드에서 result의 값은 3 이다. g 가 먼저 실행된 후 그 값이 f 의 인수로 주어지는 것이다.