👀

2023/01/20

> **abstract class :** 다형성에 날개를 달아주는 것

```
public class Car {
    private int curX, curY;
    public void reportPosition(){

    }
}
```

```
// 전
class Vehicle {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)\n", curX, curY);
  }

  // addFuel() 은 공통모듈이어서 Vehicle 클래스에서 구현했는데 사용하지 않는 메소드가 됨
  public void addFuel() {
    System.out.println("연료가 필요해");
  }
}
```

```
public class DieselSUV extends Vehicle{

  @Override
  public void addFuel() {
    System.out.println("주유소에서 주유");
  }
}

public class Lamborgini extends Vehicle {
  @Override
  public void addFuel() {
    System.out.println("집에서 주유");
  }
}
```

```
// 후
abstract class Vehicle {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)\n", curX, curY);
  }

  public abstract void addFuel(); // 메소드 선언부만 남기고, 구현부는 세미콜론으로 대체
}
```

- addFuel() 은 자손클래스에서 **반드시 재정의해서 사용하기 때문에**
  조상의 구현이 무의미한 메소드가 되어버린다.
  - 메소드 선언부만 남기고, 구현부는 세미콜론으로 대체
  - **구현부가 없다**는 의미로 메소드에 **abstract** 붙이기

- 객체를 생성할 수 없는 클래스라는 의미로 **클래스 선언부에도 abstract 추가**

→ Abstract method design pattern

- abstract 클래스는 new 못함. **상속 전용**의 클래스
  - 클래스에 구현부가 없는 메소드가 있어서 객체 생성 불가
  - **상위 클래스 타입으로써 자식을 참조할 수는 있음**
    - 근데 만약 자식이 abstract 메소드를 오버라이딩 안한 경우 오류 나나?

    `Class 'Lamborgini' ***must either be declared abstract or implement abstract method*** 'addFuel()' in 'Vehicle'

```
Vehicle[] vehicles = {
        new DieselSUV(),
        new ElectricCar(),
        new HorseCart()
};

for(Vehicle v: vehicles) {
  v.reportPosition();
  v.addFuel();
}

Vehicle v = new Vehicle(); // 불가능
```

- 조상클래스에서 상속받은 abstract 메소드를 재정의하지 않은 경우
  - 클래스 내부에 abstract 메소드가 있는 상황이므로, 자식클래스는 abstract 클래스로 선언되어야 함
    - abstract 메소드를 오버라이딩 하지 않은 경우에는 abstract 클래스로 선언해야 하는데 그러면 자식도 객체를 생성할 수 없음
  - abstract 클래스는 구현의 `강제`를 통해 프로그램 안정성 향상
    - 구현의 강제 : 오버라이드 하지 않으면 컴파일 안시켜줄거야 !

      → 실수로 오버라이드하지 않는 상황을 미연에 방지

interface

- 최고 수준의 추상화 단계 : 일반 메소드는 모두 abstract 형태
- 모든 멤버 변수 : public static final 이며 생략 가능
- 모든 메소드 : public abstract 이며 생략 가능

- 접근 제한자 : **조상이 선언한거보다 넓은 제한자 범위로만 가야 한다**

→ 인터페이스의 메소드를 재정의 했을 때는 **public 이 되어야 함**

void 만 보인다고 해서 default 가 아니라 public 이라는 것을 잊지 말자!

```
public interface MyInterface {
    (public static final) int member = 10;
    (public abstract) void method2(int param);
}
```

- 인터페이스끼리는 **다중 상속** 가능
  - 클래스 상속의 경우에는 어떤 메소드를 실행해야 할 지 모른다는 문제때문에 다중 상속이 안됐는데, 인터페이스는 **헷갈릴 메소드 구현 자체가 없음.**
- 인터페이스 상속과 인터페이스 구현의 차이는?
  - 클래스에서 `implements` 를 사용해서 인터페이스를 **구현**해줘야 함
  - **인터페이스끼리는** 다중 **상속이 가능**하다
    - interface A extends B,C

```
interface Fightable {
    public abstract int fire();
}

interface Transformable {
    public abstract void changeShape(boolean isHeroMode);
}

interface Heroable extends Transformable, Fightable {
    void upgrade();
}
// Heroable 에는 abstract 메소드가 3개 존재

class IronMan implements Heroable{

    @Override
    public int fire() {
        return 0;
    }

    @Override
    public void changeShape(boolean isHeroMode) {
        System.out.println("모양 변경");
    }

    @Override
    public void upgrade() {
        System.out.println("버전 업");
    }
}
```

- 다형성은 조상클래스 뿐만 아니라 조상 인터페이스에도 적용
  - 인터페이스도 조상타입이 될 수 있다.
    조상타입이 될 수 있다 == 다형성을 적용할 수 있다.

