# <http://multifrontgarden.tistory.com/133>

# [토비의봄#01. Double Dispatch](http://multifrontgarden.tistory.com/133)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.11.07 22:57

토비의 스프링으로 유명한 토비님의 방송을 보고 정리를 시작한다.

**1. Dispatch**

*1. (특히 특별한 목적을 위해) 보내다   2. (편지・소포・메시지를) 보내다   3. 신속히 해...*

네이버 검색결과이다. 자바는 객체지향 프로그래밍언어로서 객체들간의 메세지 전송을 기반으로 문제를 해결하게된다. 메세지 전송이라는 표현은 결국 메서드를 호출하는것인데 그것을 dispatch라고 부르는 것이다.

dispatch는 static dispatch와 dynamic dispatch가 있는데 static은 구현클래스를 이용해 컴파일타임에서부터 어떤 메서드가 호출될지 정해져있는것이고, dynamic은 인터페이스를 이용해 참조함으로서 호출되는 메서드가 동적으로 정해지는걸 말한다.

public class Test {

public static void main(String[] arg) {

Dispatch dispatch = new Dispatch();

System.*out*.println(dispatch.method());

}

}

class Dispatch{

public String method(){

return "hello dispatch";

}

}

***Static Dispatch***

자바에서 객체 생성은 런타임시에 호출된다. 즉 컴파일타임에 알수있는건 타입에 대한 정보이다. 타입자체가 Dispatch라는 구현클래스이기때문에 해당 메서드를 호출하면 어떤 메서드가 호출될지 정적으로 정해진다. 이에대한 정보는 컴파일이 종료된 후 바이트코드에도 드러나게된다.

public class Test {

public static void main(String[] arg) {

Dispatchable dispatch = new Dispatch();

System.*out*.println(dispatch.method());

}

}

class Dispatch implements Dispatchable {

public String method(){

return "hello dispatch";

}

}

interface Dispatchable{

String method();

}

***Dynamic Dispatch***

인터페이스를 타입으로 메서드를 호출한다. 컴파일러는 타입에 대한 정보를 알고있으므로 런타임시에 호출 객체를 확인해 해당 객체의 메서드를 호출한다. 런타임시에 호출 객체를 알 수 있으므로 바이트코드에도 어떤 객체의 메서드를 호출해야하는지 드러나지 않는다.

예제코드에서 method() 메서드는 인자가 없는 메서드이지만 자바는 묵시적으로 항상 호출 객체를 인자로 보내게된다. 호출 객체를 인자로 보내기때문에 this를 이용해 메서드 내부에서 호출객체를 참조할 수 있는 것이다. 또한 이것이 dynamic dispatch의 근거가 되게된다.

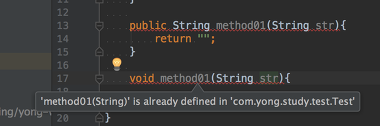
**2. Method Signature, Method Type**

면접 볼때 method signature를 적으라는 문제가 새록새록 기억나는 주제였다. 메서드 시그니처가 대충 뭔지는 알았지만 정확하게는 몰라서 당시에는 그냥 선언부를 전부 다 적었다.

*~~접근제어자 반환타입 메서드명 (인자)~~*

이번에 토비님께서 메서드 시그니처에 대해 정확히 정리해주셨다. 앞으로는 까먹지말아야지..

메서드 시그니처는 그것만으로 메서드를 구분지을수 있는 근거가 되어야한다. 그에 따라 자바에서 메서드 시그니처는 *메서드명 (인자)* 가 되게된다. 한가지 유의할 점은 반환 타입은 시그니처에 포함되지 않는다는 것이다. 쉽게 말하면 오버로딩이 되면 시그니처가 다르다고 보면 된다.



아래메서드는 접근제어자가 빠지고 반환타입이 달라졌는데 *이미 존재하는 메서드*라고 컴파일 에러가 발생하고있다.

public String method01(String str){

return "";

}

void method01(int num){

}

인자 타입이 달라지면 에러는 없어진다.

public String method01(String str){

return "";

}

void method01(String str1, String str2){

}

인자 타입이 같아도 개수가 달라지면 에러가 없어진다. 인자의 타입과 개수까지 시그니처에 포함된다는걸 알 수 있다.

그 다음으로는 Java8에서 추가된 Method Reference로 인해 등장한 용어인데 Method Type 이라는 용어가 생겼다. 메서드 타입에 포함되는것은 *반환타입 타입파라미터 인자 예외* 가 포함되게된다. 메서드 타입이 같으면 메서드 레퍼런스로 표현이 가능하게 되는것이다.

public class Test {

public static void main(String[] arg) {

Dispatch dispatch = new Dispatch();

dispatch.method("hello", System.*out*::println);

}

}

class Dispatch {

public void method(String str, Consumer<String> consumer){

consumer.accept(str);

}

}

method() 메서드는 Consumer 객체를 인자로 받는다. Consumer 구현체는 메서드 레퍼런스로 구현했는데, 저런식의 표현이 같으려면 메서드 타입이 일치해야한다는 것이다.

void accept(T t);

public void println(String x) {

synchronized (this) {

print(x);

newLine();

}

}

Consumer 인터페이스의 메서드 선언부와 println() 메서드의 선언부이다. 타입파라미터로 <String>을 지정해줬기때문에 accept() 메서드의 T는 String으로 치환해서 봤을때 위에 메서드타입이라고 말한것들이 일치하고있다. 메서드 타입이 동일할때 메서드 레퍼런스를 사용할 수 있는것이다.

**3. Double Dispatch**

1화 방송의 핵심 주제인 더블 디스패치이다. 윗 설명에서 디스패치가 무엇인지 알아보았다. 동적 디스패치(Dynamic Dispatch)는 메서드를 호출하는 인자가 구현체가 없는 인터페이스 타입인 경우 런타임시에 객체를 찾아 알맞는 메서드를 호출하는 것이다. 동적 디스패치를 이용하면 이런 코드가 가능하다.

public class Test {

public static void main(String[] arg) {

List<SmartPhone> phoneList = Arrays.*asList*(new Iphone(), new Gallaxy());

Game game = new Game();

phoneList.forEach(game::play);

}

}

interface SmartPhone{

}

class Iphone implements SmartPhone{

}

class Gallaxy implements SmartPhone{

}

class Game {

public void play(SmartPhone phone) {

System.*out*.println("game play [" +phone.getClass().getSimpleName()+ "]");

}

}

스마트폰 리스트를 순회하면서 각각 게임을 하고있다. 스마트폰 리스트 인터페이스를 타입파라미터로 전달했기때문에 동적디스패치로 인해 출력내용은 모두 다르다. 앞에서 말했듯이 이는 인터페이스로 참조하고있는 객체 레퍼런스를 동적으로 추적하기때문이다.

