OOP 정리

C : 절차지향언어로서 함수로 구성되어 있음

객체지향언어 : 속성 + 행위를 갖는다.

ex )

사람 객체

속성 – 이름, 성별, 연령, 신장, 체중..

행위 – 자다, 먹다,

객체들이 갖고 있는 속성과 행위를 묶어서 하나의 틀을 만든다.

그 틀을 클래스 라고 함

클래스도 속성과 행위를 갖는다. -> 클래스에서는 속성을 멤버변수, 행위를 멤버메소드 라고 함

(메소드에는 멤버메소드와 생성자메소드가 존재하기 때문에 멤버 메소드 라고 칭해줘야 함)

클래스 : 객체를 생성하기 위한 틀, 유일한 식별자를 갖는다 (중복된 이름을 갖지 않아야 함)

식별자 중복 허용 – C++ : namespace / Java : Package

동일한 패키지 내에서 클래스명은 유일해야 한다.

Human h = new Human();

h : 객체형 변수 ( = 객체 ) , Human 클래스형 인스턴스 객체 / h 역시 유일해야 한다.

클래스 이름 유일하듯이 객체 명도 유일해야 한다.

클래스가 갖는 행위만큼의 메소드를 갖는다.

클래스와 객체의 차이점 : 클래스는 속성만을 갖지만, 객체는 속성만 갖는 것이 아니라 속성에 대한 값( 상태 값 )을 갖는다.

이러한 상태 값을 행위를 통해 변경하게 된다.

프로그램의 주체는 객체, 그 이유는 실제 상태값을 갖는 것이 객체이기 때문이다.

클래스가 주체라면 객체지향이 아니라 클래스지향이라고 해야함,

객체가 프로그램의 주체가 되기 때문에 객체지향프로그램이라고 하는 것

클래스가 하는 역할 : 서로 관계있는 데이터나 행위를 묶어준다.

왜 묶어줄까 ? (묶는 이유) : 캡슐화 (속성 – 이름, 성별, 연령 / 행위 – 자다(), 먹다(), 듣다() 등을 하나로 묶는다.)

캡슐화를 시키는 이유는 캡슐 내부와 캡슐 외부를 구분하기 위해서

왜 구분을 할까? : 캡슐 내부에 있는 무언가를 숨기겠다는 의도가 있기 때문임

캡슐화는 무엇을 통해서 캡슐화가 되는 가 : 캡슐화는 클래스를 통해서 이루어 진다.

관련 있는 속성과 관련 있는 행위를 하나로 묶어준다.

클래스 내부와 외부를 구분 짓기 위해서~~

캡슐 내부의 것을 외부로부터 숨기는 것을 가리켜서 **정보의 은닉**이라고 한다.

내부에 있는 것을 모두 숨기면 외부에서 사용할 수 없음 – 캡슐 외부와 캡슐 내부가 서로 의사소통을 할 수 있도록 특정 행위들을 공개해야 함. 보통 속성은 일반적으로 공개하지 않는 것이 좋다.

**공개된 행위들을 가리켜서 인터페이스**라고 한다.

interface : 서로 얼굴을 맞대는 것으로 생각하면 됨 – 연결해주기 위한 것

입출력 방법을 제공한다 – 메시지를 주고받는다. -> 이런 역할을 하는 것이 **인터페이스** ( 사전적인 의미 )

**정보의 은닉** : 캡슐화 안에 있는 것을 숨긴다. – 접근제한자 라는 것을 이용함

public protected package private

protected와 package 는 동일한 패키지 내부에서는 접근이 가능하다.

import java.util.\*; // class path, java 패키지 내 util 패키지 내에 있는 모든 클래스를 import

protected : 동일 패키지 + 상속관계 접근이 가능

package : 동일 패키지 접근 가능

캡슐화는 정보를 은닉할 수 있는 방법만을 제공한 것이다.

클래스 내에 있는 모든 멤버들이 다 공개가 되어 있는 것이 아님

외부에 공개되어있는 것보다내부에 숨겾 있느 것이 많은 경우 추상화 정도가 높다고 한다.

많은 것들이 공개될수록 추상화정도가 낮아지는 것.

추상화정도가 높아지면 내부에 있는 속성이나 알고리즘을 쉽게 변경할 수 있게 된다.

추상화정도가 높아지면 유지보수성이 높아진다.

