Java8 정리해봤습니다.

01. 람다

http://multifrontgarden.tistory.com/124

02. 함수형 인터페이스

http://multifrontgarden.tistory.com/125

03. 메서드 레퍼런스

http://multifrontgarden.tistory.com/126

04. 스트림

http://multifrontgarden.tistory.com/128

05. 옵셔널

http://multifrontgarden.tistory.com/131

자바8을 도입하게되면서 공부를 하게됐네요.

이제 곧 9가 나온다는데...;; 아직 자바8에 익숙치않은 분들은 한번 확인해보시고 잘못되거나 이해안가는거 있음 말씀해주세요!

# [자바8 용어정리](http://multifrontgarden.tistory.com/121)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.09.19 18:53

***함수형 인터페이스(Functional Interface)***

추상메서드를 1개만 갖고있는 인터페이스

***동작 파라미터(Behavior Parameter)***

값이 아닌 행위를 매개변수로 전달함, 메서드를 전달한다고 생각하면 됨, template callback 패턴을 생각하면 된다.

***메서드와 함수의 차이(Method, Function)***

메서드는 클래스에 종속되어있음. 즉 파라미터를 받지않는 함수는 사실상 항상 같은 값이 반환되어야하지만 메서드의 경우 클래스에 종속적이기때문에 파라미터를 받지않음에도 항상 반환값이 다를 수 있음(인스턴스 변수, this를 이용하는 경우). 즉 this를 이용하지못하는 정적 메서드(static method)는 함수라고 할만함.

***메서드 레퍼런스(Method Reference)***

객체의 레퍼런스를 전달하여 사용하듯(call by reference) 메서드의 레퍼런스를 전달함. 엄밀히 말하면 사실 자바에서의 메서드 레퍼런스는 객체의 레퍼런스를 이용하긴 함.

***함수 디스크립터(Function Descriptor)***

메서드 시그니처를 람다 표현식 형태로 표현한 것.

boolean test(T t) 가 메서드 시그니처라면 함수 디스크립터로 표현할시 T -> boolean 과 같은 형태로 표현할 수 있음

***대상형식(Target Type)***

람다 표현식으로 구현하고자 하는 함수형 인터페이스.

Function<String> f = str -> "hello"+str;

에서 Funtion<String>이 대상형식이 됨

***자유변수(Free Variable)***

자신의 지역 바깥의 변수

***람다캡쳐(Lambda Capture)***

람다표현식에서 자유변수를 참조하는 행위

***중간 연산(Intermediate Operation)***

연산이 적용된 스트림을 반환하는 메서드. 스트림을 반환하기때문에 메서드 체이닝 형식으로 사용이 가능

***최종 연산(Terminal Operation)***

스트림에 대한 연산이 끝나는 연산. 스트림을 반환하지않음

***쇼트서킷(Short Circuit)***

&&, || 와 같은 논리연산자는 좌항에서 만족스러운 값이 반환되지않을경우 우항은 실행시키지않는다. 실행시켜봤자 결과가 달라지지않기때문이다.

variable != null && variable.hasMethod()

와 같은 비교시 variable이 null이 들어있을경우 우항의 hasMethod() 부분에서 NullPointerException이 발생하게된다. 하지만 좌항에서 null을 비교하고있고, variable이 null이라서 좌항에서 false가 뜨면 우항의 결과와 상관없이 해당 and 연산의 결과는 false이기때문에 굳이 우항을 실행하지않는다. 이때문에 NullPointerException은 발생하지않는다.

이를 쇼트서킷이라하며 자바8의 스트림연산에서도 쇼트서킷을 지원하고있다. allMatch, nonMatch, findFirst, findAny 등의 메서드는 모든 스트림요소를 검사하지않아도 중간에 false가 확정되는 경우의 수가 존재하는 연산들이다. 해당 메서드들은 쇼트서킷을 지원해 이미 실패가 확정되면 모든 스트림을 연산하지않는다.

***외부반복(External Iteration)***

코드에 직접적으로 반복이 드러나는 경우, ex) 콜렉션에 for문을 작성하는경우

***내부반복(Internal Iteration)***

코드에 반복이 드러나지않음. 스트림은 내부반복을 지원하여 코드에 반복문이 드러나지않는다.

***루프 퓨전(Loop Fusion)***

메서드 체이닝(Method Chaining)을 통해 서로 다른 연산을 연달아 하나의 연산으로 처리하는 기법

참고 : [Java8 in Action](http://book.naver.com/bookdb/book_detail.nhn?bid=8883567)

# [Java8#01. 람다 표현식(Lambda Expression)](http://multifrontgarden.tistory.com/124)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.10.02 16:43

Java8에서 뭐가 추가됐나요? 라고 물으면 가장 먼저 들리는 대답은 십중팔구 '람다와 스트림이요' 일것이다. 당연히 맞는 말이고 틀린 답은 아니지만 람다와 스트림이 왜 추가됐는지를 알아야하는데, 자바가 기존에 없던 문법까지 만들어가면서 이런것들을 추가한건 함수형 프로그래밍을 받아들이기 위해서다. 그래서 Java8을 공부하고자하는 사람이라면 그냥 단순히 '이번에 추가된 람다랑 스트림공부해야지'가 아니라 함수형 프로그래밍을 공부해야한다. 이는 프로그래밍 자체의 패러다임이 변하기때문에 그저 추가된 문법, 추가된 API만 공부하면 되는 수준이 아니라 프로그래밍 방식 자체, 문제 해결을 위한 사고방식 자체를 기존에서 탈피해야함을 의미한다. 해보니 기존에도 못하던걸 새로운걸 받아들이려고하니까 너무 어렵다;; 그래서 공부한것들을 정리해보고자 Java8의 포스팅을 시작한다.

