3장 - 람다 표현식

이 장의 내용

- 람다란 무엇인가?
- 어디에, 어떻게 람다를 사용하는가?
- 실행 어라운드 패턴
- 함수형 인터페이스, 형식 추론
- 메서드 참조
- 람다 만들기

람다

> 메서드로 전달할 수 있는 익명함수를 단순화 한 것

> 함수형 인터페이스를 인자로 받는 곳에서 람다 표현식을 사용할 수 있다.

> (parameters) → expression
> (parameters) → { statements; }

특징

- **익명**
- **함수** : 특정 클래스에 종속되지 않으므로 **함수**라고 부른다. 하지만 메서드처럼 파라미터 리스트, 바디, 반환 형식, 가능한 예외 리스트를 포함한다.

- **전달** : 메서드 인수로 전달하거나 변수로 저장할 수 있다.
- **간결성**

함수형 인터페이스라는 문맥에서 람다 표현식을 사용할 수 있다.

함수형 인터페이스

예전 예제 Predicate<T>는 함수형 인터페이스다. 오직 하나의 추상 메서드만 지정하기 때문이다.

```
public interface Predicate<T>{
  boolean test(T t);
}

public interface Comparator<T>{
  int compare(T o1, T o2);
}
```

> 함수형 인터페이스는 **정확히 하나의 추상 메서드를 지정하는 인터페이스**다.

→ Comparator, Runnable 등이 있다.

→ 많은 디폴트 메서드가 있더라도 추상 메서드가 오직 하나면 함수형 인터페이스이다.

람다 표현식으로 함수형 인터페이스의 추상 메서드 구현을 직접 전달할 수 있으므로,

전체 표현식을 함수형 인터페이스의 인스턴스로 취급 (기술적으로 따지면 함수형 인터페이스를 **구현**한 클래스의 인스턴스) 할 수 있다.

```
Runnable r1 = () -> System.out.println("hi 1");
process(r1);
process(()->System.out.println("hi 2"));
```

Runnable이 오직 하나의 추상 메서드 run을 정의하고 있으므로 올바른 코드이다.

함수 디스크립터

> 람다 표현식의 시그니처를 서술하는 메서드

함수형 인터페이스의 추상 메서드 시그니처는 람다 표현식의 시그니처를 묘사한다고 한다.
아직은 무슨소린지 잘 모르겠으니 읽으면서 생각해보자.

람다 표현식은 변수에 할당하거나 함수형 인터페이스를 인수로 받는 메서드로 전달할 수 있고 함수형 인터페이스의 추상 메서드와 같은 시그니처를 갖는다.

왜 함수형 인터페이스를 인수로 받는 메서드만 람다 표현식을 쓸 수 있나?

- 함수 형식을 추가하는 방법도 생각했지만, 복잡해서 안씀
- 자바 프로그래머는 이미 하나의 추상 메서드를 갖는 인터페이스에 익숙하다

람다 활용 : 실행 어라운드 패턴

실행 어라운드 패턴

- 자원처리에 사용하는 순환패턴은 자원을 열고, 처리한 후, 닫는 순서로 이뤄짐
- 실제 자원을 처리하는 코드를 설정과 정리 두 과정이 둘러싸는 형태를 갖는다.

기존 실행 어라운드 패턴

1. 원래 함수가 있음 (고쳐야할 대상)
2. 함수형 인터페이스 정의 후 1에서 인자로 전달
3. 인자를 쓸 곳에 p.process처럼 적기 (filter할 내용을 사용)
4. 람다식으로 넘겨주기

함수형 인터페이스 사용

자바 API는 Comparable, Runnable, Callable 등의 다양한 함수형 인터페이스를 포함한다.

Predicate

- test라는 추상 메서드를 정의한다.
- T객체를 받아 불리언을 반환한다.

Consumer

- accept라는 추상 메서드를 정의한다.
- T객체를 받아서 void를 반환한다.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

public <T> void forEach(List<T> list, Consumer<T> c) {
    for(T t: list){
      c.accept(t);
    }
}
forEach(
    Arrays.asList(1,2,3,4,5),
    (Integer i) -> System.out.println(i)
);
```

Function

- 제네릭 T를 인수로 받아서 제네릭 R을 반환하는 추상 메서드 apply 정의
- 입력을 출력으로 매핑하는 람다를 정의할때 Function을 사용할 수 있다.

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t);
}

public <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T t: list){
      result.add(f.aplly(t));
    }
    return result;
}
// [ 7, 2, 6]
List<Integer> l = map(
    Arrays.asList("lambdas", "in", "action"),
    (String s) -> s.length()
);
```

박싱 언박싱

- 박싱 : primitive type → reference type
- 언박싱 : reference type → primitive type
- 오토박싱 : 자동으로 이뤄짐

박싱한 값은 힙에 저장돼서 메모리를 더 소비하고 탐색할때 메모리를 탐색해야함

→ 자바 8은 기본형 입출력 사용하는 상황에서 오토박싱을 피할 수 있도록 특별한 버전의 함수형 인터페이스를 제공한다.

