Chapter 3

람다 표현식

요약

- 개요
- 어디서 어떻게 람다식을 쓰는지
- 실행 패턴
- 함수적 / 타입 인터페이스
- 람다 구성

개요

`람다 표현식` 은 매개변수로 보낼 수 있는 익명 메소드의 간략한 표현법이라고 볼 수 있다. 이름이 없지만 파라미터, 내용, 리턴 타입, 예외를 가진다.

- 익명성 : 일반적으로 가지는 메소드 이름이 없기 때문에 익명성을 갖는다.
- 함수 : 람다는 클래스가 아니고, 파라미터, 내용, 리턴 타입, 예외를 갖기 때문에 함수라고 한다.
- 매개변수 : 람다식은 메소드로 매개변수로 전달될 수 있다.
- 간략함 : 보통의 익명 클래스들처럼 복잡하게 작성할 필요가 없다.

`람다식` 의 기원은 계산을 설명하는 람다 계산식이라는 시스템에서 왔다. 왜 람다식을 사용할까? 이전 챕터에서 본 내용들은 자바에서 굉장히 복잡해 보인다. 람다는 이 문제에 대해서 간편하게 메소드에 보낼 수 있다.

람다를 통해서 `Comparator` 객체를 간편하게 작성할 수 있다.

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

- 파라미터 : `compare` 메소드로 전달되는 파라미터이다.
- 화살표 : 파라미터와 람다 바디를 분리한다.
- 람다 바디 : 메소드의 동작부분

이러한 문법은 C#이나 스칼라에서도 사용되고, Javascript에도 유사한 구문이 있다. 람다 스타일은 크게 두 가지로 나뉜다.

1. expression-style `(parameters) → expression`
2. block-style `(parameters) → { statements; }`

어디서 어떻게 사용하는가

람다는 기능적인 인터페이스가 필요할 때 사용한다.

기능적 인터페이스

기능적 인터페이스란 특정한 단 한개의 추상 메소드이다. `Comparator` 이나 `Runnable` 이 예시가 된다.

함수 디스크립터

함수형 인터페이스가 가지는 추상 메소드의 시그니처는 람다 표현식을 서술한다. 이것을 `함수 디스크립터` 라고 한다. 예를 들어, `Runnable` 인터페이스는 아무것도 하지 않고, 리턴하지 않는 함수의 시그니처로 볼 수 있다. `run` 이라는 추상메소드가 단 한개의 추상 메소드며, 아무것도 리턴하지 않기 때문이다.

ex

`() -> void` 는 파라미터가 없고, void를 리턴한다는 의미다.

`(Apple, Apple) -> int` 는 2개의 Apple 타입 파라미터를 갖고, int 타입을 리턴한다는 의미다.

람다를 함수형 인터페이스가 필요한 곳에만 사용하는 이유는 복잡하게 사용하지 않은 그대로도 기능에 잘 맞고, Java 프로그래머들이 이미 단일 추상 메소드에 익숙하기 때문이다. (event handling 등) 가장 중요한 이유는, Java 8 이전에도 쓰였기 때문이다. 이는 람다식 표현으로의 전환에 좋은 기회가 된다.

실행 어라운드 패턴

자원 처리에서의 `recurrent pattern` 은 자원의 사용, 처리, 종료 동작을 한다 (File 및 DB 사용). 준비 과정에서는 항상 비슷하고 처리 과정에 있어서 중요한 역할을 한다. 이를 실행 어라운드 패턴이라고 부른다.

```
public String processFile() throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
                return br.readLine();
        }
}
```

현재 코드에는 제한이 있다. 파일에 가장 첫 번째 줄만 읽을 수 있다. 이 다음을 읽으려면 `processFile` 에 다른 동작을 하도록 지정해야 한다. 이는 `BufferedReader` 에 다른 동작을 하도록 하는 것과 같다.

동작 파라미터화는 람다가 하는 일과 일치한다. `Lambda` 로 표현하자면 `BufferedReader` 를 보내서 `String` 을 반환하도록 하는 것과 같다. 람다 표현식은 `인라인` 형식으로 함수적 인터페이스의 추상 메소드를 제공하는 방법이다.