**1. 람다 표현식(Lambda Expression)**

람다의 핵심은 지울수 있는건 모두 지우자는 것이다. 모든걸 컴파일러의 추론에 의지하고 코드로 표현하는건 다 없애버려 코드를 간결하게 만드는 것이다.

interface Movable{

void move(String str);

}

class Car implements Movable{

@Override

public void move(String str) {

System.*out*.println("gogo car move" + str);

}

}

Movable movable = new Movable() {

@Override

public void move(String str) {

System.*out*.println("gogo move move" + str);

}

};

Movable 인터페이스를 구현하고있다. 따로 Movable 인터페이스를 구현하는 클래스를 만들어 객체화하거나 재사용성이 없다면 그 자리에서 바로 생성하는 익명클래스 객체를 구현해야하는게 기존 방식이다. 어떤 방식을 쓰든 @Override 애노테이션을 제외해도 5줄은 필요하다.

재사용이 필요한 Car 클래스같은경우는 재사용을 위해 클래스로 남겨두고 익명 클래스부분을 람다표현식으로 수정해보자. 먼저 저기서 어떤부분을 지워도될지 한번 생각해보자.

1) 이미 대상타입(Target Type)에서 Movable 이라고 명시했기때문에 new Movable부분은 없어도 컴파일러가 추론할 수 있다.

2) 구현하려고보니까 구현해야할건 move() 메서드밖에 없다. 만약 구현해야할 메서드가 1개뿐이라면 메서드명칭도 없어도 되지않을까

3) 컴파일러가 인터페이스와 메서드를 추론했다면 인자도 추론할 수 있을 것이다. 구현해야할때 인자를 써야하니 인자를 아예 다 지우진 못해도 타입까지는 명시하지않아도 될 것 같다.

이를 토대로 람다표현식으로 고쳐보면

Movable movable1 = (str) -> {

System.*out*.println("gogo move move" + str);

};

이런 결과가 나온다. 여기서 더 없앨 수 있을게 있는지 생각해보자.

1) 인자가 여러개면 몰라도 1개일땐 괄호가 없어도 되지 않을까

2) 실행구문이 지금처럼 1줄일땐 중괄호가 없어도 되지 않을까

Movable movable2 = str -> System.*out*.println("gogo move move" + str);

5줄이던 코드가 1줄로 줄어들었다. 람다 표현식은 자바에서 기존에 지원하지않던 문법이 추가되어서 놀라운거지 사실 그 구현 자체가 어렵진않다.

**1-1. 함수형 인터페이스(Functional Interface)**

위 예제에서 확인한 람다 표현식은 *구현해야될 추상 메서드가 1개인 인터페이스를 구현한 것*이다. 인터페이스를 구현하고자하는데 어차피 구현해야될 메서드가 1개뿐이니 이름이고 뭐고 다 지워버린것이다. 그럼 메서드가 2개일땐 어떻게 해야할까? 람다로는 지원하지않는다. *람다 표현식으로 구현이 가능한 인터페이스는 오직 추상 메서드가 1개뿐인 인터페이스만 가능하며 그렇기때문에 추상 메서드가 1개인 인터페이스를 부르는 명칭이 추가됐다. 그것이 함수형 인터페이스다.* 그럼 가만히 생각해보자. 추상 메서드가 1개뿐인 인터페이스가 있고, 난 이걸 다른곳에서 람다 표현식으로 구현했다. 그런데 시간이 흘러 다른 동료가 해당 인터페이스를 수정할 일이 생겼고, 추상 메서드를 추가하고자한다. 인터페이스에는 당연히 추상메서드의 개수를 제한하는 방법은 없으므로 다른 곳에 람다 표현식이 있는걸 모르고 추상메서드를 추가하는 일이 발생할 수 있다. 열심히 추상 메서드를 추가하고 구현한다음 빌드를 하는데 여기저기서 빌드가 깨지는걸 보면 그 동료(후임자)는 멘탈이 붕괴될 것이다. 이런 불상사를 막기 위해 *@FunctionalInterface 애노테이션*이 존재한다.

@FunctionalInterface

interface Movable{

void move(String str);

}

@FunctionalInterface 애노테이션은 해당 인터페이스가 함수형 인터페이스라는걸 알려준다. 추상메서드가 1개가 아닐경우에 컴파일에러를 내뿜는다. 아무 생각없이 추상 메서드를 추가하면 이러면 안된다는것을 알려주는 것이다. @Override 애노테이션과 마찬가지로 꼭 해당 애노테이션이 붙지 않아도 람다 표현식으로 구현할 수 있으나 이 인터페이스의 용도를 알리기위해 붙여주는것이 좋다.

**1-2. 상태가 없는 객체(Stateless Object)**

추상 메서드가 1개인 인터페이스를 구현할 때는 람다 표현식을 이용하면 매우 간결한 코드로 구현할 수 있다는 것을 알았다. 하지만 클래스를 구현할때 클래스에는 메서드(행위)만 존재하란 법은 없다. 경우에 따라서는 인스턴스 필드(상태)가 존재하는 클래스가 있을 수도 있는데 아래같은 클래스를 말한다.

Movable movable = new Movable() {

private int speed;

@Override

public void move() {

System.*out*.println("gogo move move current speed : " + speed);

}

};

익명객체를 구현할때는 인스턴스 필드를 추가할 수 있다. 하지만 람다는 바로 메서드를 구현해버리니 인스턴스 필드가 들어갈 공간이 없어보인다. 인스턴스 필드를 추가하려면 어떻게 해야할까? 방법은 없다. 메서드와 함수의 가장 큰 차이는 메서드는 객체에 종속되어있다는 것이다. 함수란 인풋(Input)에 의해서만 아웃풋(Output)이 달라져야하는데 메서드는 객체에 종속적이기때문에 인풋이 달라지지않아도 객체의 상태에 따라 결과값이 다를 수 있다. 함수형 프로그래밍에서 함수는 인풋에 의해서만 아웃풋이 달라져야하며 그것을 지원하기위해 람다 표현식으로 구현할때 객체는 상태를 가질 수 없다. 이는 추후에 알아볼 스트림(Stream)을 이용한 병렬(Parallel)적 프로그래밍시 매우 중요한 요소다.

