2023/01/19

접근 제한자와 Encapsulation

Encapsulation : 데이터를 캡슐안에 넣는 것. 보호하기 위해서!

→ 변수는 **private 접근**으로 막기

→ **공개되는 메소드를 통한 접근 통로** 마련 setter/getter

(메소드에 정보보호 로직 작성 → ex. null 값이 들어오면 저장하지 않도록 하는 로직 추가)

- **Singleton 디자인 패턴 (객체를 하나만 생성하는 것)**

→ 싱글톤 패턴을 사용하는 경우 2가지

1. **여러 개의 객체가 필요 없는 경우**
    - 객체를 구별할 필요가 없는 경우 == 수정 가능한 멤버 변수가 없고 기능만 있는 경우
      → 이런 객체를 stateless 한 객체라고 함
      → 멤버 변수가 상태이므로, 수정 가능한 멤버 변수가 없는 것은 상태가 없다는 것임
2. **객체를 계속 생성/삭제 하는데 많은 비용이 들어 재사용이 유리한 경우**

- 외부에서 생성자에 접근 금지, **생성자 접근 제한자 private 설정**
- 외부에서 접근 가능한 getter 생성
- 객체 없이 외부에서 접근 가능하도록 getter 와 변수에 static 추가
- 외부에서는 언제나 getter 를 통해 객체 참조하므로 하나의 객체 재사용

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {
        // 생성자는 외부에서 호출못하게 private 으로 지정해야 한다.
    }
    public static Singleton getInstance() {
        return instance;
    }
    public void say() {
        System.out.println("hi, there");
    }
}
```

```
public class SingletonTest {
    public static void main(String[] args){
      Singleton sc1 = Singleton.getInstance();
      Singleton sc2 = Singleton.getInstance();
    }
}
```

다형성 ★★★★★

- 하나의 객체가 **많은 형 (타입)** 을 가질 수 있는 성질
- **상속 관계**에 있을 때, ***조상클래스의 타입으로 자식 클래스 객체를 레퍼런스*** 할 수 있다.

```
SpiderMan onlyOne = new SpiderMan("피터파커", false);
SpiderMan sman = onlyOne;
Person pson = onlyOne;
Object obj = onlyOne;

// Object > Person > SpiderMan
```

- 부모님은 마음이 넓어서 자식을 품을 수 있지만

  Object obj = SpiderMan 객체;
  Person pson = SpiderMan 객체;
- 자식은 마음이 좁아서 부모님을 담을 수 없다는 말이 떠오르네요

  SpiderMan sman = (SpiderMan) Person 객체

  → 결국 부모가 자식에게 맞춰준다.

예1) **다른 타입의 객체**를 다루는 배열

- 배열 : 같은 타입의 데이터를 묶음으로 다룬다
- **다형성으로** 다른 타입의 데이터 (Person, SpiderMan) 를 **하나의 배열로 관리**

```
Person[] persons = new Person[10];
persons[0] = new Person();
persons[1] = new SpiderMan();
```

- Object 는 모든 클래스의 조상이므로, Object 형 배열은 어떤 타입의 객체라도 저장 가능

  → 자바의 자료구조를 간단하게 처리 가능

  → **어떤 타입의 객체라도 다 저장한다고 했는데 기본형은 다 담을 수 있을까?**

→ 기본형은 Object 를 상속받지 않았지만 Object 배열에 기본형이 들어간다.

```
Object[] objs = new Object[4];
objs[0] = "Hello";
objs[1] = new Person();
objs[2] = new SpiderMan("", false);
objs[3] = 3; // auto boxing -> 자동 형변환
// - wrapper class 가 object 를 상속받고 있어서 가능한거라고?


for(int i=0;i<objs.length;i++) {
   System.out.println(objs[i].getClass().getName()); // objs[3] 을 찍었을 때 java.lang.Integer
}
```

- **AutoBoxing UnBoxing**

```
int a = 10;
Integer aobj = a; // int -> Integer : auto boxing

int sum2 = a + aobj.intValue(); // 원래 이렇게 해야 했는데 아래와 같이 해도 문제 없음
int sum = a + aobj; // Integer -> int : unboxing
```

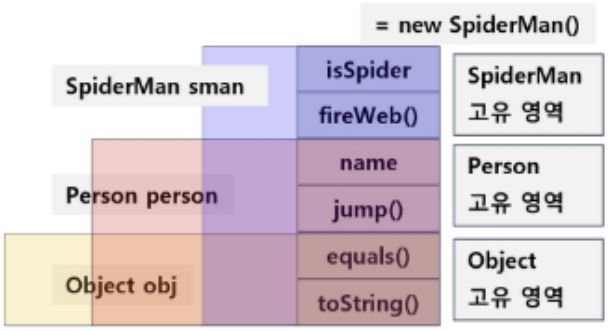
예2) 매개변수의 다형성

- 조상을 파라미터로 처리한다면, 객체의 타입에 따라 메소드를 만들 필요 없음

```
public void println(Object x){
    ...
}
```

다형성의 활용

- 메모리에 있는 것과 사용할 수 있는 것은 다르다
  - 메모리에 있어도 **참조하는 변수의 타입에 따라 접근할 수 있는 내용이 제한됨**