형식 검사,형식 추론, 제약

람다로 함수형 인터페이스의 인스턴스를 만들 수 있는데, 어떤 함수형 인터페이스를 구현하는지 정보가 포함되어있지 않다. 어떻게 이렇게 될까?

형식 검사

```
List<Apple> heavyThan150g = filter(intventory, (Apple apple) -> apple.getWeight() > 150);
```

1. filter 메서드의 선언을 확인한다
2. filter 메서드 두 번째 파라미터로 Predicate<Apple> 형식(대상 형식)을 기대한다.
3. Predicate<Apple>은 test라는 한 개의 추상 메서드를 정의하는 함수형 인터페이스다.
4. test 메서드는 Apple을 받아 boolean을 반환하는 함수 디스크립터를 묘사한다.
5. filter 메서드로 전달된 인수는 이와 같은 요구사항을 만족해야 한다.

형식 추론

대상형 식을 이용해서 함수 디스크립터를 알 수 있으므로 컴파일러는 람다의 시그니처도 추론할 수 있다.

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

이와 같이 형식을 추론할 수 있다.

지역 변수 사용

람다도 자유변수(파라미터로 넘겨진 변수가 아닌 외부에서 정의된 변수)를 활용할 수 있다.

- 자유변수는 final이나, final처럼 변하지 않는 놈을 써야만 한다.

❓왜 그런감

- 인스턴스 변수는 힙에 저장되는 반면, 지역변수는 스택에 위치한다.

- 람다가 스레드에서 실행된다면, 변수를 할당한 스레드가 사라져서 변수 할당이 해제됐음에도 사용될려면, 지역 변수의 복사본을 제공받는데, 이 값은 바뀌지 않아야 하므로 제약이 생긴다.

❓클로저에 부합하는가

- 클로저는 비지역 변수를 자유롭게 참조할 수 있는 함수의 인스턴스를 가르킨다.
- 람다는 외부의 변수에 접근할 수 있지만, 람다가 정의된 메서드의 지역 변수의 값은 바꿀수 없다.

메서드 참조

- 기존 메서드 정의를 재활용해서 람다처럼 전달할 수 있다.

```
// 이거에서
imventory.sort((Apple a1, Apple a2)->a1.getWeight().compareTo(a2.getWeight()));

// 이렇게 변신!
inventory.sort(comparing(Apple::getWeight));
```

이렇게 가독성을 높여준다.

메서드를 참조하는 람다를 더 편리하게 표현하는 문법으로도 볼 수 있다.

람다가 굉장히 간단해지는데, 잘 사용하기 위해서는 함수 인터페이스가 어떤것이 있는지 잘 알아야할거같다.

```
static Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
  map.put("apple", Apple::new);
  map.put("orange", Orange::new);
  // 등등 여러 클래스의 생성자들
```

```
}

public static Fruit giveMeFruit(String fruit, Integer weight){
  return map.get(fruit.toLowerCase()).apply(weight);
}
```

위와 같이 String과 Integer가 주어졌을 때 다양한 무게를 갖는 여러 종류 과일을 만드는 메서드를 만들 수 있다.

람다, 메서드 참조 활용하기

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

위 코드 만들어보기!

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

Comparator는 Comparable 키를 추출해서 Comparator 객체로 만드는 Function 함수를 인수로 받는 정적 메서드 comparing을 포함한다. 다음처럼 comparing 메서드를 사용할 수 있다. (람다 표현식은 사과를 비교하는데 사용할 키를 어떻게 추출할 것인지 지정하는 한 개의 인수만 포함한다)

```
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

이제 코드를 다음처럼 간소화 할수있다.

```
import static java.util.Comparator.comparing;
inventory.sort(comparing(apple -> apple.getWeight()));
```

이제 메서드 참조로 표현하면 맨 위의 코드가 완성된다!

람다 표현식을 조합할 수 있는 유용한 메서드

여러 개의 람다 표현식을 조합해서 복잡한 람다 표현식을 만들 수 있다.

→ 디폴트 메서드가 가능하게 만든다.

역정렬을 할 때 디폴트 메서드를 이용해서 구현 가능하다.

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

reversed가 디폴트 메서드로 돼있기 때문에 가능하다.

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

이런식으로 thenComparing으로 무게가 같은 녀석들을 두번째 비교자로 비교할수도 있다.