**1-3. 행위 파라미터화(Behavior Parameterize)**

보통 코드를 짤때 데이터를 매개변수로 전달하고 해당 데이터를 가지고 무언가 행위를 하는 메서드를 구현하기 마련이다. 하지만 데이터를 전달하는 것이 아닌 행위를 전달하게되면 좀 더 유연한 코드가 될 수 있는데 '이게 뭔말인가..'싶다면 예제 코드를 봐보자.

class Fruit{

private String name;

private String color;

Fruit(String name, String color){

this.name = name;

this.color = color;

}

String getName(){

return this.name;

}

String getColor(){

return this.color;

}

}

List<Fruit> extractApple(List<Fruit> fruits){

List<Fruit> resultList = new ArrayList<>();

for(Fruit fruit : fruits){

if("apple".equals(fruit.getName())){

resultList.add(fruit);

}

}

return resultList;

}

List<Fruit> extractRed(List<Fruit> fruits){

List<Fruit> resultList = new ArrayList<>();

for(Fruit fruit : fruits){

if("red".equals(fruit.getColor())){

resultList.add(fruit);

}

}

return resultList;

}

Fruit 클래스가 있고, 해당 객체들의 리스트를 받아 특정 조건으로 추출해내는 메서드 2개다. 메서드구현부분을 잘 보면 if문 외에는 전부 동일한 걸 볼 수 있다. 이걸 행위 파라미터를 이용해 1개의 메서드로 합치면 훨씬 보기가 좋을것 같다.

static List<Fruit> extractFruitList(List<Fruit> fruits, Predicate<Fruit> predicate){

List<Fruit> resultList = new ArrayList<>();

for(Fruit fruit : fruits){

if(predicate.test(fruit)){

resultList.add(fruit);

}

}

return resultList;

}

그래서 합쳤다. 비교하는 행위를 파라미터로 입력받아 if문 내 조건을 하드코딩에서 동적으로 처리하게끔 파라미터가 1개 늘었다. 호출하는 부분은 이렇다.

List<Fruit> fruits = Arrays.*asList*(new Fruit("apple", "red"), new Fruit("melon", "green"), new Fruit("banana", "yellow"));

List<Fruit> appleList = *extractFruitList*(fruits, new Predicate<Fruit>() {

@Override

public boolean test(Fruit fruit) {

return "apple".equals(fruit.getName());

}

});

List<Fruit> redList = *extractFruitList*(fruits, new Predicate<Fruit>() {

@Override

public boolean test(Fruit fruit) {

return "red".equals(fruit.getColor());

}

});

2개의 메서드는 합쳐져서 만족스러운데 호출하는 구문이 지저분해진거같다. 메서드가 2개일때와 마찬가지로 오히려 이제는 호출부분이 return문만 제외하고 똑같은 코드가 반복된다. 그런데 잘 보니 Predicate<T>라는 인터페이스는 뭔진 모르겠지만 추상메서드가 1개뿐인것같다(test). 그럼 람다 표현식으로 고칠 수 있지 않을까?

List<Fruit> appleList = *extractFruitList*(fruits, fruit -> "apple".equals(fruit.getName()));

List<Fruit> redList = *extractFruitList*(fruits, fruit -> "red".equals(fruit.getColor()));

코드가 상당히 줄어든걸 볼 수 있다. 행위 파라미터라는 기법 자체는 기존에도 익명 클래스를 이용해 매우 활용되고 있던 기법이다. 자바 개발자라면 안쓸수가 없는 Spring Framework가 익명 클래스를 이용한 행위 파라미터를 적극 활용해 Template Callback Pattern이라는 디자인패턴까지 만들정도로 이미 유용히 사용되던 기법이다. 이것이 함수형 인터페이스와 람다 표현식으로 인해 더욱 간결하게 사용될 수 있는것이다.

# [Java8#02. 함수형 인터페이스(Functional Interface)](http://multifrontgarden.tistory.com/125)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.10.17 20:00

앞선 포스팅에서 함수형 인터페이스(Functional Interface)가 무엇인지 알아봤다. 이번 포스팅에서는 Java8에서 기본적으로 제공하는 함수형 인터페이스들을 알아보자. 보통 'API는 외울필요없다. 이런게 있다는것만 알고있으면 된다.' 이런말을 많이 하는데 개인적인 생각에 Java8에서 제공하는 API는 최대한 외울수 있는데까지는 외워야 한다고 생각한다. 다음번에 알아볼 Stream은 선언형 프로그래밍으로 API만 계속 등장하게되는데 그때마다 찾아서 쓰려면 오히려 생산성이 저하되는 일이 발생할 수도 있기때문이다.

**1) Runnable**

기존부터 존재하던 인터페이스로 스레드를 생성할때 주로사용하였으며 가장 기본적인 함수형 인터페이스다. void 타입의 인자없는 메서드를 갖고있다.

Runnable r = () -> System.*out*.println("hello functional");

r.run();

**2) Supplier<T>**

인자는 받지않으며 리턴타입만 존재하는 메서드를 갖고있다. 순수함수에서 결과를 바꾸는건 오직 인풋(input) 뿐이다. 그런데 인풋이 없다는건 내부에서 랜덤함수같은것을 쓰는게 아닌이상 항상 같은 것을 리턴하는 메서드라는걸 알 수 있다.

Supplier<String> s = () -> "hello supplier";

String result = s.get();

**3) Consumer<T>**

void 타입의 인자를 받는 메서드를 갖고있다. 인자를 받아 소모한다는 뜻으로 인터페이스 명칭을 이해하면 될듯 하다.

Consumer<String> c = str -> System.*out*.println(str);

c.accept("hello consumer");

**4) Function<T, R>**

인터페이스 명칭에서부터 알 수 있듯이 전형적인 함수를 지원한다고 보면 된다. 하나의 인자와 리턴타입을 가지며 그걸 제네릭으로 지정해줄수있다. 그래서 타입파라미터(Type Parameter)가 2개다.

Function<String, Integer> f = str -> Integer.*parseInt*(str);

Integer result = f.apply("1");

**5) Predicate<T>**

하나의 인자와 리턴타입을 가진다. Function과 비슷해보이지만 리턴타입을 지정하는 타입파라미터가 안보인다. 반환타입은 boolean 타입으로 고정되어있다. Function<T, Boolean>형태라고 보면된다.