그럼 여기서 한가지 고민해보자. Game play가 스마트폰 구현체별로 다르게 구현되어야한다면 어떻게될까?

class Game {

public void play(SmartPhone phone) {

if(phone instanceof Iphone) {

System.*out*.println("iphone play [" + phone.getClass().getSimpleName() + "]");

}

if(phone instanceof Gallaxy) {

System.*out*.println("gallaxy play [" + phone.getClass().getSimpleName() + "]");

}

}

}

가장 쉽게 떠올릴수 있는 해결책이다. 하지만 그냥 한눈에 봐도 뭔가 아닌거같다. 만약 SmartPhone의 구현체로 Optimus가 추가된다면 Game 클래스까지 변경이 발생하게된다. 전혀 OOP적인 방법이 아니다.

class Game {

public void play(Iphone phone) {

System.*out*.println("iphone play [" + phone.getClass().getSimpleName() + "]");

}

public void play(Gallaxy phone) {

System.*out*.println("gallaxy play [" + phone.getClass().getSimpleName() + "]");

}

}

이런 방법은 어떨까?

public static void main(String[] arg) {

List<SmartPhone> phoneList = Arrays.*asList*(new Iphone(), new Gallaxy());

Game game = new Game();

phoneList.forEach(game::play); // 컴파일 에러 발생

}

직접 해보면 알겠지만 컴파일 에러가 발생한다. 자바는 하위 타입으로의 묵시적 형변환을 지원하지 않기때문에 명시적으로 형변환을 해야한다. 하지만 2개의 요소가 서로 타입이 다르므로 명시적 형변환을 하려면 반복문을 사용할 수 없다. 제네릭같은 방법을 이용한다고 하더라도 구현체마다 다른 행동을 해야하므로 하나의 메서드로 모으는건 의미가 없다.

자바에서 지원을 하고말고는 논외로하고 잠깐 생각해본다면, 어차피 런타임시에 어떤 객체가 들어오는지를 확인해서 서로 다른 메서드를 호출해주는 동적 디스패치가 존재한다면 이를 인자에도 적용할 수 있지 않을까? 그렇게되면 알맞은 SmartPhone 구현체를 확인해 알아서 각각 메서드를 호출시켜주면 참 고마울것이다.

하지만 당연하게도 자바에서는 그런걸 지원하지않는다. 이런 이유로 자바를 *싱글 디스패치(Single Dispatch) 언어*라고 한다.

자, 이제 if문을 없애자. 이런식으로 수정해주면 된다.

interface SmartPhone{

void game(Game game);

}

class Iphone implements SmartPhone{

@Override

public void game(Game game) {

System.*out*.println("iphone play [" + this.getClass().getSimpleName() + "]");

}

}

class Gallaxy implements SmartPhone{

@Override

public void game(Game game) {

System.*out*.println("gallaxy play [" + this.getClass().getSimpleName() + "]");

}

}

class Game {

public void play(SmartPhone phone) {

phone.game(this);

}

}

기존 Game클래스에 존재하던 비즈니스 로직을 각각 자기자신이 직접 처리하게끔 수정했다. 다형성을 한번 더 꼬아서 사용하는 것이다. 이때는 디스패치가 2번 일어나게되는데 play() 메서드를 찾기위한 정적 디스패치가 발생하고, game()메서드를 호출하는 객체를 찾기위한 동적 디스패치가 발생하게된다. 나는 지금 Game 클래스를 구현 클래스로 만들었기때문에 정적1번 동적1번이 발생하지만 Game 클래스역시 인터페이스를 기반으로 구현하여 인터페이스로 참조를 하게된다면 동적 디스패치가 2번 발생할것이다. 처음이 정적이든 동적이든 play() 메서드를 찾기위한 디스패치만 발생하던 기존 코드에서 play() 메서드 내부에 비즈니스로직을 호출하는 실제 객체를 찾기위한 *동적 디스패치가 1번 더 발생하면서 더블 디스패치(Double Dispatch)가 되는 것*이다.

또한 SmartPhone의 구현체들인 Iphone, Gallaxy가 비즈니스 로직을 직접 구현하기때문에 추후 Optimus 클래스 등 신규 구현체가 추가되더라도 Game 클래스에는 변경이 없다. *OCP를 만족*하게 바뀐 것이다.

언어의 한계적 특성으로 인해 인자에 동적 디스패치를 활용하지못하기때문에, 이를 위해 한번 더 다형성을 이용함으로서 호출객체 동적 디스패치를 이용하는 기법이다.

얼마전 실무에서 인터페이스로 인자를 받고 구현체별로 다른 행위를 했어야하는 일이 있었다. 그때 당시에 instanceof 연산사를 이용해서 해결했었는데, 다시 생각해봐도 더블 디스패치를 적용하기는 약간 무리가 있는 코드라 그걸 고치진 못할것같다. 하지만 구현체별로 다른 행위를 해야하는 상황이 또 온다면 그때는 더블 디스패치를 활용해보도록 노력해봐야겠다.

# [토비의봄#02. Super Type Token](http://multifrontgarden.tistory.com/135)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.11.12 13:46

**1. Generic**

java5에서 부터 추가된 제네릭은 타입을 파라미터로 만들어 넘어오는 파라미터에 따라 다른 타입이 되게끔한다. 이로서 얻을 수 있는 이득은 타입이 동적으로 변하게 되기때문에 개발자가 직접 타입체크를 하고 타입 캐스팅을 하는 코드가 없어지게되어 정적인 타입 안전성을 확보할 수 있다.

public class Test {

public static void main(String[] arg) {

MyOptional<String> stringOptional1 = MyOptional.*ofNullable*("hello");

MyOptional<String> stringOptional2 = MyOptional.*ofNullable*(123);

}

}

class MyOptional<T> {

T t;

private MyOptional(T t){

this.t = t;

}

public static <T> MyOptional<T> ofNullable(T t){

return new MyOptional<>(t);

}

}

MyOptional<T>라는 제네릭 클래스를 정의했다. 제네릭으로 타입을 제한하고있기때문에 stringOptional2 변수는 컴파일에러가 발생한다. 제네릭 포스팅은 이전에 한적이 있다(<http://multifrontgarden.tistory.com/104>).