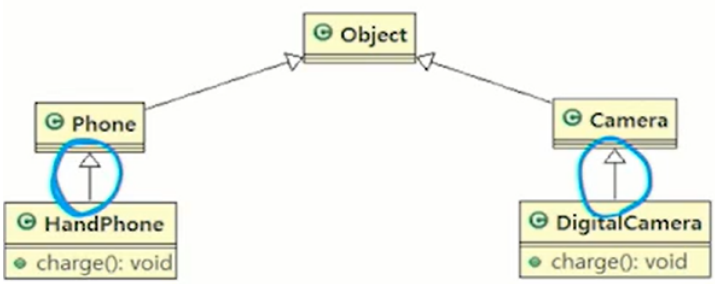
```
public class IronManTest {
  public static void main(String [] args) {
    IronMan iman = new IronMan();
    Object obj = iman; // 조상클래스인 object 로 참조하는 것이 가능하다.

    // 인터페이스들로 참조도 가능
    Transformable t = iman;
    Heroable h = iman;
    Fightable f = iman;
  }
}
```

- 인터페이스의 필요성
  1. 구현의 강제로 표준화 처리
     - 인터페이스에 선언된 메소드들이 모두 abstract 메소드 → 구현 강제
  2. 인터페이스를 통한 간접적인 클래스 사용으로 **손쉬운 모듈 교체 지원**

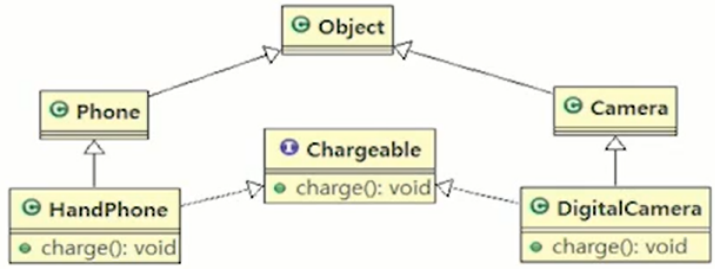
- 도트프린터 → 레이저프린터로 변경하는 것이 쉬움

```
public interface Printer {
  void print(String fileName);
}

public class LaserPrinter implements Printer {
  @Override
  public void print(String fileName){
    System.out.println("레이저프린터");
  }
}

public class DotPrinter implements Printer {
  @Override
  public void print(String fileName){
    System.out.println("도트프린터");
  }
}
```

3. 서로 상속 관계가 없는 클래스들에게 **인터페이스를 통한 관계부여**로 다형성 확장
    - 상속은 단일 상속만 가능하므로, 관계를 맺는 것에 한계가 있음




```
void badCase() {
  Object[] objs = { new HandPhone(), new DigitalCamera() };
  for(Object obj: objs){
    if(obj instanceof HandPhone){
      HandPhone phone = (HandPhone) obj;
      phone.charge();
    } else if(obj instanceof DigitalCamera){
      DigitalCamera camera = (DigitalCamera) obj;
      camera.charge();
    }
  }
}
// DigitalCamera 는 Camera 를 이미 상속받았으므로,
// 충전 가능한~ 이러한 내용을 상속받을 기회가 없음
```




```
void goodCase(){
  Chargeable[] objs = { new HandPhone(), new DigitalCamera() };
  for(Chargeable obj: objs){
    obj.charge();
```

```
  }
}
```

4. 모듈 간 독립적 프로그래밍 가능

   → 개발기간 단축

- **Default method**
  - 인터페이스에 선언된 구현부가 있는 일반 메소드

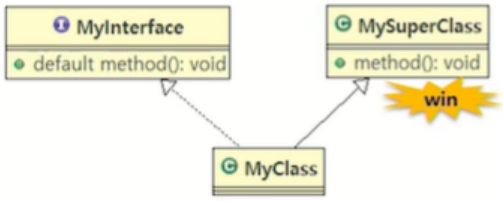
```
interface DefaultMethodInterface {
  void abstractMethod();
  (default) void defaultMethod(){}
}
```