(인터페이스인 메소드를 이용하게 되므로)

예를 들어 이름 속성을 공개하게 되면 이름 속성에 직접적으로 접근이 가능하게 되는데 이렇게 되면 매번 다른 것도 직접적으로 바꿔줘야 한다.

\* 상속

상속 : Is-A 관계

상속의 사용 목적은 재사용성의 향상이다.

캡슐화 한다는 것은 행위를 하는 알고리즘

각각의 클래스를 만들게 되면 어떤 문제가 발생하는 가 ?

동일한 속성 level, name 등과 fight() 행위가 겹친다. -> 클래스마다 재정의를 해야 한다.

이렇게 되면 각각 Archer 클래스형 인스턴스, Knight 클래스형 인스턴스 … 로 만들어지므로 인스턴스가 일치되지 않는다.

따라서 공통된 속성을 추출하게 된다. -> Character 클래스 (level이라는 공통속성과 fight()라는 공통행위)

Is-A 관계를 얘기할 때 하위 클래스에서 상위 클래스를 얘기해야 한다.

Archer, Knight, Lancer는 Character다 (O)

Character는 Archer다 (X)

공통된 특징을 추출해서 하나의 클래스로 만든 것 -> 일반화시킴

상속은 크게 일반화와 특수화가 있다. (일반화가 주로 쓰이고 특수화는 잘 사용되지 않는다.)

특수화 : 특정 클래스로부터 확장된 새로운 클래스를 만드는 것

속성에 대한 상태값은 객체가 갖는다.

그렇기 때문에 level에 대한 속성 값은 객체마다 할당이 된다.

fight()라는 메소드를 객체마다 fight()를 갖는게 아니고, 메소드 영역이 따로 있다.

Method는 Method 영역에 한 번만 로딩되는 것

Character 를 상속받아서 Archer, Knight, Lancer 클래스를 만들었는데 싸우는 부분은 다르다.

즉 fight() 부분이 각 클래스마다 바뀌어야 함 (level, name 등은 동일함)

fight라는 것을 별도로 추출해서 싸우는 행위를 갖는 클래스를 만든다.

OOP에서 제일 중요한 것 : **다형성** - 주어진 조건에 따라 다른 행위를 일으키는 것

다형성

오버로딩

1. 동일한 클래스 내에 동일한 식별자를 갖는 행위가 여러 개 존재한다.

2. 상속관계에서 부모 클래스의 행위를 자식 클래스에서 구현하는 것

public void sleep();

public void sleep(argslist) throws … {};

식별자는 같지만 매개변수의 개수나 타입이 다르다.

식별자와 매개변수를 이용해서 어떤 메소드를 실행할 것인지가 결정된다.

매개변수의 형태를 이용해서 어떤 메소드를 실행할 것인지 결정한다.

식별자와 매개변수를 합쳐서 **시그니처**(signature)라고 한다. -> 시그니처에 따라서 다른 메소드를 호출함 – 다형성

캐릭터 상속관계에서 knight는 곧 character 가 된다.

Knight a = new Knight();

**Character c = new Knight(); // Knight는 Character 가 성립하기 때문에 가능하다.**

**c는 Character형 클래스 객체이지만 Knight 형 인스턴스이다.**

**인스턴스 생성된 것은 Knight 인스턴스 이고 Character타입의 객체..**

c.fight();

c라는 Character 객체의 fight 메소드를 호출한다. – Character 타입의 객체이므로 Character 클래스에 fight 메소드가 있어야함

그러나 실행이 될 때는 인스턴스를 보기 때문에 Knight의 fight를 실행하려고 하지만 Knight에 fight가 없으면 Character의 fight가 호출이 된다. 만약 Knight에 fight 메소드가 있으면 Knight에 있는 fight 메소드를 실행한다.

c가 만약에 Archer()의 인스턴스라면 Archer의 fight가 실행이 된다.

인스턴스가 Knight인지 Archer인지에 따라서 (조건에 따라서) 다른 메소드가 호출된다. (다른 행위를 수행한다.) – 오버로딩

단 상위 클래스에서 또는 인터페이스에서 선언한 메소드를 하위 클래스에서 사용할 때 하위 클래스에서 재정의가 되어 있으면 재정의가 된 메소드를 호출하게 된다.