제네릭은 클래스나 메서드를 유연하게 만들어주지만 깊게 사용하기가 어려운 문법이다. 가끔 이해할 수 없는 결과를 보일때도 있는데(가끔 타입 안전성이 깨질때가 존재), 그런 이유는 대부분 *타입 소거(Type Erasure)*때문이다. 타입 소거는 어느정도 알고는 있었는데 글로 정리하기가 힘들어 지난 포스팅에서는 아예 언급을 안했는데 이번에 토비님 방송을 보며 어느정도 정리가 되어 언급을 하려한다(물론 이번에도 타입 소거가 메인 주제는 아니다;;).

**2. TypeErasure**

다양한 자료들을 다루는 자료구조의 구현체는 Collection 구현체들은 자바의 가장 대표적인 제네릭 클래스들이다. 어떤 자료가 들어갈지 모르기때문에 제네릭으로 타입을 강제하고 사용하기 마련인데 제네릭으로 타입을 강제하지않아도 컴파일은 성공하고, 런타임시에도 (타입체크만 잘해준다면) 예외없이 잘 작동한다.

List<Integer> numbers1 = Arrays.*asList*(1, 2, 3, 4, 5);

List numbers2 = Arrays.*asList*(1, 2, 3, 4, 5);

타입을 제한하고 싶지 않은 경우가 있을 수도 있잖아? 라고 생각할 수도있겠지만 그런경우에도 <Object>와 같은 형태로 제네릭을 사용해야한다. 단지 제네릭을 사용하지않고도 제네릭 클래스를 사용할 수 있는 이유는 제네릭이 없던시절(Java5 미만)의 코드 하위호환을 위해서다.

Java5 이상의 컴파일러도 이전 코드를 문제없이 컴파일 하기위해 *자바는 컴파일시에 제네릭 정보들을 소거*시킨다. 이것이 타입 소거인것이다. 제네릭정보를 소거시키기때문에 바이트코드에 제네릭에 대한 정보는 아무것도 없고 런타임시에는 제네릭이 없는 셈이다.

class MyOptional<T> {

T t;

private MyOptional(T t){

this.t = t;

}

public static <T> MyOptional<T> ofNullable(T t){

return new MyOptional<>(t);

}

public T get(){

return this.t;

}

}

위 코드를

class MyOptional {

Object t;

private MyOptional(Object t){

this.t = t;

}

public static MyOptional ofNullable(Object t){

return new MyOptional(t);

}

public Object get(){

return (String)this.t;

}

}

컴파일러는 이런식으로 바꾼다. 제네릭은 사라진다.

public class Test {

public static void main(String[] arg) {

MyOptional<String> stringOptional = MyOptional.*ofNullable*("hello");

System.*out*.println(stringOptional.getTypeName());

}

}

class MyOptional<T> {

T t;

private MyOptional(T t){

this.t = t;

}

public static <T> MyOptional<T> ofNullable(T t){

return new MyOptional<>(t);

}

public T get(){

return this.t;

}

public Type getTypeName(){

try {

return this.getClass().getDeclaredField("t").getType();

} catch (NoSuchFieldException e) {

e.printStackTrace();

return null;

}

}

}

좀 더 확인해보고싶다면 이 코드를 실행해보자. 리플렉션을 이용해 t 필드의 타입을 확인하는 코드다. Object가 출력되는걸 확인 할 수 있다.

\* C#은 타입 소거가 아니라 타입 구체화(Type Reification)을 통해 제네릭을 구현하여 런타임시에도 제네릭이 소거되지 않는 방법을 사용했다. 이를 구현하기위해 이전코드들과의 하위호환을 포기했다.

**3. TypeSafetyMap**

이펙티브자바에서도 나왔던 예제인데 이번에 토비님 방송에서도 나왔다. 제네릭과 Map을 이용하면 TypeSafetyMap을 만들 수 있는데, Map은 Key와 Value로 이루어진 자료구조이다. 간단히 말하면 Key를 타입으로 지정하는 것이다.

public class Test {

public static void main(String[] arg) {

TypeSafetyMap map = new TypeSafetyMap();

map.put(Integer.class, 1);

map.put(Integer.class, "hello");

map.put(String.class, "hello");

map.put(String.class, 2);

}

}

class TypeSafetyMap {

Map<Class<?>, Object> map;

{

this.map = new HashMap<>();

}

public <T> void put(Class<T> clazz, T t){

map.put(clazz, t);

}

public <T> T get(Class<T> clazz){

return clazz.cast(map.get(clazz));

}

}

TypeSafetyMap은 타입 클래스를 key로 받고 그걸로 들어오는 데이터의 타입체크를 하게된다. 제네릭덕분에 컴파일시에 타입안전성을 확보할 수 있는데 이때 key에 들어오는 클래스 객체를 *타입 토큰(Type Token)*이라 부른다. 타입 안전성이 확보되기 때문에 put()메서드를 호출한 4줄중 2, 4번째 줄은 컴파일에러가 발생한다. 코드는 특별히 어려운 부분은 없다. Map을 한번 래핑해 제네릭 메서드를 이용해 타입안전성을 확보한다.

map.put(List.class, Arrays.*asList*(1, 2, 3, 4, 5));

map.put(List.class, Arrays.*asList*("a", "b", "c"));

애초에 제네릭 클래스인 리스트와 같은 클래스도 적용이 가능하다. 하지만 이 코드는 뭔가 이상한걸 알 수 있다. List<Integer>와 List<String>는 *불변(Invariant) 타입*으로 서로 완전히 다른타입의 리스트임에도 불구하고 해당 코드는 List<String>이 List<Ineger>를 덮어쓰게 될것이다. 심지어는 이런 코드도 나오게 된다.

map.put(List.class, Arrays.*asList*(1, "a"));

List에 제네릭으로 타입제한을 걸지 못해서 그런것같다. 가장 단순하게 드는 생각은 이것이다.

map.put(List<Integer>.class, Arrays.*asList*(1, 2, 3, 4, 5));

map.put(List<String>.class, Arrays.*asList*("a", "b", "c"));

코드를 짜보면 알겠지만 이 코드는 컴파일 에러가 발생한다. 타입 소거에서 말한대로 컴파일시에는 사라질 정보들이라서 저런식으로 타입안전성을 확보할 수가 없기때문이다.