람다 전달

다른 람다를 전달해서 기존의 `processFile` 을 재사용할 수 있다.

```
String oneLine = processFile((BufferedReader br) -> br.readLine());
String twoLines = processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

함수적 인터페이스를 활용하는 것

`함수적 인터페이스`는 단 한 가지의 `추상 메소드` 를 특정한다. 그리고 람다 표현식을 서술하는 추상 메소드로써도 유용하다. 이렇게 사용되는 함수적 인터페이스를 `함수 디스크립터` 라고 한다. 서로 다른 람다 표현식을 사용하기 위해서, 일반적인 함수 디스크립

터를 설명할 수 있는 `함수적 인터페이스 집합` 을 사용해야 한다. `Comparable, Runnale, Callable` 과 같은 Java API가 이미 등록되어 있다.

Example

`Predicate<T>` , `Consumer<T>` , `Function<T, R>`

`T` 타입은 `reference` 타입만 파라미터로 사용할 수 있다. 결과적으로, 타입 분류가 다른 파라미터들에 대해서는 `autoboxing` 이 이루어진다. 따라서, boxing된 값들은 메모리 영역 (heap)을 추가로 사용하게 된다.

이러한 상황에 따라서, `autoboxing` 을 피하기 위해서 특별한 버전의 `함수적 인터페이스` 를 제공한다.

```
public interface IntPredicate {
        boolean test(int t);
}
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);
Predicate<Integer> oddNumbers = (Integer i) -> i % 2 != 0;
oddNumbers.test(1000);
```

위의 코드에서, `IntPredicate` 는 boxing을 피하고, `Predicate<Integer>` 는 Integer 타입을 boxing 한다.

`lambda` 를 다음과 같이 사용할 수 도 있다.

```
// Boolean
(List<String> list) -> list.isEmpty();
//Create
() -> new Apple;
//Consuming
(Apple a) -> System.out.println(a.getWeight());
// Select
(String s) -> s.length();
// Combine
(int a, int b) -> a * b;
// Compare
(Apple a1, Apple a2) -> a.getWeight().compareTo(a2.getWeight);
```

람다의 다른 기능

람다 표현식 자체로는 어떤 함수적 인터페이스가 정보를 가지고 있는지 알 수 없다. 람다 표현식에 대해 이해하기 위해, 어떤 타입의 람다가 있는 지 알아야 한다.

타입 체크

람다 타입은 람다가 사용되는 유형에 따라 추론할 수 있다. (`메소드 파라미터` , `지역 변수` 등)

이렇게 사용된 타입들은 `target type` 이라고 한다. 다음의 예제 코드를 보자.

```
List<Apple> heavierThan150g =
                                                    filter(inventory, (Apple apple) -
> apple.getWeight() > 150);
```

타입 체크는 다음의 과정을 따른다.

1. `filter` 가 선언됨을 확인한다.
2. 파라미터의 형식이 `target type` 임을 확인한다.
3. `Predicate<Apple>` 은 test라는 단일 추상 메소드인 `함수적 인터페이스`다.
4. test 메소드는 `boolean` 을 리턴하는 `함수 디스크립터` 의 내용이 들어 있다.
5. `filter` 메소드는 이 요구 조건에 해당해야 한다.

같은 람다, 다른 함수적 인터페이스

`타겟 타이핑` 때문에 같은 람다 표현식은 다른 함수적 인터페이스를 구현할 수 있다.

예를 들어, `Callable` 과 `PrivilegedAction` 은 파라미터가 없고, generic typed을 리턴하는 동작을 한다.

타입 추론

자바 컴파일러는 target type으로 사용된 람다 함수적 인터페이스가 어떤 것인가 추론하고, 또한 적절한 시그니처를 추론할 수 있다. 이는 함수 디스크립터가 target type 이기 때문이다. 장점은 컴파일러가 람다 표현식의 파라미터에 접근할 수 있고, 람다 구문에 의해 생략될 수도 있다.

다음의 `Comparator` 예시를 보자. 이 예시에서는 람다 구문에 의한 생략이 코드를더 읽기 쉽게 만들어 준다는 내용을 포함한다.

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

어떤 경우에는 타입을 명시하는 편이 코드를 이해하기 쉽고, 어떤 경우에는 타입을 명시하지 않은 편이 이해하기 쉽다. 무엇이 더 낫다는 규칙은 없으며, 더 읽기 쉬운 코드를 위해 스스로 결정해야 한다.

지역 변수 사용

지금까지 살펴 본 모든 람다 함수는 내용 안의 받는 파라미터가 되었다. 그러나 람다 표현식은 `free variables` 를 지원한다. (파라미터가 아니며 메소드 외부에 선언됨) `익명 클래스` 가 비슷한 예다. 이러한 람다식은 `capturing lambda` 라고 부른다.

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

이 코드에는 변수로 가능한 약간의 규칙이 있다. 람다는 규칙 없이도 지역 변수와 전역 변수를 사용할 수 있었다. 그런데 지역 변수들이 잡히면, 이 변수들은 명시적 / 최종적으로 `final`로 선언되어야 한다. 람다 표현식이 가지는 지역 변수는 한 번만 할당될 수 있기 때문이다.

그렇다면 왜 로컬 변수들에게 이런 제약이 있을까?

1. 인스턴스와 로컬 변수가 구현되는데에 있어서 핵심적인 차이점이 있다. `인스턴스 변수` 는 `heap` 영역에 저장되고, `local 변수` 는 `stack` 영역에 저장된다. 만일 람다가 스레드에서 사용되며 지역 변수로 바로 접근이 가능하다면, lambda를 사용하는 스레드는 이전에 할당이 해제되었다가 재할당이 된 변수에 접근하게 된다. 하지만, 자바 구성은 로컬 원본이 아닌 복사본을 가지고 접근하게 된다.
2. 규약은 일반적인 명령형 프로그램 패턴에 동의하지 않는다. `Closer` 는 지역 변수가 아닌 것들을 참조하는 함수 인스턴스이다. 예를 들어, `closer` 은 다른 함수로 argument로서 전달될 수 있다. 범위 외의 변수에 대해 접근과 수정도 가능하다. 이제, Java 8의 등장으로 람다와 익명 클래스는 이와 비슷한 기능을 하게 되었다. 이들은 argument로서 범위 외의 변수에 접근이 가능하지만, 람다가 선언된 클래스와 메소드들은 변수의 직접 수정이 불가능하다. 따라서 이 변수들은 `final` 로 선언되어야 한다. 이 규약들은 지역 변수들인 stack에 있고 암묵적으로 제한이 걸린 스레드에 있다. 변경 가능한 지역의 변수를 바꿔버린다면 스레드에 안정하지 못한 결과가 발생할 수 있다.

메소드 참조

메소드 참조는 존재하는 메소드 정의를 재사용하고 람다로 전달할 수 있도록 해준다. 몇몇 상황에서는 더 읽기 쉽고, 사용하기에도 편하다.