  - 기존 인터페이스 기반으로 동작하는 라이브러리에 인터페이스 추가해야 하는 일 생김
  - 기존 방식이면 모든 구현체들이, 추**가되는 메소드를 오버라이드해야함**
  - default 메소드는 abstract 가 아니므로 반드시 구현해야 할 필요는 없음

```
interface Aircon {
    void makeCool();
    default void dry(){
        System.out.println("hi");
    };
}

class OldisButGoodies1 implements Aircon {
    @Override
    public void makeCool() {
        System.out.println("전체 냉각");
    }
}

class OldisButGoodies2 implements Aircon {
    @Override
    public void makeCool() {
        System.out.println("집중 냉각");
    }
}

class NoWindAircon implements Aircon {
    @Override
    public void makeCool() {
        System.out.println("바람 없이 시원하다");
    }

    @Override // default method 로 선언한 것도 재정의 가능
    public void dry() {
        System.out.println("종료시 자동 건조");
    }
}

public class poli {
    public static void main(String[] args) {
        Aircon[] aircons = {
                new OldisButGoodies1(),
                new OldisButGoodies2(),
                new NoWindAircon()
        };
        for (Aircon aircon : aircons) {
            aircon.makeCool();
            aircon.dry();
            /*
              이후에 추가된 dry 메소드는 default method로 선언하면
              바로 접근해서 dry() 도 사용가능
            */
```

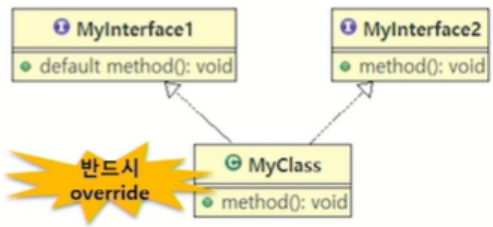
```
        }
    }
}
```

- 인터페이스를 다중상속할 수 있는 근거는, 여러 개의 인터페이스에 **동일한 이름의 메소드가 있더라도 구현부가 없으니까 충돌날 일이 없다**
  - 근데 default method 는 구현부가 있는 메소드이다. 이 경우 어떻게 되는가?
  - Method의 우선순위에 따른다
    - **super class 의 method 우선** : super **class가 구체적인 메소드 갖는 경우** default method 는 무시됨




```
interface MyInterface {
    default void hi() {
        System.out.println("hi");
    }
}

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

class MyClass extends MySuperClass implements MyInterface {

}

public class poli {
    public static void main(String[] args) {
        MyClass my = new MyClass();
        my.hi();
        // super hi가 출력됨 -> class 가 가지고 있는 구체적인 메소드가 우선
    }
}
```

- **interface 간 충돌** : 하나의 인터페이스에서 default method 제공, 다른 인터페이스에서도 같은 이름의 메소드 (default 여부와 무관) 가 존재할 경우,
  sub class는 반드시 오버라이드해서 충돌 해결해야 함




```
interface MyInterface1 {
    default void method() {
        System.out.println("interface1 hi");
```

```
    }
}

interface MyInterface2 {
    void method();
}

class MyClass implements MyInterface2, MyInterface1 {
    @Override
    public void method(){ // MyInterface2 의 method 를 오버라이드함
        System.out.println("myclass");
    }
}

public class poli implements MyInterface1, MyInterface2 {
// Class 'poli' must either be declared abstract or implement abstract method 'method()' in 'MyInterface2'
    public static void main(String[] args) {
        MyClass my = new MyClass();
        my.method(); // myclass 가 출력됨
    }
}
```

- **Static method**
  - 구현체 클래스 없이 바로 인터페이스 이름으로 접근하여 사용가능

```
interface StaticMethodInterface {
  static void staticMethod(){
    System.out.println("static method");
  }
}

public class StaticMethodTest {
  public static void main(String[] args) {
    StaticMethodInterface.staticMethod();
  }
}
```

- 다형성 : 형태가 다양한 특징(성질)
  - **상속 관계**에 있을 때, ***조상클래스의 타입으로 자식 클래스 객체를 레퍼런스*** 가능

  "상속이 전제조건"

  → 상속을 통해 `클래스 계층도` 가 나옴

  → 상속트리를 통해 타입의 크기판별이 가능해지기 때문에 상속이 필요함

  기능이 많고 적음으로 **타입 크고 작음을 판별할 수 없음 !!**

  개념적으로 상위에 있는게 큰 타입

  - 위로 갈수록 추상화된 상위 타입
  - 밑으로 갈수록 구체화된 하위타입

  = 위와 아래를 따질 수 없는 **형제들은 다형성을 사용할 수 없음**

  ex) 물을 어디에 담느냐에 따라 (ex. 텀블러, 머그잔, 소주잔)