**4. Super Type Token**

그럼 자바에서는 저런경우에는 타입안전성을 보장할 방법이 없는걸까? 결과부터 말하자면 제네릭은 컴파일시에 제거되지만 단 하나의 경우엔 소거되지않는다. 그리고 그 기법을 이용하여 위의 경우 타입안전성을 확보할 수 있다.

public class Test {

public static void main(String[] arg) {

Sub sub = new Sub();

System.*out*.println(((ParameterizedType)sub.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

}

}

class TypeReference<T>{

}

class Sub extends TypeReference<String>{

}

바로 이런 경우이다. 제네릭 클래스를 정의한 후 그 제네릭 클래스를 상속하여 사용하는 경우 그때 제네릭을 지정하면 그 제네릭정보는 소거되지 않고 런타임시에도 정보가 남게된다.

코드에 대해 조금 설명하자면

1. 인스턴스의 구현 클래스를 찾고

2. 그 클래스의 상위 클래스인 제네릭 클래스를 찾아서 ParameterizedType으로 캐스팅을 한다.

3. 명시적 캐스팅이 발생한다는건 다운캐스팅이라는 것이다. getGenericSuperclass()가 반환하는 타입은 ParameterizedType의 상위 타입일것이다. 슈퍼 클래스가 제네릭 클래스가 아니라면 캐스팅시 예외가 발생한다.

4. 제네릭 타입 파라미터에서 실제 인자들을 찾아 그중의 첫번째 파라미터를 가져온다.

5. 제네릭은 Map<K, V>와 같이 2개 이상도 들어갈수있으니 필요에따라 숫자가 꼭 0이어야 하는건 아니다.

6. 그 첫번째 파라미터의 타입이름을 가져온다.

뭐 대충 이런 구조다.

아직 감은 오지않겠지만 일단 실마리는 찾았다. 이걸 좀 더 응용하면 애초에 이걸 고민하게했던 타입안전성 맵에도 활용할 수 있게된다. 일단 이걸 여러 타입에 사용하려면 그때마다 클래스를 만들어야한다. 그럼 제네릭에 타입이 추가될때마다 새 클래스를 정의해야할까? 참 미련한 짓이다. 익명클래스를 활용하자.

public class Test {

public static void main(String[] arg) {

TypeReference sub1 = new TypeReference<String>() {};

TypeReference sub2 = new TypeReference<Integer>() {};

System.*out*.println(((ParameterizedType)sub1.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

System.*out*.println(((ParameterizedType)sub2.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

}

}

class TypeReference<T>{

}

익명 클래스를 활용하면 각종 타입에 대응하기가 좀 더 수월해진다. 한가지 주의할점은 제네릭 정보가 남는건 하위 타입에서 상위 타입의 타입 파라미터를 전달했을때이므로 반드시 하위 타입이어야한다는것이다. 그리고 하위타입임을 알리기위해 구현부 {}를 붙여야한다. 이걸 빼도 컴파일시 에러는 발생하지 않는다. TypeReference의 상위 클래스는 Object 이므로 타입파라미터가 존재하지않아 캐스팅시 런타임 예외만 발생하게된다.

public class Test {

public static void main(String[] arg) {

TypeReference intType = new TypeReference<Integer>();

TypeReference stringType = new TypeReference<String>();

System.*out*.println(((ParameterizedType)sub1.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

System.*out*.println(((ParameterizedType)sub2.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

}

}

얘는 하위 타입이 아니라서 타입소거가 이루어진다. 뿐더러 런타임 예외까지 발생.

가능한 TypeReference는 객체화가 이루어지지않는게 컴파일시 안전성을 높여줄것같다. 추상클래스로 정의하자.

public class Test {

public static void main(String[] arg) {

TypeReference intType = new TypeReference<Integer>(); // 컴파일 에러 발생

TypeReference stringType = new TypeReference<String>();

System.*out*.println(((ParameterizedType)sub1.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

System.*out*.println(((ParameterizedType)sub2.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

}

}

abstract class TypeReference<T>{

}

슈퍼 타입에 대한 제네릭 정보는 얼마나 남게되는걸까? 리스트같은 클래스를 이용해 중첩적으로 활용해도 정보가 남는걸까?

public static void main(String[] arg) {

TypeReference intType = new TypeReference<List<Integer>>(){};

TypeReference stringType = new TypeReference<Set<Map<String, List<Integer>>>>(){};

System.*out*.println(((ParameterizedType)sub1.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

System.*out*.println(((ParameterizedType)sub2.getClass().getGenericSuperclass())

.getActualTypeArguments()[0]

.getTypeName());

}

오 대박! 왜 대박인지 알고싶으면 출력해보자.

class TypeSafetyMap {

Map<TypeReference<?>, Object> map;

{

this.map = new HashMap<>();

}

public <T> void put(TypeReference<T> typeReference, T t){

map.put(typeReference, t);

}

public <T> T get(TypeReference<T> typeReference){

Type type = ((ParameterizedType)typeReference.getClass().getGenericSuperclass())

.getActualTypeArguments()[0];

if(typeReference.type instanceof Class<?>) {

return ((Class<T>) type).cast(map.get(typeReference));

}

return ((Class<T>)((ParameterizedType)type).getRawType())

.cast(map.get(typeReference));

}

}

이제 제일 처음 만들었던 TypeSafetyMap을 수정하자. key를 TypeReference로 하자. 다른부분은 크게 안바꼈는데 get()이 좀 복잡해졌다. get()에 있는 if문은 중첩제네릭클래스인지 아닌지를 확인하는 것이다. <List> 일경우엔 파라미터타입 자체가 List.class 라서 캐스팅을 하면 되지만 중첩 제네릭, 즉 <List<String>> 같은 타입일경우 List.class로 타입캐스팅이 되지않는다. 때문에 raw type을 구해서 raw type으로 캐스팅하는 코드가 추가되었다. 이외에 상위 클래스가 제네릭 클래스가 아닐경우를 체크하는부분은 굳이 넣지 않았다.

public static void main(String[] arg) {

TypeSafetyMap map = new TypeSafetyMap();

map.put(new TypeReference<Integer>() {}, 1);

System.*out*.println(map.get(new TypeReference<Integer>() {}));

}

그리고나서 테스트를 해보자. null이 출력되는걸 볼 수 있다. 왜 null일까? 잘 생각해보면 결국 TypeSafetyMap이 래핑하고있는 내부 Map의 key는 TypeReference다. *Map은 key 중복을 체크할때 key로 들어온 객체의 hashCode() 와 equals() 메서드를 호출하게 된다.* 현재 put 과 get을 할때 사용할때는 둘다 익명 객체 생성으로 *서로 다른 인스턴스*이기때문에 hashCode()와 equals()는 당연히 다르다고 판별되어 같은 키가 아니라고 판단할 것이다. 일단 빠르게 잘되는지부터 확인하고싶다면 이렇게 하면 된다.