Predicate<String> p = str -> str.isEmpty();

boolean result = p.test("hello");

**6) UnaryOperator<T>**

하나의 인자와 리턴타입을 가진다. 그런데 제네릭의 타입파라미터가 1개인걸 보면 감이 오는가? 인자와 리턴타입의 타입이 같아야한다.

UnaryOperator<String> u = str -> str + " operator";

String result = u.apply("hello unary");

**7) BinaryOperator<T>**

동일한 타입의 인자 2개와 인자와 같은 타입의 리턴타입을 가진다.

BinaryOperator<String> b = (str1, str2) -> str1 + " " + str2;

String result = b.apply("hello", "binary");

**7) BiPredicate<T, U>**

서로 다른 타입의 2개의 인자를 받아 boolean 타입으로 반환한다.

BiPredicate<String, Integer> bp = (str, num) -> str.equals(Integer.*toString*(num));

boolean result = bp.test("1", 1);

**8) BiConsumer<T, U>**

서로 다른 타입의 2개의 인자를 받아 소모(void)한다.

BiConsumer<String, Integer> bc = (str, num) -> System.*out*.println(str + " :: " + num);

bc.accept("숫자", 5);

**9) BiFunction<T, U, R>**

서로 다른 타입의 2개의 인자를 받아 또 다른 타입으로 반환한다.

BiFunction<Integer, String, String> bf = (num, str) -> String.*valueOf*(num) + str;

String result = bf.apply(5, "678");

**10) Comparator<T>**

자바의 전통적인 인터페이스 중 하나이다. 객체간 우선순위를 비교할때 사용하는 인터페이스인데 전통적으로 1회성 구현을 많이 하는 인터페이스이다. 람다의 등장으로 Comparator의 구현이 매우 간결해져 Comparable 인터페이스의 실효성이 많이 떨어진듯 하다.

Comparator<String> c = (str1, str2) -> str1.compareTo(str2);

int result = c.compare("aaa", "bbb");

# [Java8#03. 메서드 레퍼런스(Method Reference)](http://multifrontgarden.tistory.com/126)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.10.17 20:04

앞선 포스팅에서 메서드를 전달하기 위한 익명 클래스를 작성하는 반복되고 지저분한 코드를 줄이는게 람다 표현식이라고 설명했다. 메서드 레퍼런스는 이름에서 알 수 있듯이 메서드의 레퍼런스를 전달한다는 의미이며, 람다표현식에서 메서드 호출 1회로 코드가 끝나는 경우 메서드 레퍼런스를 이용해 이미 줄인 코드를 더 줄일수도있다.

메서드의 레퍼런스라고하지만 엄밀히 말해 자바에서의 메서드는 일급 객체가 아니기때문에 객체의 레퍼런스를 전달하는 방식으로 작동한다.

Function<String, Integer> f = str -> Integer.*parseInt*(str);

Function 인터페이스는 하나의 인자를 받아 다른타입의 리턴타입을 갖는 apply() 메서드를 갖고있는 함수형 인터페이스이다. 람다를 사용해 String 객체를 인자로 받아 Integer 타입으로 변환하도록 구현했다. 메서드 구현부분이 Integer.parseInt() 메서드 한번 호출로 끝나버린다. 이럴때 메서드 레퍼런스를 사용할 수 있다.

Function<String, Integer> f = Integer::*parseInt*;

Integer result = f.apply("123");

Integer 클래스의 정적 메서드인 parseInt()를 메서드 레퍼런스로 전달했다. 해당 구현체 소스에서 사용할 수 있는 지역변수는 String 타입의 str밖에 없다. 그리고 리턴타입은 제네릭을 이용해 Integer를 건네줘야하는걸 추론할 수 있다. 때문에 메서드만 전달해주면 컴파일러가 알아서 호출해주는것이다. 지금은 예제코드라 이걸 왜 쓰는지 이해가 안갈수도있지만 복잡한 코드에서 어떤 메서드를 사용했는지 선언적으로 코딩을 할 수 있도록 해준다.

Function<String, Boolean> f = String::isEmpty;

Boolean result = f.apply("123");

isEmpty() 메서드는 정적메서드는 아니다. 하지만 인자로 넘어오는 타입이 String 이기때문에 해당 타입의 메서드를 호출해준다.

메서드 레퍼런스를 사용할 수 있는 경우에 따라 사용법이나 추론법이 약간 다르다.

1) 정적 메서드 레퍼런스

위에서 본 예제이다. 호출하고자하는 정적 클래스::메서드명 형식으로 작성한다.

Function<String, Integer> f = Integer::*parseInt*;

Integer result = f.apply("123");

2) 인자 인스턴스 메서드 레퍼런스

이것 역시 위에서 본 예제이다. 인자의 타입이 명확할때 해당 타입 클래스::메서드명 형식으로 작성한다.

Function<String, Boolean> f = String::isEmpty;

Boolean result = f.apply("123");

3) 생성자 메서드 레퍼런스

Supplier<String> s = String::new;

생성자도 메서드 레퍼런스로 표현할 수 있다.

Supplier는 인자없이 리턴만 하는 메서드를 갖고있다. 메서드를 호출하면 new로 생성된 String 객체를 리턴한다. 물론 기본생성자를 호출하게 된다.

4) 바깥 인스턴스 메서드 레퍼런스

람다 캡쳐링(Lambda Capturing)을 이용해 람다 표현식 바깥에있는 인스턴스의 메서드를 호출할때 사용한다. 인자 인스턴스와는 분명히 다르다. 코드를 보고 이해하자.

String str = "hello";

Predicate<String> p = str::equals;

p.test("world");

equals() 메서드는 호출하는 객체와 인자로 넘기는 객체. 총 2개의 객체가 필요한 메서드이다. Predicate를 구현할때 인자로 넘어오는 객체는 1개뿐이기때문에

Predicate<String> p = String::equals;

이런 형태로는 작성할 수 없다. 인자로 넘어오는 String 객체의 equals() 메서드를 호출해버리면 인자로 보낼 객체를 추론할 수 없기때문이다.

바깥에 있는 str의 equals를 호출해야 인자로 넘어오는 String 객체를 equals() 메서드의 인자로 보내겠다는 의도를 추론할 수 있다.