```
// before
inventory.sort((Apple a1, Apple a2)
a1.getWeight().compareTo(a2.getWeight()));

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

정리 - 왜 메소드 참조를 사용해야 하는가?

메소드 참조는 특정 메소드에서 호출 가능한 짧은 구문이다. 기본적으로 람다는 `이 메소드르 직접 호출해라` 라고 나타낸다. 이것은 어떻게 호출할지 직접 명시하는 것 보다 훨씬 편리하다. 메소드 참조는 존재하는 함수 구현체로부터 람다 표현식을 만들도록 한다. 명시적으로 함수 이름을 작성해서, 이해하기에 편하게 만들어준다. `타겟 참조` 는 `::` 이전에 위치하고, 메소드 이름은 그 이후에 위치한다. 예를 들어, `Apple :: getWeight` 은 `Apple` 클래스에 정의된 `getWeight` 를 참조하라는 것과 같다. 이 메소드 참조는 `(Apple apple) -> apple.getWeight();` 과 같은 동작을 한다.

메소드 참조의 종류

1. `static method` 참조 `Integer :: parseInt`
2. 임의의 유형 참조 `String :: length`
3. 객체 내에 존재 참조 `expensiveTransaction :: getValue`

2번째의 경우는 람다의 파라미터로 사용될 객체의 내용에 사용된다. 예를 들어, `String s -> s.toUpperCase()` 는 `String :: toUpperCase` 와 같다. 그러나 3번째 경우에는 이미 존재하는 외부 오브젝트의 메소드를 호출할 때 사용한다. 예를 들어, `() -> expensiveTransaction(getValue()` 는 `expensiveTransaction :: getValue` 로 사용될 수 있다. 이러한 종류의 메소드 참조는 private로 작성된 메소드를 처리할 때 유용하다.

구조체 참조

이미 존재하는 구조체를 참조할 때 `구조체의 이름`과 `new`를 사용할 수 있다. 이는 정적 메소드와 비슷하게 동작한다. 예를 들어, 파라미터가 없는 경우에는 다음과 같이 작성할 수 있다.

```
//아래 두 코드는 같은 동작을 하는 코드이다
Supplier<Apple> c1 = Apple::new;
Apple a1 = c1.get();

Supplier<Apple> c1 = () -> new Apple();
Apple a1 = c1.get();
```

만약 파라미터가 있는 생성자라면, 다음과 같은 사용도 가능하다.

```
//아래 두 코드는 같은 동작을 하는 코드이다
Function<Integer, Apple> c2 = Apple::new;
Apple a2 = c2.apply(110);

Function<Integer, Apple> c2 = (weight) -> new Apple(weight);
Apple a2 = c2.apply(110);
```