public static void main(String[] arg) {

TypeSafetyMap map = new TypeSafetyMap();

TypeReference tr = new TypeReference<Integer>() {};

map.put(tr, 1);

System.*out*.println(map.get(tr));

}

인스턴스를 변수에 담아 동일한 인스턴스로 put() 했다가 get()하니 1이 출력되는걸 볼 수 있다. 하지만 이건 테스트를 위한거고 궁극적으로는 hashCode()와 equals()를 오버라이딩 해줘야한다.

abstract class TypeReference<T>{

Type type;

protected TypeReference(){

type = ((ParameterizedType)this.getClass().getGenericSuperclass())

.getActualTypeArguments()[0];

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass().getSuperclass() !=

o.getClass().getSuperclass()) return false;

TypeReference<?> that = (TypeReference<?>) o;

return type != null ? type.equals(that.type) : that.type == null;

}

@Override

public int hashCode() {

return type != null ? type.hashCode() : 0;

}

}

인스턴스가 같은지를 비교하기위한 용도로 Type 필드를 추가하고 IDE가 지원하는 오버라이딩을 이용했다. equals() 같은 경우는 조금 수정을 해야한다.

public static void main(String[] arg) {

TypeSafetyMap map = new TypeSafetyMap();

map.put(new TypeReference<Integer>() {}, 1);

System.*out*.println(map.get(new TypeReference<Integer>() {}));

}

그리고 다시 인스턴스 생성방식으로 호출하면 정상출력을 확인할 수 있다.

자, 이제 원래 이런 짓(?)을 시작하게만들었던 중첩 제네릭도 확인해보자.

public static void main(String[] arg) {

TypeSafetyMap map = new TypeSafetyMap();

map.put(new TypeReference<Integer>() {}, 12);

map.put(new TypeReference<List<String>>() {}, Arrays.*asList*("a", "b", "c"));

map.put(new TypeReference<List<Integer>>() {}, Arrays.*asList*(1, 2, 3));

System.*out*.println(map.get(new TypeReference<Integer>() {}));

System.*out*.println(map.get(new TypeReference<List<Integer>>() {}));

System.*out*.println(map.get(new TypeReference<List<String>>() {}));

}

아.. 성공적으로 돌아간다. 포스팅에 올라오는 코드는 돌아가는걸 확인하고 올리고있지만 방송에 토비님 코드를 그대로 따라치는게 아니라 방송한번 보고 따로 타이핑을 하는거라 중간중간 생각치못한 일이 많았기에 성공적으로 돌아가는게 나왔을땐 정말 눈물이 날뻔했다.

equals() 구현하는것도 몇분을 헤맸는지 ㅜㅜ

#추가방송

**5. 리팩토링**

private Map<TypeReference<?>, Object> map;

맵의 키를 TypeReference로 사용하고 있다. 그리고 키같은경우는 일시적으로 사용하는 객체이므로 익명클래스를 이용해 사용하고있다. 꼭 익명 클래스를 사용할 필요는 없지만 맵의 키로 쓰겠다고 타입마다 일일이 클래스파일을 만드는건 그다지 효율적으로 보이지 않는다.

익명 클래스를 이용할때는 메모리 측면의 이슈가 하나있는데 일단 익명클래스의 본질을 약간 생각해볼 필요가 있다.

익명클래스는 사용 코드에서 즉시적으로 클래스를 정의하고 객체를 생성하는 *내부 클래스(Inner Class)*이다. 외부 클래스의 인스턴스가 생성되고 그 인스턴스내에서 생성되는 내부 객체이기때문에 내부객체가 맵의 key로 사용되게되면 외부 객체에 대한 연결끈을 놓지않아 key로 사용된 TypeReference 객체뿐만 아니라 그 외부 객체까지도 GC대상에서 벗어나게되어 메모리에 부하를 주게된다. 그런데 사실 우리가 진짜 필요한건 TypeReference 객체가 아니라 그 객체가 갖고있는 슈퍼클래스의 제네릭 타입 정보이다. key를 TypeReference로 주기때문에 고생해가며 equals(), hashCode()까지 재정의 했어야하는데 key를 type으로 주게되면 그런것들이 필요없어지게된다.

private Map<Type, Object> map;

진짜 필요한건 Type이므로 key자체를 type으로 잡자.

public abstract class TypeReference<T> {

private Type type;

public TypeReference(){

this.type = ((ParameterizedType)getClass().getGenericSuperclass())

.getActualTypeArguments()[0];

}

public Type type(){

return this.type;

}

}

TypeReference에 hashCode()와 equals()를 재정의 했던 이유는 map의 key로 사용될때 같은 타입일 경우 같은 객체로 보기위함이었다(동등성). 그런데 key를 Type으로 쓸거니 얘네가 같은지를 확인할 필요는 없어졌으므로 해당 코드는 삭제한다. 그리고 type을 꺼내오기위한 메서드를 추가했다.

public class TypeSafetyMap {

private Map<Type, Object> map;

{

map = new HashMap<>();

}

public <T> void put(TypeReference<T> typeReference, T t){

this.map.put(typeReference.type(), t);

}

@SuppressWarnings("unchecked")

public <T> T get(TypeReference<T> typeReference){

Type type = ((ParameterizedType)typeReference.getClass().getGenericSuperclass())

.getActualTypeArguments()[0];

Class<T> clazz;

if(type instanceof ParameterizedType){

clazz = (Class<T>)((ParameterizedType)type).getRawType();

}else{

clazz = (Class<T>) type;

}

return clazz.cast(map.get(typeReference.type()));

}

}

크게 달라진점은 없다. 단지 map에 put()하고 get()할때 Type을 사용하는 것 뿐이다. 작동은 동일하게 하는걸 확인할 수 있다.

**6. ResolvableType in Spring**

어렵게어렵게 리플렉션까지 써가며 슈퍼타입토큰을 구현했는데 스프링에선 좀 더 좋은 API를 제공하고있다.

public class Test {

public static void main(String[] arg) {

ResolvableType type = ResolvableType.*forInstance*(new TypeReference<List<String>>() {

});

System.*out*.println(type.getSuperType().getGeneric(0).getType().getTypeName());

}

}

리플렉션으로 작성하게되면 이런저런 타입 캐스팅도 필요하게되고, Checked Exception도 신경써줘야하는데, 스프링의 ResolvableType을 사용하게되면 좀 더 편리하게 사용할 수 있다.