Comparator<String> c = String::compareTo;

이런식의 작성도 할 수 있다. compareTo() 메서드를 호출하기위해선 호출하는 객체와 인자로 보내는 객체, 객체 2개가 필요한데 compare() 메서드에는 마침 2개의 인자가 넘어온다. 첫번째인자로 호출하고 두번째인자를 인자로 보내는 것이다.

예제코드만 봤을땐 특별히 어려울게 없을것같지만 필요할때 사용한다거나 남이 짠코드를 파악할때 메서드 레퍼런스를 만나면 의외로 헷갈릴일이 많을것이다. 이런일이 없게하려면 람다를 메서드 레퍼런스로, 메서드 레퍼런스를 람다로 바꾸는 연습을 많이 해보길 바란다.

# [Java8#04. 스트림(Stream)](http://multifrontgarden.tistory.com/128)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.10.19 19:58

개인적으로 자바8의 꽃이라 생각하는 스트림 포스팅이다. 내용이 워낙 방대하긴하나 쉽게 생각하면 스트림은 결국 API들의 모임이기때문에 외워야할게 많기도하다.

**1. 스트림(Stream)**

스트림은 자바8에 추가된 API로 자바의 자료구조들을 *선언적*으로 다루는 역할을 한다. 앞선 [함수형 인터페이스 포스팅](http://multifrontgarden.tistory.com/125)에서 설명했던 인터페이스들이 엄청나게 등장을 하니 스트림을 다룬다면 외우기 싫어도 외워질수밖에 없을 것이다.

자료구조들을 다루는 역할을 하기때문에 스트림은 배열이나 List처럼 생성한 다음 요소를 추가하는 형태가 아니다. 정적 팩토리 메서드(Static Factory Method)를 이용해 자료구조로부터 생성한다.

int[] numberArr = {1, 2, 3, 4, 5, 6};

List<Integer> numberList = Arrays.*asList*(1, 2, 3, 4, 5, 6);

Set<Integer> numberSet = new HashSet<>(numberList);

Arrays.*stream*(numberArr);

Stream.*of*(1, 2, 3, 4, 5, 6);

numberList.stream();

numberSet.stream();

**1-1. 스트림 vs 컬렉션(Collection)**

위에서 스트림은 자료구조들을 다룬다고했다. 하지만 이미 자바에서는 각종 자료구조들의 구현체를 손쉽게 사용할 수 있도록 util패키지를 제공하고있다. util패키지에서 제공하는 컬렉션 프레임워크는 자바의 자랑이자 강점이기도한데 무엇이 아쉬워서 스트림이 또 나온걸까? 앞선 표현에 이미 답이 있는데 *컬렉션은 자료구조들의 구현체고 스트림은 자료구조들을 다루는 역할*을 한다. 컬렉션은 데이터를 담는 것이 제역할이기때문에 제공하는 API들도 데이터를 넣었다 빼는것들이 대부분이다.

List<Integer> numbers = new ArrayList<>();

numbers.add(1);

numbers.get(0);

numbers.remove(1);

그렇기때문에 컬렉션을 이용해 코딩을 할때는 컬렉션에 데이터들을 담고, 그 컬렉션을 순회하고 꺼내면서 직접적으로 연산을 하는것이 주를 이룬다.

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5, 6);

List<Integer> evenList = new ArrayList<>();

for(int number : numbers){

if(number % 2 == 0){

evenList.add(number);

}

}

System.*out*.println(evenList);

(numbers 리스트에서 짝수를 추출하는 코드. 리스트는 요소를 추가, 삭제, 순회하는 API만 제공하기때문에 내가 어떻게 짝수를 걸러내야하는지를 짜야된다.)

이에반해 스트림은 연속된 자료들을 다루고 연산하는 API를 지원한다. 컬렉션과는 다르게 요소를 추가한다거나 삭제하는건 불가능하다. 연산을 지원하기때문에 짝수만 걸러내는것도 매우 명시적이고 선언형으로 짤 수 있다.

List<Integer> evenList = Stream.*iterate*(1, n -> n+1)

.limit(6)

.filter(number -> number % 2 == 0)

.collect(*toList*());

System.*out*.println(evenList);

아주 간단한 예제다보니 라인수에는 별 차이가 없는것 같지만 내가하고싶은일이 뭔지 명령을 내리는 코드형태이지 그 명령을 어떤식으로 처리하는지가 코드에 나타나진 않는다는 차이점이 있다. 스트림은 내부반복을 지원하기때문에 반복문같은것도 코드에서 전혀 나타나지않는다.

연산이 복잡해지면 복잡해질수록 컬렉션을 이용한 코드는 반복문이 계속 등장하고 연산을 저장할 변수들이 나타나면서 뭘 하고싶어하는지 알아보기가 힘들어지게되는 반면 스트림을 이용하면 코드량과 가독성을 한번에 취할 수 있게되는 것이다.

**1-2. 중간 연산, 최종 연산**

위에 짧은 예제코드를 보면 알 수 있겠지만 스트림을 사용한 코드는 계속해서 도트(Dot) 연산자로 메서드 체이닝(Method Chaining)을 일으킨다. 초보개발자의 경우 도트가 연속해서 나오면 코드를 이해하기 어려워하는 경우도 많은데 저런 경우는 한가지만 명확히 생각하면 된다. 메서드 체이닝 중간에 있는 메서드는 절대 void 형태가 아니며 메서드가 반환하는 어떤 객체가 도트 뒤에 나오는 메서드를 갖고있다고 보면 되는것이다.

/\* limit() 메서드는 무언가 객체를 반환할것이다.

void타입의 메서드라면 filter() 메서드를 호출할 수가 없기때문이다.

limit()이 반환하는 객체는 filter() 메서드를 갖고있는 객체다.

그리고 filter() 메서드역시 collect() 라는 메서드를 갖고있는 객체를 반환한다.

\*/

.limit(6)

.filter(number -> number % 2 == 0)

.collect(*toList*());

limit(), filter(), collect() 같은 메서드들은 전부 스트림에서 제공하는 메서들이다. 결국 *스트림 메서드들은 전부 연속해서 스트림을 반환*하고 있기때문에 저런 코드가 가능한것이다.