  마시는 방법, 양의 차이가 나타남

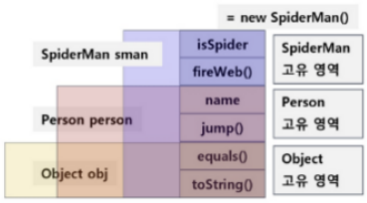
물 (객체) : 그릇 (참조변수)

1 : N

SpiderMan Person, Object

- 그릇이 물마시는 방법을 결정함
- **객체를 어떤 참조변수에 담느냐에 따라 객체 사용방법 (할 수 있는 일)이 달라진다**

> 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
```

- (좌변 = 우변)

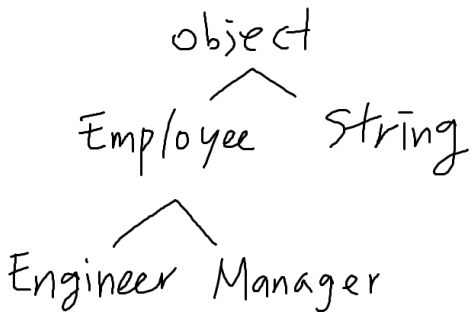
  ≥ 이면 묵시적 형변환 / ex. double a = int 형변수;

Engineer e = new Engineer();

담을 수 있는 참조 변수가 Employee, Object 도 가능

→ Employee e = new Engineer(); 를 사용하면 Employee, Object 의 메소드만 사용가능 (Engineer의 메소드 사용불가)

→ **내가쓰고싶은 기능을 다 갖고 있는 타입을 사용해라 !!!!!!**



- 상위타입 (그릇이 커지면, 담는대상은 많지만 할 수 있는 일은 줄어든다.
  모든 것을 아우르는 /표현하는 공통적인 일만 할 수 있기 때문)
- 하위 타입 (그릇이 작아지면, 할 수 있는 일은 많아진다. 구체적인 일들)

- 다형성 사용 (적용 예시)
  1. **배열의 이형집합화 :** 껍데기는 똑같은데 들어가는 게 다름

```
<  > employees[] = new < >[2];
employess[0] = new Engineer();
employees[1] = new Manager();
```

→ Manager, Engineer 를 둘 다 담을 수 있는 타입이 상위 타입으로 와야 함

< > 후보는 : Employee, Object

→ 사용목적에 따른 타입을 선택해야 함 (저 객체들로 어떤 일을 할 것인지가 중요함)

→ **참조타입이 담긴 객체의 사용법을 결정한다.**

Object o = new Employee(); → **(Employee 타입이 담긴) o 객체의 사용법을 결정**

2. **매개변수의 다형성**

```
class Company {
    increaseSalary(<> e){
      e.salary +=100000000;
    }
}
```

- 다양한 타입을 e 변수에 받아서 salary 에 1억씩 더해줘야 하기때문에
  e는 salary 라는 변수에 접근해야 하므로 Object 타입이 아니라 Employee 타입을 해줘야 함.

3. **리턴타입의 다형성**

```
< > xxx() {
    return new Employee();
    return new Manager();
}
// 직원으로서 쓰게하고 싶으면 Employee, 객체로 쓰게하고 싶으면 Object 사용
```

- **동적바인딩**

  = 컴파일 시에 인식되는 메소드가 무조건 실행(런타임) 되는 것이 아니고,
  런타임에 실제 가리키는 객체의 구현 메소드를 따져보고
  자식 쪽에 재정의된 메소드가 있을 경우, 자식의 재정의된 메소드를 결정하여 실행한다.

  - 다형성은 `동적바인딩`을 기반으로 하는 것!
  - 재정의는 부모의 것이 마음에 안들어서 재정의한거니까 그걸 써야함
  - 컴파일러는 구현부가 아니라 프로토타입만 보고 컴파일하는거임!
  - 동적바인딩을 기대해볼 수 있는 상황은 상위타입으로 하위 타입을 가리키는 상황!!!!!!!!!!!!