# [DesignPattern#01 Strategy Pattern](http://multifrontgarden.tistory.com/98)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.03.01 13:52

디자인패턴의 꽃이라 불리는 Strategy Pattern 에 대해 알아보자. 기본적으로 자바를 배울때 자바는 객체지향언어이고 그로인한 장점을 상속을 통해 이미 작성되어있는 코드를 재사용할 수 있는 장점이 있다고 배운다. Strategy Pattern은 그런 상속을 통한 소스 재사용을 상속이 아닌 합성(composite)을 통해 사용하는것에 초점을 맞춘다. 굳이 이렇게 거창하게 디자인패턴이라는 단어가 아니라도 다중상속이 지원되지않는 자바에서 상속 대신 합성을 사용하여 소스를 재사용하고자 하는 경우는 많다. 대표적으로 is a, has a 로 구분해서 상속과 합성을 알맞게 사용하자는 말이 있는데 합성이 구체적으로 뭔지부터 알아보자.

//getString() 을 재사용 하고싶다.

class A{

  public String getString(){

    return "Hello World";

  }

}

//상속을 이용한 방법

class B extends A{

}

//합성을 이용한 방법

class C{

  public String getString(){

    A a = new A();

    return a.getString();

  }

}

//테스트

public class Test{

  public static void main(String[] arg){

    B b = new B();

    C c = new C();

    System.out.println(b.getString());  //Hello World 출력!

    System.out.println(c.getString());  //Hello World 출력!

  }

}

두 경우 모두 A클래스 소스를 재사용함으로서 Hello World를 출력하게된다. 합성을 이용했을경우 변수도 선언해야하고 소스도 좀 더 길어지지만 단일상속만을 지원하는 자바에서 상속이라는 유일무이한 자원을 아껴두고있는 이점이 있다. 또한 상속같은 경우 상위 클래스와 하위 클래스간의 결합도가 높아져 재사용하고자하는 상위 클래스를 수정할 경우 하위클래스까지 영향을 미치게된다. Strategy Pattern은 합성과 다형성까지 이용해 언제든 재사용하고자 하는 객체를 교체해가며 필요한 전략(strategy)을 호출하는 패턴이다.

interface A{

  public String getString();

}

//인터페이스 A 구현

class B implements A{

  @Override

  public String getString(){

    return "Hello World";

  }

}

//인터페이스 A 구현

class C implements A{

  @Override

  public String getString(){

    return "Hello LichKing":

  }

}

//클라이언트 클래스

class D{

  A a;

  public void setA(A a){

    this.a = a;

  }

  public String playStrategy(){

    return a.getString();

  }

}

//테스트

public class Test{

  public static void main(String[] arg){

    D d = new D(); //클라이언트 객체 생성

    d.setA(new B()); //전략 전달

    System.out.println(d.playStrategy()); //전략 실행

    d.setA(new C()); //전략 교체

    System.out.println(d.playStrategy()); //전략 실행

  }

}

setter 메서드를 통해 필요할때마다 전략을 교체해가며 달라진 객체의 메서드를 호출하게된다. 전략을 사용하는 클라이언트 클래스인 D는 자신이 직접 전략객체를 생성해서 호출하는 것이 아니라 외부에서 주입받은 객체를 사용하기때문에 소스가 유연해지고 언제든 다른 객체를 주입받아(해당 인터페이스만 구현해준다면) 호출할수있기때문에 확장성도 뛰어나진다. 주입이라는 단어를 보니 떠오르는게 없는가? 스프링의 DI 원리가 바로 이 Strategy Pattern에 기반한 것이다. 이 패턴이 디자인패턴의 꽃이라 불리는 이유는 그만큼 객체지향설계원칙을 준수하고 널리 사용되기떄문일 것이다. 잘 익혀두고 사용하자.

# [DesignPattern#02 Template Method Pattern](http://multifrontgarden.tistory.com/100)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.03.02 15:40

자바에서 기 작성된 소스를 재사용하는 방법은 사용하고자 하는 클래스에게 상속하거나 혹은 인스턴스멤버로 합성(composite)하는 방법이 있다. 지난번 포스팅인 Strategy Pattern은 두 가지 방법중 합성을 선택해 외부에서 전략을 주입하며 사용하는 법에 대해 알아봤다.

이번에는 상속을 사용하는 패턴에 대해 알아보자. 바로 Template Method Pattern(이하 템플릿 메서드 패턴) 인데 이름에서 느껴지듯 기 작성된 소스(Template)가 존재하고 해당 소스에서 부분적으로 메서드를 오버라이딩하는 패턴이다. 이렇게 작성할 경우 전체적인 뼈대인 기존 소스는 동일하게 작동하며 오버라이딩한 부분만 구현클래스별로 작동할것이다. 코드로 살펴보자

abstract class A{ //추상클래스로 작성한다.

public int templateMethod(int num1, int num2){  //재사용하고자하는 template 메서드

System.out.println("입력받은 값 => [" + num1 + ", " + num2 + "]");

int result = calculation(num1, num2);

return result;

}

public abstract int calculation(int num1, int num2);    //오버라이딩에 의해 구현체가 바뀔수있는 추상메서드

}

class Sum extends A{

@Override

public int calculation(int num1, int num2){

return num1 + num2;    //더하기 연산을 하도록 오버라이딩

}

}

class Multi extends A{

@Override

public int calculation(int num1, int num2){

return num1 \* num2;    //곱하기 연산을 하도록 오버라이딩

}

}

public class Test{

public static void main(String[] arg){

A a = new Sum();

System.out.println(a.templateMethod(5, 10));    //더하기 수행

a = new Multi();

System.out.println(a.templateMethod(5, 10));    //곱하기 수행

}

}

갑자기 예제를 짜려니까 너무 생각이 안나서 너무 간단하게 아닌가 싶기도하지만... 이런식으로 진행되게 된다. 굳이 추상메서드가 아니라도 오버라이딩이 가능하기때문에 templateMethod도 오버라이딩이 될수도있다는점을 명심하자. 오버라이딩을 막고 싶다면 templateMethod를 final로 선언하면 된다.

그리고 꼭 오버라이딩을 해야하는 추상메서드가 아니라 선택적으로 오버라이딩을 해야하는 hook메서드를 집어넣기도 한다.

abstract class A{ //추상클래스로 작성한다.

public int templateMethod(int num1, int num2){  //재사용하고자하는 template 메서드

System.out.println("입력받은 값 => [" + num1 + ", " + num2 + "]");

int result = calculation(num1, num2);

hook();

return result;

}

public abstract int calculation(int num1, int num2);    //오버라이딩에 의해 구현체가 바뀔수있는 추상메서드

public void hook(){}    //아무것도 하지않는 '구현' 메서드

}

class Multi extends A{

@Override

public int calculation(int num1, int num2){

return num1 \* num2;    //곱하기 연산을 하도록 오버라이딩

}

@Override

public void hook(){

System.out.println("연산 종료");    //구현메서드이기때문에 필요에따라 오버라이딩 하면된다.