마지막으로 collect() 메서드는 스트림이 아니라 List 타입을 반환하기때문에 List 변수에 무언가 값을 저장하고있는걸 알 수 있는데 스트림 API는 이런식으로 스트림을 반환하는 메서드와 스트림이 아닌 값을 반환하는 메서드로 나뉜다.

그리고 계속해서 스트림을 반환하여 메서드 체이닝의 근간이 되게하는 메서드들을 중간 연산 메서드라 부르고 스트림이 아닌 값을 반환하여 메서드 체이닝을 끊는 메서드를 최종 연산 메서드라 부른다.

이 둘을 이렇게 구분짓는건 생각보다 중요한데, 최종 연산이 존재하지 않으면 중간연산들로만 이루어진 메서드 체이닝은 실행되지 않는다. 이 코드는 아무런 값도 찍히지 않는다.

Stream.*iterate*(1, n -> n+1)

.limit(6)

.peek(System.*out*::println) // 스트림을 순회하며 하나씩 요소를 꺼내 출력하는 구문

.filter(number -> number % 2 == 0);

중간 연산이고 최종 연산이고 중간에 출력문을 하나 넣어놨으니 이 코드는 실행하면 출력문이 나와야 될듯 하지만 아무것도 찍히지 않는다. 최종연산이 없기때문에 중간연산만으로는 실행이 되지 않는것이다.

Stream<Integer> stream = Stream.*iterate*(1, n -> n+1)

.limit(6)

.peek(System.*out*::println)

.filter(number -> number % 2 == 0);

stream.collect(*toList*());

이런식으로 최종 연산 메서드가 실행될때 비로소 중간 연산들도 실행이 된다.

**1-3. Lazy & ShortCircuit**

스트림은 게으르(Lazy)다. 결론부터 말하자면 최종연산이 존재하지않으면 중간연산은 실행되지 않는다. 위에서부터 말했지만 스트림은 어디까지나 *연산을 위한 객체*로 그 자체로 자료구조의 역할을 하지 않는다. 때문에 최종연산이 존재하지않는 스트림은 그 의미가 없다고 볼 수 있다. 스트림이 게으르다는걸 확인할 수 있는 샘플코드는 어렵지않게 짤 수 있다.

Stream<Integer> stream = Stream.*of*(1, 2, 3, 4, 5);

Stream<Integer> s = stream.peek(System.*out*::println);

peek()메서드는 forEach()처럼 스트림의 요소를 순회하며 소비(Consumer<T>)하는 메서드이다. forEach()와 한가지 다른점은 중간 연산이라는점이다. 때문에 해당코드를 실행하면 우리가 기대하는 출력문은 출력되지 않는다.

Stream<Integer> stream = Stream.*of*(1, 2, 3, 4, 5);

Stream<Integer> s = stream.peek(System.*out*::println);

s.collect(*toList*());

최종연산이 호출될때 비로소 중간연산들도 실행되는걸 볼 수 있다.

Lazy외에도 스트림은 여러 최적화기법들을 도입했는데 그 중하나가 Short Circuit이다. 이건 뭔말일꼬.. 할 수 있는데 &&(And), ||(Or) 연산을 생각하면 이해가 쉽다.

&&연산은 좌항과 우항 모두 true일때 true을 반환하는 연산인데 좌항이 false면 우항은 쳐다도보지않는다.

Object obj = null;

boolean b = 1 == 2 && obj.toString().equals(123);

엉성한 코드지만..;; obj가 null이기때문에 거기다가 toString()를 호출하면 NullPointerException이 발생해야한다. 하지만 이미 좌항이 false이기때문에 우항은 실행도 하지않게되고, 그리하여 아무런 예외없이 코드는 실행된다.

이런기법이 Short Circuit이다.

List<String> list = Arrays.*asList*("a", "b", "c");

boolean b = list.stream().allMatch(str -> {

System.*out*.println(str);

return str.equals("d");

});

스트림 요소를 순회하면서 모든 요소에 Predicate이 true인지를 확인하는 구문이다. 출력된 내용을 보면 a만 찍히는걸 볼 수 있다. 전부다 d여야 true를 반환하는데 처음부터 d가 아니니 연산을 끊고 false를 반환하는 것이다.

**1-4. 기본형 스트림**

숫자타입 리스트를 선언할때는 제네릭을 이런식으로 선언한다.

List<Integer> list1 = new ArrayList<>();

Auto Boxing, Auto Unboxing이 지원되면서 그냥 기본형 쓰듯 사용할 수 있지만 내부적으로까지 boxing 비용이 없어진건 아니다. 하지만 자바의 제네릭은 기본형을 지원하지않기때문에 어쩔수 없이 저렇게 사용할 수 밖에 없다. 스트림 역시 기본형으로 제한을 걸기위해서는 저런식으로 해야한다.

Stream<Integer> stream = Stream.*of*(1, 2, 3, 4, 5);

스트림은 이런 boxing 비용을 줄이기위해 기본형에 특화된 객체를 따로 제공하고있다.

IntStream intStream = IntStream.*of*(1, 2, 3, 4, 5);

DoubleStream doubleStream = DoubleStream.*of*(1.0, 2.0, 3.0, 4.0, 5.0);

LongStream longStream = LongStream.*of*(1L, 2L, 3L, 4L, 5L);

기본형 스트림을 사용하면 boxing비용을 줄일 수 있을 뿐더러 해당 타입에 알맞는 연산들을 메서드로 제공하고있다.

intStream.sum(); // Stream<Integer> 로도 가능하다. 다만 기본메서드로 제공되지는 않는다.

한가지 주의할 점은 스트림과 기본형 스트림은 관계가 없기때문에 단순하게 타입변환이 되지않는다. 그래서 다형성을 이용하기가 어렵다. 물론 그에 대한 API도 제공해주고있다.

//이외에도 mapToObj, mapToDouble 같은 메서드를 제공한다.