각 `List` 와 `Integer` 는 `map`과 유사한 역할로 묶여 있다.

```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
List<Apple> apples = map(weights, Apple::new);
public List<Apple> map(List<Integer> list, Function<Integer, Apple> f) {
        List<Apple> result = new ArrayList<>();
                for(Integer i: list) {
                        result.add(f.apply(i));
                }
          return result;
}
```

만약 생성자에 들어갈 파라미터가 2개라면, 다음과 같이 작성할 수 있다.

```
// 아래 두 코드는 같은 도작을 한다.
BiFunction<Color, Integer, Apple> c3 = Apple::new;
Apple a3 = c3.apply(GREEN, 110);

BiFunction<String, Integer, Apple> c3 = (color, weight) -> new Apple(color, weight);
Apple a3 = c3.apply(GREEN, 110);
```

Code Example - sort

1. 코드 전달

```
public class AppleComparator implements Comparator<Apple> {
        public int compare(Apple a1, Apple a2){
                return a1.getWeight().compareTo(a2.getWeight());
        }
```

```
}
inventory.sort(new AppleComparator());
```

2. 익명 클래스 사용

`Comparator` 를 한번만 사용하기 위한 목적으로 구현하는 것 보다, 익명 클래스를 사용하면 재사용이 가능하다.

```
inventory.sort(new Comparator<Apple>() {
        public int compare(Apple a1, Apple a2){
                return a1.getWeight().compareTo(a2.getWeight());
        }
});
```

3. 람다식 사용

여전히 코드는 복잡하다. 같은 동작을 하는 경량화된 문법을 제공하기 때문에 이를 사용하는 것이 좋다. 이 경우에는, 함수 디스크립터를 `(T, T) -> int` 로 나타낼 수 있다. 여기서는 `Apple` 타입을 사용하기 때문에 고쳐 말하면 `(Apple, Apple) -> int` 로 쓰는 것과 같다.

앞서 자바 컴파일러는 람다 표현식의 파라미터를 람다가 쓰여진 구문을 보고 추론한다고 했다. 그러므로 다음과 같이 고쳐쓸 수 있다.

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

`Comparator` 는 정적 메소드인 `comparing` 을 포함하며, `Comparable` 키와 객체를 가진다. 따라서 다음과 같이 표현할 수 있다.

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

결과적으로는 다음과 같이 표현할 수 있다.

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

4. 메소드 참조 사용

메소드 참조는 람다식의 꽃이다. 위의 코드를 더 간략화해서 다음과 같이 작성할 수 있다.

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

람다 표현식을 만들기 위한 유용한 메소드들

`Comparator` , `Function` , `Predicate` 와 같이 람다 표현식으로 사용되는 함수적 인터페이스들은 구성을 허가하는 메소드를 제공한다. 작은 메소드들을 결합하거나 연산을 하는 경우인데, 이 외에도 메소드를 다른 메소드로 전달해서 함수적 인터페이스를 만드는 것도 가능하다. 이것은 `default method` 라고 한다. (CH.13)

Example - `Comparator`

Reversed order

기본적으로 제공하는 `sort` 메소드는 오름차순이다. 이 결과를 내림차순으로 바꾸기 위해서는 개발자가 새로운 메소드를 만들 필요가 없다. 단지 `reversed()` 메소드를 사용하기만 하면 된다.

이렇게 람다식과 내부 메소드를 결합하여 새로운 함수적 인터페이스를 구현한 것과 같은 효과를 준다.

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

Changing comparators

비교 조건을 여러개 둘 때 사용하는 방법이다. 두번째 비교선택자를 추가할 때는 `thenComparing()` 메소드를 사용하면 된다. 즉, 동적 비교 메소드를 만들지 않고도 다중 조건에 의한 비교 메소드를 구현할 수 있다.

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

Example - `Predicates`

`Predicate` 인터페이스는 이미 존재하는 객체에 대해 상태를 반환하는 `negate`, `and`, `or`를 가진다.

특정 조건에 부합하는 객체를 가져올 때, 조건들을 결합해서 사용할 수 있다.

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

간단한 람다 표현식으로 복잡한 표현을 할 수 있지만 읽는데에는 훨씬 간편하다. 그리고 위에서부터 순차적으로 처리되는 것을 기억해두어야한다. (순서와 결과가 관련 있음)

Example - `Functions`

`Function` 인터페이스는 `andThen` 과 `compose` 로 구성되고, `Function` 타입을 리턴한다.

```
// f -> g 연산 = 4
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);

// g -> f 연산 = 3
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);
```