}

}

public class Test{

public static void main(String[] arg){

A a = new Multi();

System.out.println(a.templateMethod(5, 10));    //곱하기 수행

}

}

이 패턴은 상속을 이용하기때문에 이 패턴을 구현하기위해서는 다른 클래스는 상속받을수가 없다는 단점이 있다. effective java에서도 그렇고 하도여기저기서 상속은 가급적 쓰지말라고하다보니 상속은 일단 제끼고 생각하게되는 경향이 있는데 필요할땐 써야하지않겠는가? 디자인패턴을 공부하는것도 힘들지만 언제 어떤 패턴을 써야하는지 판단하는것도 참 힘든것 같다

# [DesignPattern#03 Template Callback Pattern](http://multifrontgarden.tistory.com/101)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.03.04 22:46

앞선 두 포스팅을 통해 전략패턴(Strategy Pattern)과 템플릿 메서드 패턴(Template Method Pattern)에 대해 알아봤다. 이번에 알아볼 패턴은 템플릿 콜백 패턴(Template Callback Pattern)인데 앞선 두 패턴을 적절히 혼합한 형태라 크게 어려운 부분은 없다.

전략 패턴의 경우 재사용하고자하는 클래스를 상속이 아닌 인스턴스 변수로 취급하는 합성(Composite)을 이용한다. 그리고 인터페이스를 활용한 다형성과 setter 메서드를 이용해 사용하고자하는 전략(인스턴스 변수가 참조하는 객체)을 필요할때마다 변경해가며 호출함으로서 변화에는 닫히고 확장에는 열린 소스를 짤수있게하는 패턴이다.

템플릿 메서드 패턴의 경우는 템플릿이라는 단어에서부터 알수있듯이 크게 돌아가는 메서드 내에서 부분적으로 오버라이딩을 활용해 메서드의 동작을 제어한다.

전략 패턴이 클래스 레벨의 확장을 추구한다면 템플릿 메서드 패턴은 메서드 레벨의 확장을 추구하는 것이다. 그렇다면 이 둘을 적절히 섞는다면 어떻게 하면 될까? 크게 돌아가는 메서드 내에서 부분적으로 확장이 필요한 메서드를 오버라이딩이 아니라 메서드(전략)를 주입받아 해당 메서드를 실행하면 된다. 말로만 설명하면 잘 이해가 안갈수있는데 사실 자바스크립트같은 언어에서는 아무렇지않게, 매우 당연하다는 듯이 받아들이고 사용하고있는개념이다. 잠시 jQuery 사용 소스를 보자.

jQuery(document).ready(function(){

});

다년의 경험이 쌓인 개발자가 아니라도 jQuery를 사용하는 프로젝트를 한번이라도 진행해봤다면 저 코드는 눈감고도 칠수있을정도로 친숙한 코드일것이다. 다만 저 코드를 분석해가며 쓰는경우는 흔치않을텐데 공백제외하고 1줄이라고도 할수있는 저 소스를 분석해보자면

jQuery() 함수다. ()가 붙은걸보니 함수를 호출하는 부분이다. ()안에 document가 들어가있다. jQuery() 함수를 호출하는데 인자로 document 객체를 보낸다. 그 뒤에 .ready() 가 있다. jQuery(document) 가 뭘반환하는지는 잘 모르겠지만 저 소스가 에러없이 실행된다는건 ready() 속성을 갖고있는 객체가 반환되는것같다. ready() 도 ()가 붙는걸보니 함수다. 더욱이 ()가 붙는다는건 함수를 호출한다는 뜻이다. ()안에 인자를 보니 function이다. 인자로 익명함수를 전달하고있는것이다. 그럼 이 인자는 누가 실행하나?

말로한 설명이 어렵다면 이렇게 풀어서 볼수있다.

function callback(){

}

var obj = jQuery(document);

obj.ready(callback);

jQuery() 함수가 반환한 객체를 이용해 ready() 함수를 실행시키고, 해당 함수에 callback 함수를 전달한다. 그런데 가만히 보니 callback()이 아니라 callback이다. 함수를 내가 실행하는게 아니라 함수 그 자체를 인자로 보낸다는것이다. 함수 그 자체를 인자로 전달받은 ready() 함수가 자기가 해야할일들을 한 다음 인자로 받은 callback함수를 실행시키는 것이다.

자바에서는 콜백이라는 개념이 좀 생소할수도있어서 자바스크립트 소스를 갖고 설명했는데 결국 콜백이란 어렵게 생각할 필요없이 함수 그 자체를 전달하는것이다.

결국 외부에서 함수를 주입한다는 뜻인데 이는 곧 위에서 말한 템플릿 콜백 패턴의 설명과 일치한다. 그런데 자바에서는 메서드를 전달할 수가 없는데?

자바는 언어 자체가 메서드의 전달을 허용하지않는다. 좀 어렵게 말하면 함수형 프로그래밍이니 일급객체니 하는 단어들이 나오는데 그런건 다른데서 배우고 메서드를 전달하는것 자체만 알아보자.

언어 자체가 메서드만 왔다갔다 하는걸 지원하지않기때문에 자바에서 콜백을 구현하기 위해서는 어쩔수없이 객체를 생성해야한다. 그런데 고작 콜백으로 호출할 메서드하나 보내자고 클래스파일을 만들면 너무 난잡해진다. 그리고 그렇게할거면 그냥 템플릿 메서드 패턴쓰지 굳이 공부하기힘들게 디자인패턴 하나 더 만들이유도없다. 그래서 이부분을 '익명 내부 클래스'로 해결한다. 예제로 보자.

interface Calculator{

public int calculation(int num1, int num2);

}

class A{

public int templateMethod(int num1, int num2, Calculator c){

System.out.println("입력받은 값 => [" + num1 + ", " + num2 + "]");

int result = c.calculation(num1, num2);

return result;

}

}

public class Test{

public static void main(String[] arg){

A a = new A();

System.out.println(a.templateMethod(5, 10, new Calculator(){

@Override

public int calculation(int num1, int num2){

return num1 + num2;

}

}));    //더하기 수행

System.out.println(a.templateMethod(5, 10, new Calculator(){

@Override

public int calculation(int num1, int num2){

return num1 \* num2;