```
Person person = new SpiderMan();

// SpiderMan 객체를 생성해도, Person 타입의 변수에 넣기 때문에
// Object + Person 의 내용들만 접근이 가능하다
```

→ 메모리에 있는 전체 영역을 다 쓸 수 있도록 해보자

- **참조형 객체의 형 변환**
  - 작은 → 큰 : 묵시적 캐스팅
  - 자손 타입의 객체를 조상타입으로 참조하는 것은 문제 없음!
    - **조상의 모든 내용이 자식에 있기 때문에 걱정할 필요 없다**

```
Phone phone = new Phone();
Object obj = phone;
```

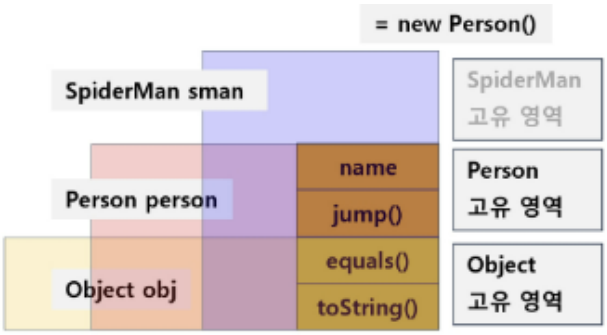
  - 큰 → 작은 : 명시적 캐스팅
  - 조상 타입을 자손 타입으로 참조할 때는 형변환 생략 불가

```
Phone phone = new SmartPhone(); // Phone - 조상, SmartPhone - 자식
SmartPhone sPhone = (SmartPhone) phone;
```

< 코드 풀이>

  - 아래의 코드에서는 Object, Person 만 메모리에 있음.
  - Person 객체를 SpiderMan 이라는 형으로 강제 형변환했기에 SpiderMan 객체는 메모리에 초기화되지 않은 상태임
  - 따라서 sman 객체에서 fireweb함수를 호출하면 메모리에 없는 객체이기 때문에 오류뜸

```
Person person = new Person();
SpiderMan sman = (SpiderMan) person;
sman.fireweb();
```



→ 조상을 무작정 자손으로 바꿀 수는 없다.

`instanceof` 로 실제 메모리 객체가 특정 클래스 타입인지 확인하고 형변환 해야 한다.

```
if(person instanceof SpiderMan){
   SpiderMan sman = (SpiderMan) person;
}
```

```
class SuperClass {
  String x = "super";

  public void method() {
    System.out.println("super class method");
  }
}

class SubClass extends SuperClass {
  String x = "sub";

  @Override
  public void method() {
    System.out.println("sub class method");
  }
}
```

- **참조 변수의 레벨에 따른 객체 멤버 연결**

```
SubClass subClass = new SubClass();
SuperClass superClass = subClass;
System.out.println(superClass.x); // super
superClass.method(); // sub class method

// superClass 타입 변수에 subClass 객체를 넣었음
// superClass.x 인 경우에는 superClass 타입으로 보기 때문에 super 출력
```

```
// superClass.method() 인 경우에는 superClass의 method 를 호출하려다가 !
// superClass 를 상속받고 있는 subClass 가 오버라이드한 method 를 발견하여
// sub class 의 method 가 호출됨

상속관계에서 조상클래스 메소드를 자식클래스에서 오버라이드했다면,
조상클래스 메소드 콜하는 순간, 어? 더 좋은 거 있네? 하면서 자식클래스 메소드를 호출해줌
(오버라이딩 하는 이유 : 더 좋게 구현한 것을 쓰려고 하기 때문에)
```

- **정적 바인딩**
  - **컴파일** 단계에서 **참조변수 타입에 따라** 연결이 달라짐

- **동적 바인딩**
  - 다형성 이용해서 메소드 호출 발생 시, **runtime**에 메모리의 실제 객체 타입으로 결정
  - 상속 관계에서 객체의 **instance method 가 재정의**되었을 때, **마지막에 재정의 된 자식 클래스 메소드 호출** (최대한 메모리에 생성된 실제 객체의 최적화된 메소드 동작)

`용도 : 점프하고 싶다!`

- 용도에 따른 가장 적합한 메소드 구성은?

```
public void useJump1(Object obj) {
  if (obj instanceof Person) {
    Person casted = (Person) obj;
    casted.jump();
  }
}

public void useJump2(Person person) {
  person.jump();
}

public void useJump3(SpiderMan spiderMan) {
  spiderMan.jump();
}
```

```
Object obj = new Object();

// 점프할 수 있는 애들
Person person = new Person();
SpiderMan sman = new SpiderMan();
```

1번) 활용성은 좋은데, Object 의 경우 jump() 할 수 없기 때문에, ap.useJump1(obj); 는 어차피 들어가도 아무 것도 못하고 나와야 하는 코드이다

→ 즉, 불필요한 객체들이 들어가서 쓸데없는 코드가 늘어날 수도 있음

2번) Person, SpiderMan 이 jump 라는 특성을 모두 활용하여 점프할 수 있음

3번) Person 이 jump 하지 못함

- Java API 처럼 공통기능인 경우에는 Object 를 파라미터로 쓰겠지만
  - **비즈니스 로직 상의 최상위 객체 사용**을 권장한다