

객체 지향 핵심



작성자: 신채린

상속

1. 객체 간의 상속은 어떤 의미일까
 - 클래스에서 상속의 의미
 - 상속을 구현하는 경우
2. 상속을 활용한 멤버십 클래스 구현하기
 - 멤버십 시나리오
 - 일반 고객(Customer) 클래스 구현
 - 우수 고객(VIPCustomer) 구현
 - protected
 - Customer와 VIPCustomer 테스트하기
 - 접근 제한자(access modifier)의 가시성
3. 상속에서 클래스 생성 과정과 형 변환
 - 하위 클래스가 생성되는 과정
 - 상속에서의 메모리 상태
 - 예시
 - 상속에서의 형 변환
4. 메서드 재정의하기 (Overriding)
 - 하위 클래스에서 메서드 재정의하기
 - @Override 애노테이션 (Annotation)
 - 형변환과 오버라이딩 메서드 추출
 - 가상 메서드 (virtual method)
5. 메서드 재정의와 가상 메서드 원리
 - 메서드는 어떻게 호출되고 실행 되는가?
 - 가상 메서드의 원리
 - 재정의된 메서드의 호출 과정

다형성 (polymorphism)

- 다형성과 다형성을 사용하는 이유
 - 다형성이란
 - 다형성의 예
 - 다형성을 사용하는 이유?
 - 다형성을 활용한 멤버십 프로그램 확장
 - 다형성을 사용함으로써 갖는 장점?
- 7. 상속은 언제 사용할까?
- 8. 다운 캐스팅과 instanceof
 - 하위클래스로 형 변환, 다운 캐스팅

[instanceof](#)

[예시](#)

[추상 클래스](#)

[9. 추상 클래스의 의미와 구현하는 방법](#)

[추상 클래스란](#)

[주의하기](#)

[예시](#)

[추상 클래스 구현하기](#)

[추상클래스 사용하기](#)

[10. 추상 클래스를 활용한 템플릿 메서드 패턴](#)

[템플릿 메서드](#)

[템플릿 메서드 구현하기 예제](#)

[final 예약어](#)

[public static final 상수값 정의하여 사용하기](#)

[템플릿 메서드 활용하기 \(예제\)](#)

[인터페이스](#)

[11. 구현 코드가 없는 인터페이스](#)

[인터페이스란](#)

[인터페이스의 요소](#)

[인터페이스 선언과 구현](#)

[예시](#)

[타입 상속과 형 변환](#)

[12. 인터페이스는 왜 쓰는가?](#)

[인터페이스를 활용한 다형성 구현](#)

[인터페이스의 역할은? 인터페이스가 하는 일](#)

[13. 프로그램에서 인터페이스의 역할과 다형성](#)

[인터페이스와 다형성 구현하기](#)

[인터페이스와 strategy pattern](#)

[인터페이스의 사용 예](#)

[인터페이스를 활용한 dao 구현하기](#)

[14. 인터페이스의 요소들](#)

[여러 개의 인터페이스 구현하기](#)

[인터페이스 상속](#)

[인터페이스 구현과 클래스 상속 함께 사용하기](#)

[15. 여러 인터페이스 구현하기, 인터페이스의 상속](#)

[여러 개의 인터페이스 구현하기](#)

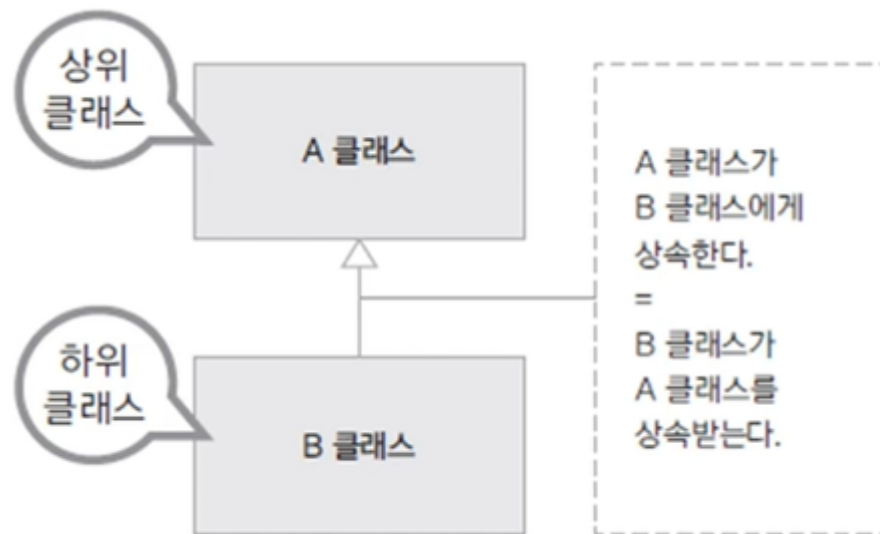
[인터페이스 상속](#)

상속

1. 객체 간의 상속은 어떤 의미일까

클래스에서 상속의 의미

- 새로운 클래스를 정의할 때 이미 구현된 클래스를 상속(inheritance) 받아서 속성이나 기능이 확장되는 클래스를 구현함
- 이미 구현된 클래스보다 더 구체적인 기능을 가진 클래스를 구현해야 할때 기존 클래스를 상속함
- 유사한 클래스를 만들 때, 기존의 클래스를 가져다 만든다.