List<Integer> interger = intStream.boxed().collect(*toList*());

적은 범위의 값들을 다룬다면 그냥 Boxing 비용을 감안하고서라도 Stream을 사용하는게 맘편해보이지만 그런 비용을 무시할 수 없을만큼 큰 범위의 데이터라면 기본형 스트림을 고려해보는것이 좋을것같다.

**1-5. 대표적인 API들**

람다는 기존에 없던 문법들이 추가되어 신기한 맘으로 공부해야하지만 사실 스트림은 특별할건 없다. 가볍게본다면 새로운 API가 추가된정도로 볼 수도있다. 개인적으로 스트림을 공부하면서 느낀건 그 철학이나 탄생배경도 중요하지만 일단 실무에서 써먹으려면 API를 달달 외워야 한다는것이다. 대표적인 API들을 소개하면서 포스팅을 마치겠다.

1) 중간연산

-Stream<R> map(Function<A, R>)

-Stream<T> filter(Predicate<T>)

-Stream<T> peek(Consumer<T>)

2) 최종연산

-R collect(Collector)

-void forEach(Consumer<T>)

-Optional<T> reduce(BinaryOperator<T>)

-boolean allMatch(Predicate<T>)

-boolean anyMath(Predicate<T>)

# [Java8#05. Optional](http://multifrontgarden.tistory.com/131)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.10.31 09:32

자바8을 처음 접했을때 느꼈던 점은 자바8의 꽃은 스트림이라는 것이었다. 그러나 내 맘을 가장 매혹시키고, 언젠가 자바8을 쓸날이 오면 이것만큼은 꼭 잘 쓰고싶다고 생각했던것은 오늘 포스팅할 Optional이었다.

**1. null**

자바는 좀 더 쉽게 프로그래밍하자는 취지에서 개발됐다. OSMU(One Source Multi Use)를 위해 하나의 코드로 OS에 상관없이 돌아갈수 있게 개발됐고, C계열을 처음 공부할때 지옥이라 불리는 포인터를 모두 감췄다. 그러나 단 하나 감추지 못한 포인터가 있는데 그것이 null포인터다.

자바 프로그래밍에서 NullPointerException은 언제 터질지 모르는 에러의 근원이었고 이를 방지하기위해 대부분의 메서드 앞줄에선 중복적인 방어코드가 들어가는 문제가 있었다.

public void method01(Object obj){

if(obj == null){

return;

}

// 로직

}

public String method02(String str){

if(str == null){

// null을 그대로 리턴하는건 클라이언트에서 다시금 null체크를 해줘야하기떄문에 기본값을 리턴하는걸 권장한다.

return null;

return "";

}

// 로직

}

매우 익숙한 코드일것이다. 하지만 저런 null체크는 반복적이고 코드가 지저분해지는 문제가 있고, 더욱이 쉽게 까먹고 넣지 못할경우 어김없이 NPE를 만나게된다. 때문에 저런 반복적인 코드를 안빼먹고 넣어주는게 오히려 꼼꼼한 개발자라는 말을 듣게 만들었다. 지금 저 예제는 그나마 인자로 넘어오는 객체의 레퍼런스만 체크하고있지만 이런 코드는 어떨까?

public void method01(Object obj) {

// getter를 이용해 내부 필드를 깊게들어가서 값을 가져옴

Object obj1 = obj.getObject().getObject.getObject();

// 로직

}

이런코드는 obj의 레퍼런스만 검사한다고해서 NPE를 벗어날수는 없다. 이런 코드가 나오게 된다.

public void method01(Object obj) {

Object obj1 = null;

if(obj != null){

if(obj.getObject() != null){

if(obj.getObject().getObject() != null){

obj1 = obj.getObject().getObject().getObject();

}

}

}

// 로직

}

깊은 의심(Deep Doubt) 패턴

보기만해도 뭔가 아닌거같다. 리팩토링을 해야할것같다. 그나마 해본게 이거다.

public void method01(Object obj) {

Object obj1 = null;

if (obj != null && obj.getObject() != null && obj.getObject().getObject() != null) {

obj1 = obj.getObject().getObject().getObject();

}

// 로직

}

if문은 한줄로 줄었지만 보기엔 역시 좋지 않다. 헬퍼메서드를 사용하는게 최선인것 같다.

public void method01(Object obj) {

Object obj1 = null;

if (isValidObject(obj)) {

obj1 = obj.getObject().getObject().getObject();

}

// 로직

}

private boolean isValidObject(Object obj){

return obj != null && obj.getObject() != null && obj.getObject().getObject() != null;

}

**2. Optional**

이러한 반복적인 null 체크를 없애기위해 Java8에 Optional<T>이라는 클래스가 추가됐다. 기술적으로 크게 어려운점은 없다. 실제 레퍼런스를 한번 감싸는 래퍼 객체를 만들어 null 체크를 내부로 숨겼기에 외부코드에선 null 체크가 보이지않게 감춘것에 불과하다.

class Optional<T>{

private T t;

}

Stream과 마찬가지로 생성자가 아니라 Static Factory 메서드를 이용해 객체를 생성한다.

public static void main(String args[]) {

String str = "hello";

Optional<String> o1 = Optional.*of*(str); // str이 null이면 NPE 발생

Optional<String> o2 = Optional.*ofNullable*(str); // str이 null이면 빈 Optional 객체 반환

Optional<String> o3 = Optional.*empty*(); // 빈 Optional 객체 반환

}

of() 같은 경우는 인자가 null이면 NPE를 발생시키기때문에 Optional을 쓸때 크게 쓸일이 없는것 같고.. of()를 쓰는 경우는 null을 예방하기위한다기보다는 Optional이 제공하는 API를 이용하고자 할때 쓸듯 하다. 슬슬 실무에 Optional을 활용하고있는데 ofNullable()만 사용하고 있는듯...

한번 더 감싸고있는 형태이기때문에 API는 내부 객체에대한 연산이 주를 이룬다.

**2-1. Optional API**

**-boolean isPresent()**

내부객체가 null이 아닌지 확인한다. null이면 false를 반환한다.