```
Engineer employees[] = new Engineer[2];

employees[0] = new Engineer();
employees[1] = new Manager();
// 여기서 에러남
// 왜냐면 Engineer 랑 Manager는 그냥 부모만 같을 뿐 연관성이 없음

Employee employees[] = new Employee[2];

employees[0] = new Engineer();
employees[1] = new Manager();

for(Employee em: employees){
```

```
  System.out.println(em.xxx);
}
```

추상 클래스

- 미완성, 덜 구현된 클래스
  → 직접 객체 생성 불가 (new 를 못한다는 거임. 객체를 생성하면, 미완성인 메소드를 호출해버릴까봐 = 컴파일을 막을 길이 없기 때문에) , 하위 클래스가 요구됨

```
abstract class Shape {
  int area;
  // 모든 도형에서 가능한 변수를 넣어야 함
  // height, width 같은거주면 안됨. 걔는 사각형 용이니까
  void calc();
}

class ShapeUtil { // 어떤 도형이든지 면적을 계산해주는 것
  calcArea(Object o -> Shape s){
  // object 라는 그릇으로는 도형에 관한 어떤 것도 할 수 없음.
  // 모든 도형을 표현할 수 있는 상위타입이 필요 -> Shape 추상 클래스

  s.calc();
  //shape 의 calc() 라고 쓰고, 실행은 넘어오는 구체적 도형의 calc() 라고 읽는다
  // 넘어오는 객체는 무조건 calc를 구현하고 잇는 서브타입의 객체거나 shape 객체
  }
}
```

```
Shape
↑
class Circle {
  int radius;
  calc(){
    area = π * r^2;
  }
}
```

```
Shape
↑
class Rectangle {
  int width;
  int height;
  calc(){
    area = width*height;
  }
}
```

- 왜 추상클래스를 제시할 수 밖에 없었는지 Shape 관점에서 접근하자
- ShapeUtil 을 짜는 입장에서는 이 로직이 필요한 대상을 받아들여야하는데,
  그 대상을 받아들일 만한 적당한 타입이 없었음. 내가 필요한 그릇을 만들어보자!
- 공통상위 타입인 Shape 을 만들었고, 도형으로서 필요한 것들을 shape 안에 다 넣는다.
- ShapeUtil 을 쓰고 싶은 모든 사람들에게, 도형을 다루는 기능은 다 만들어놧으니까, 이걸 상속받은 도형만 계산할 수 있다고 말한다.
- 작성하는 코드는 Shape 기준이지만, 런타임에는 동적바인딩을 이용해서 rectangle이나 circle 이 들어와도 됨.

추상클래스는 변수, 상수, 구현메소드, 추상메소드 다 가질 수 있음

→ 하지만 추상클래스로 만들었기에 직접 new 를 못할 뿐이고, 상속받은 하위타입을 통해 완전해지고 **추상클래스의 모든 것들이 메모리에 올라감**. 직접적으로 new만 못할뿐

```
// Rectangle.java
package com.ssafy.shape;

import com.shape.Shape;

public class Rectangle extends Shape{
  private int width;

  private int height;

  public Rectangle(int width, int height) {
    super();
    this.width = width;
    this.height = height;
  }

  public int getWidth() {
    return width;
  }

  public void setWidth(int width) {
    this.width = width;
  }

  public int getHeight() {
    return height;
  }

  public void setHeight(int height) {
    this.height = height;
  }

  @Override
  public void calcArea() {
    // TODO Auto-generated method stub
    setArea(width*height);
  }

}
```

```
// shapetest
package com.ssafy.shape;

import com.shape.Shape;
import com.shape.ShapeUtil;

public class ShapeTest {

  public static void main(String[] args) {
    // TODO Auto-generated method stub
    Shape r = new Rectangle(3,4);
    ShapeUtil.calcArea(r);

  }

}
```

```
// ShapeUtil
class ShapeUtil {
    void calcArea(Shape s) {
      System.out.print(s.
}
```

> 인터페이스 : How 가 아니라 What
>
> - **인터페이스가 추상클래스랑 뭐가다른거야 ?!!?**

- 사용자 측면 : 사용 방법
- 구현자 측면 : 약속대로 동작하는 구현에 책임이 생김

→ 구현과 사용의 분리

- 인터페이스의 특징
  1. 객체 생성 불가 : 클래스가 아님 (클래스가 지녀야하는 명세를 가진 것이 인터페이스)
  2. Java 8 이전 : 상수 + 추상 메소드로 구성

     Java 8 이후: 상수 + default method + 추상 메소드 + static method 로 구성
  3. 하위 클래스 입장에서 다중상속 가능
  4. “Is a” 관계가 아닌 관계 표현

리모콘의 관점에서 봐라 ~

객체를 식별하는 것이 우선!