}

}));    //곱하기 수행

}

}

지난번 템플릿 메서드 패턴의 예제를 템플릿 콜백 패턴 형태로 바꿨다. 원래 사용될 템플릿 메서드를 갖고있는 클래스를 추상클래스로 선언한 후 해당 클래스를 상속받는 서브클래스에서 확장이 필요한 부분을 오버라이딩해서 사용하는 형태였는데, 확장해야하는 메서드를 전략 패턴처럼 외부에서 주입받아 호출하는 형태로 바꿨다. 그리고 누누히 말하듯 메서드를 인자로 쓸수없기때문에 익명내부클래스를 이용해 객체를 전달하고있다. 저부분을 완전한 전략패턴으로 구현한다거나 템플릿 메서드패턴으로 구현했다면 추가적인 요구사항(나누기, 빼기 등등)이 들어올때마다 메서드하나 때문에 구현클래스를 늘려나가야하는 상황이 왔을수도 있다. 또한 기존 템플릿 메서드 패턴의 단점이었던 상속을 이용했기때문에 슈퍼클래스와 서브클래스간의 결합도가 너무나도 단단해지는 점 마저 타파가 가능해졌다.

익명클래스를 잘 사용하지않아본 사람들은 저 소스가 좀 생소할수도있겠지만 패턴자체가 이해하기힘들다거나 하는부분은 없을것이다(전략패턴을 이미 잘 이해하고있다는 전제가 있어야한다.). 앞으로도 디자인패턴 포스팅을 몇개 더 하겠지만 이름은 분명히 쌩판 다른데 구현소스를 보니 별 차이가없어보인다고 생각될수도있는 패턴들이 생각보다 많다. 디자인패턴은 소스 그자체로 패턴들을 구분하기보다는 애초에 어떤 목적으로 이런 패턴이 나왔는지를 고민해야 구분하기가 쉽다. 서로 다른 문제를 해결하기위해 패턴이 나왔는데 구현해놓고보니 원래있던것과 크게 다르지않을수있다는것이다. 그래서 소스보다는 목적을 잘 파악하고 이해하는게 중요하다. 그리고 그 이해를 바탕으로 현재 직면한 문제에 적절한 패턴을 고르는것이 사실 디자인패턴 그 자체를 공부하는것보다 어렵다.

다시 돌아와서 위 소스를 잠깐만 더 보자면, 메서드하나 보내겠다고 .java파일이 늘어가는것 자체는 막았지만 생각하기에 따라 메서드하나 보내겠다고 저런식으로 작성하는것도 난잡하게 보일수도있다. 이 문제는 java 1.8에서부터 지원하는 람다를 이용해 해결할수있다. 람다예제를 끝으로 오늘 포스팅을 마치겠다.

@FunctionalInterface    //함수형인터페이스라는걸 보장해주는 애노테이션.

interface Calculator{

//추상메서드가 1개인 인터페이스를 함수형인터페이스라한다.

public int calculation(int num1, int num2);

}

class A{

public int templateMethod(int num1, int num2, Calculator c){

System.out.println("입력받은 값 => [" + num1 + ", " + num2 + "]");

int result = c.calculation(num1, num2);

return result;

}

}

public class Test{

public static void main(String[] arg){

A a = new A();

//람다표현식을 이용한 더하기

System.out.println(a.templateMethod(5, 10, (num1, num2) -> num1 + num2));

//람다표현식을 이용한 곱하기

System.out.println(a.templateMethod(5, 10, (num1, num2) -> num1 \* num2));

}

}

# [DesignPattern#04 Decorator Pattern](http://multifrontgarden.tistory.com/102)

[Java/Servlet/JSP](http://multifrontgarden.tistory.com/category/Java/Servlet/JSP) 2016.03.10 15:04

이번에 알아볼 패턴은 Decorator Pattern(데코레이터 패턴)이다. 구현법이 데코레이터 패턴과 비슷한 패턴들이 몇 존재하므로 구현법으로 구분하려하지말고 목적이 뭔지 잘 파악하는게 중요할듯 하다.

데코레이터 패턴은 전략 패턴(Strategy Pattern)과 마찬가지로 합성(Composite)을 이용하는 패턴이다. 내부적으로 실제 비즈니스로직을 처리하는 객체를 인스턴스변수로 갖고있으면서 해당 비즈니스로직의 처리 결과값을 자신의 이름이 말하든 한번 더 가공해주는 패턴이다.

소스로 확인해보자.

//놀이공원 인터페이스 처음엔 A, B였는데 예제도 진화하고있다...

interface AmusementPark {

public int getTicketCost();    //티켓 값을 반환하는 추상메서드

}

//놀이공원 인터페이스를 구현한 디즈니랜드 클래스

class DisneyLand implements AmusementPark {

@Override

public int getTicketCost(){

return 10000;    //티켓 값은 10000원

}

}

영업 잘하고있었는데 평일 이용객이 너무 적은것같다. 월요일은 20% 할인을 하기로한다.

class MondayDecorator implements AmusementPark {

AmusementPark park;

public MondayDecorator(AmusementPark park){

this.park = park;

}

@Override

public int getTicketCost(){

return (int)park.getTicketCost() \* 0.8;

}

}

public class Test {

public static void main(String[] arg){

AmusementPark park = new DisneyLand();

System.out.println("Ticket Cost => " + park.getTicketCost());

park = new MondayDecorator(park);

System.out.println("Ticket Cost => " + park.getTicketCost());

}

}

카드사와 제휴를 통해 할인도 하게됐다. 월요일할인과 중복할인이 적용된다.

class CardDecorator implements AmusementPark {

AmusementPark park;

public CardDecorator(AmusementPark park){

this.park = park;

}

@Override

public int getTicketCost(){

return (int)park.getTicketCost() \* 0.9;

}

}

public class Test {

public static void main(String[] arg){

AmusementPark park = new DisneyLand();

System.out.println("Ticket Cost => " + park.getTicketCost());

park = new MondayDecorator(park);

System.out.println("Ticket Cost => " + park.getTicketCost());

park = new CardDecorator(park);

System.out.println("Ticket Cost => " + park.getTicketCost());

}

}

월요일 할인 20% 를 받고 제휴카드 10% 할인까지 받았다.

이런식으로 실제 비즈니스 로직을 처리하는 객체를 인스턴스 멤버로 사용하고 메서드 요청이 들어오면 해당 객체가 처리한 로직에 적절한 가공을 해서 반환해주는게 데코레이터 패턴이다.