인터페이스 또는 추상클래스에서 정의된 메소드를 하위클래스에서 호출할 경우에는 오버로딩이라는 단어가 맞음

Character 클래스에서 fight() 메소드를 이미 정의 해놓았는데 하위 클래스에서 다시 fight()메소드를 정의 한다는 것은 기존에 있는 fight()메소드를 안 쓰겠다는 의미와 같으므로 오버라이딩

분류하기 헷갈리니까 다른 클래스의 메소드를 재정의 할 경우에는 전부 오버라이딩으로 하자 라는 개념이 된 것

사용자는 인터페이스를 사용할 줄만 알면 된다.

ex) 프로젝터라는 객체를 사용할 때 리모콘이라는 인터페이스를 이용한다.

프로젝터가 어떤 개발사든 간에 상관없이 사용자는 리모콘만 쓸줄 알면 됨

API가 그런 용도(Application Programing Interface) : 안에 돌아가는 로직이 어떻게 되는지 알 필요 없고 사용만 할 줄 알면 된다.

파일명은 파일안에 포함된 클래스 명과 동일하게 하깅

실행되는 도중에 특정 이벤트, 특정 조건에 의해서 싸우는 방법을 바꿀 수 있다는 장점이 있음

데코레이션 패턴

보통은 stream으로 끝나거나 reader 또는 writer

reader/wirter : 문자 입출력

stream : 바이트 입출력

reader : 입력스트림 (프로그램에 입력됨)

writer : 출력스트림 ( 프로그램 밖으로나감 )

InputStream : 바이트 입력 스트림

OutputStream : 바이트 출력 스트림

실제로 데이터를 기록하는 것은 FileWriter가 함

입력받은, 출력할 데이터를 변환하는 건 프로세싱 스트림

BufferedWriter는 FileWriter를 받아서 fw를 포장한다.

bw는 fw가 작업할 수 있도록 도와준다.

bw가 fw 객체를 꾸미고 있다고 할 수 있다. -> 데코레이팅 패턴

체인연쇄 패턴과 비슷하지만 약간 틀리다.

추상클래스는 객체를 생성할 수 없다 -> 인터페이스 역시 동일

인터페이스나 추상클래스를 구현한 / 상속받은 클래스는 객체 생성이 가능하다.

이런 클래스를 구현클래스, concrete 클래스, 구체화 클래스라고 한다.

첨가물이 추가된 경우

값을 구할 때 멤버로 갖고 있는 Espresso 가격 + 자기자신의 가격을 출력하게 되는 것

객체를 꾸며준다는 것은 기존 객체에 새로운 기능(행동)을 추가할 수 있다는 것이다.

체인연쇄패턴

명령을 수행하는 객체들이 있고 이 객체들을 묶는다.

이 객체안에 객체를 멤버로 가지고 그 객체안에 나를 넣는다.

또 나는 그 객체를 멤버로 갖기 때문에 계속 연결된다. - > 링크드리스트를 생각하면 됨

객체가 객체를 멤버로 가지고 그 객체가 다른 객체를 가리킨다.

데코레이션과는 반대의 개념

체인에서는 첫 번째 객체로부터 조건을 따진다. (처리 가능 여부)

만약에 처리할 수 없다고 하면 다른 객체로 넘어가서 다시 조건검사를 한다.

조건이 맞으면 수행하게 된다.

그렇다면 왜 체인연쇄인가?

첫 번재 명령을 던졌을 때 첫 번째 객체가 처리할 수 없다고 함

두 번째 객체가 처리 가능할 경우 두 번째 객체에서 실행을 한 후 나머지를 전부 실행하게 된다. (물론 하나만 실행하도록 할 수 있다.)

제일 첫번째 객체는 StdoutLogger

이 로거는 StderrLoger 인스턴스를 멤버로 갖고 StderrLoger는 FileLogger 인스턴스를 멤버변수로 갖는다.

각 레벨에서 하나씩만 출력하게 하려면

if(this.priority <= priority) 를 if(this.priority == priority) 으로 변경햊면 됨

새로운 Logger 클래스를 추가할 경우 추가해서 연결해주기만 하면 됨

여기서 Logger가 static final 상수값을 꼭 가질 필요가 없음

즉, 각각의 Logger들이 자신의 static final 값을 가지고 있으면 된다.

새로운 클래스를 만들 때는 레벨값만 중복이 안되게 하면 됨