**-void ifPresent(Consumer<T>)**

Consumer<T>는 함수형 인터페이스 포스팅에서 봤듯 void 추상메서드를 갖고있다. null이 아닐때만 실행된다.

**-Optional<T> filter(Predicate<T>)**

스트림은 여러 데이터를 들고있는 객체다보니 filter로 걸러지는 데이터들이 반환됐지만, Optional은 내부객체가 단일객체인만큼 해당 조건을 만족하는지만 확인하는 정도로 사용할 수 있을 것 같다.

**-Optional<U> map(Function<T, U>)**

스트림과 같다. 내부 객체를 변환하는 용도로 사용한다.

**-T get()**

내부 객체를 반환한다. 다만 내부 객체가 null이면 NPE가 발생한다. null이 아니라는 확실한 경우에만 사용을 권장한다.

**-T orElse(T)**

내부 객체를 반환한다. 내부 객체가 null이면 인자로 들어간 기본값을 반환한다.

**-T orElseGet(Supplier<T>)**

orElse()와 동일한데 orElse()가 기본값 레퍼런스를 인자로 받는다면 orElseGet()은 내부 객체가 null일때 기본값을 반환할 객체를 인자로 받는다.

**-T orElseThrow(Supplier<U>)**

내부 객체가 null이면 인자로 전달받은 예외를 발생시킨다.

**2-2. Anti Pattern**

Optional을 처음 접하면 익숙치 않은맘에 이런 코드를 작성하게 된다.

Optional<Integer> integer = Optional.*empty*();

if(integer.isPresent()){

// ...

}

// ...

기존의 == null 체크와 다를게 없는 코드다. 이런코드가 나온다면 Optional을 잘못쓰고있을 확률이 높다. 가장 주의하고, 절대 지양해야할 패턴이다.

**2-3. orElse() / orElseGet()**

둘다 내부객체가 null일때 뭘 줄건지를 지정하는 메서드이다. 둘이 차이가 뭘까? null 이면 Object 객체를 반환한다고 가정해보자.

Optional<Object> objectOptional = Optional.*empty*();

Object object1 = objectOptional.orElse(new Object());

Object object2 = objectOptional.orElseGet(() -> new Object());

Object object3 = objectOptional.orElseGet(Object::new);

3개가 모두 같은 표현이다. 2번째와 3번째는 람다를 메서드 레퍼런스로 바꾼것이다. 이건 지극이 개인적인 생각이라 잘못 이해했을수도있는데 내가 이해하기로 orElse()는 값을 지정할때 사용하고 orElseGet()은 레퍼런스를 지정할때 사용하는것이 좋은것같다. 즉, 좀 더 Lazy하게 사용할 수 있는게 orElseGet()이라는 것이다.

orElse(new Object()); 같은 구문은 실제 내부 객체가 null이든 아니든 일단 Object 의 객체는 생성하게된다. 객체를 생성해서 해당 레퍼런스를 전달하고 orElse() 내부에서 null인지 아닌지 판별 후 이미 생성된 Object 객체를 사용하거나 사용하지않는것을 결정한다는 것이다. 그에 반해 orElseGet()은 객체를 생성하는 행위를 하는 메서드를 전달한다. orElseGet() 내부에서는 내부객체가 null일때만 해당 메서드를 실행하기때문에 null이 아니면 객체를 생성되지 않는다. 차이가 이해되는가? 또는 단순히 빈객체만 생성해서 반환하는게 아니라 객체가 생성될때 실행되어야할 행위들이 있다면? 그럴때도 orElseGet()이 유용할 것이다. 간단히 setter를 실행한다고 생각해보자.

Object object2 = objectOptional.orElseGet(() -> {

Object obj = new Object();

obj.setItem();

obj.setName();

return obj;

});

orElse()로 이런것들을 하려면 미리 setter를 다 호출해서 값을 넣어두고 그 객체를 인자로 보내야한다. null일때만 사용할, 실제 사용할지말지 알지도못하는 객체를 위해 그런 행위들이 무조건적으로 수행되고 바깥 스코프에 변수가 생기는건 좋은 코드라고 보기 힘들것같다. 그에 반해 값들은 orElse()를 쓰는것이 좀 더 좋을것같다.

Optional<Integer> integerOptional = Optional.*empty*();

Integer number1 = integerOptional.orElse(5);

Integer number2 = integerOptional.orElseGet(() -> 5);

이 경우는 orElse()가 좀 더 바람직한 방법이라고 보인다.

**2-4. 활용**

이제 위에서 봤던 깊은 의심 패턴 예제를 Optional로 바꿔보자.

Optional<Object> objectOptional = Optional.*empty*();

Object obj = objectOptional.map(Object::getObject)

.map(Object::getObject)

.orElse(null); // orElse(new Object), orElseGet(Object::new)

Optional 내부에있는 Object에 getter를 연속적으로 호출한다. 메서드가 진행되면서 중간에 null이 발견되면 NPE없이 orElse()가 호출되게된다. 어차피 값이 1개뿐인 filter보다 Optional에서는 map이 거의 주로 사용될 듯하다.

**3. 기본형 Optional**

Stream은 Auto Boxing, Auto Unboxing비용을 줄이기위해, 그리고 좀 더 기본형에 특화된 API 제공을 위해 기본형 스트림을 제공한다. IntStream, DoubleStream이 대표적인데 Optional도 기본형 Optional을 제공한다.

OptionalInt, OptionalDouble, OptionalLong이 있으며 사용법은 크게 다르지 않다.

OptionalInt optionalInt1 = OptionalInt.*empty*();

OptionalDouble optionalDouble1 = OptionalDouble.*empty*();

OptionalLong optionalLong1 = OptionalLong.*empty*();

다만 스트림같은경우 자료구조를 다루는 API이기때문에 대량의 형변환 비용을 줄이는 이점이 있는 반면 Optional은 어차피 내부에 1개의 레퍼런스만을 갖고있기때문에 기본형 사용의 이점이 크게 반감된다. 거기다 기본형 Optional과 Optional은 관계도 없어 형변환이 되는것도 아니기때문에 혼용한다면 불편함만 생길수있어 기본형스트림에 비해서는 그다지 환영받지 못하는 것 같다.