- 상속하는 클래스: 상위 클래스, parent class, base class, super class
- 상속받는 클래스: 하위 클래스, child class, derived class, subclass

- `class B extends A {}`
 - extends 뒤에 하나의 클래스만 올 수 있다. (C++은 다중 상속이 가능하지만, Java에서는 single inheritance 만 가능하다)

상속을 구현하는 경우

- 상위 클래스는 하위 클래스보다 일반적인 개념과 기능을 가짐
- 하위 클래스는 상위 클래스보다 구체적인 개념과 기능을 가짐
- 하위 클래스가 상위 클래스의 속성과 기능을 확장 (extends)한다는 의미



```

class Mammal{
}

class Human extends Mammal{
}
  
```

2. 상속을 활용한 멤버십 클래스 구현하기

멤버십 시나리오

회사에서 고객 정보를 활용한 맞춤 서비스를 하기 위해 일반고객(Customer)과 이보다 충성도가 높은 우수고객(VIPCustomer)에 따른 서비스를 제공하고자 함

물품을 구매 할때 적용되는 할인율과 적립되는 보너스 포인트의 비율이 다름
여러 멤버십에 대한 각각 다양한 서비스를 제공할 수 있음
멤버십에 대한 구현을 클래스 상속을 활용하여 구현해보기

일반 고객(Customer) 클래스 구현

- 고객의 속성 : 고객 아이디, 고객 이름, 고객 등급, 보너스 포인트, 보너스 포인트 적립비율
- 일반 고객의 경우 물품 구매시 1%의 보너스 포인트 적립

```

package ch01;

public class Customer {

    private int customerID;
    private String customerName;
    private String customerGrade;
    int bonusPoint;
    double bonusRatio;

    public Customer() {
        customerGrade = "SILVER";
        bonusRatio = 0.01;
    }
  
```

```

    }

    public int calcPrice(int price) {
        bonusPoint += price * bonusRatio;
        return price;
    }

    public String showCustomerInfo() {
        return customerName + "님의 등급은 " + customerGrade +
            "이며, 보너스 포인트는" + bonusPoint + "입니다";
    }
}

```

우수 고객(VIPCustomer) 구현

매출에 더 많은 기여를 하는 단골 고객

제품을 살때 10%를 할인해 줌

보너스 포인트는 제품 가격의 5%를 적립해 줌

담당 전문 상담원이 배정됨

- Customer 클래스에 추가해서 구현하는 것은 좋지 않음
- VIPCustomer 클래스를 따로 구현
- 이미 **Customer**에 구현된 내용이 중복되므로 **Customer**를 확장하여 구현함(상속)

```

public class VIPCustomer extends Customer{

    private int agentID;
    double salesRatio;

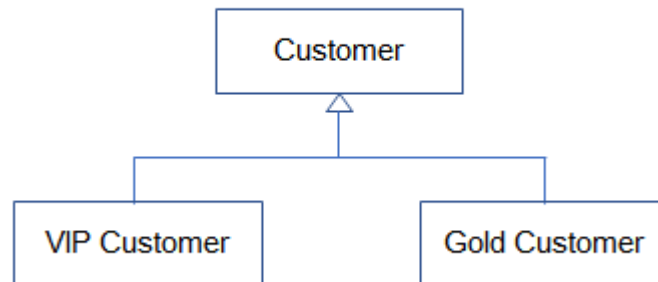
    public VIPCustomer() {
        customerGrade = "VIP";    //Customer.java에서 customerGrade 변수가 private으로 세팅되어
        있기 때문에 오류 발생
        bonusRatio = 0.05;
        salesRatio = 0.1;
    }

    public int getAgentID() {
        return agentID;
    }
}

```

- private

- 해당 클래스 자체에서만 사용이 가능하며, 상속받은 하위 클래스 역시 사용하지 못함.
- Protected만이 패키지 내부와 내부 및 하위 클래스에서 사용이 가능하다.



protected

- 상속받은 관계에서는 public이고, 외부에서는 private의 기능을 구현한 키워드이다.
- 상위 클래스에 protected로 선언된 변수나 메서드는 다른 외부 클래스에서는 사용할 수 없지만 하위 클래스에서는 사용이 가능하다.
- 상위 클래스에 선언된 private 멤버 변수는 하위 클래스에서 접근 할 수 없음
- 외부 클래스는 접근 할 수 없지만, 하위 클래스는 접근 할 수 있도록 protected 접근 제어자를 사용
- 예시

```

protected int customerID;
protected String customerName;
protected String customerGrade;

//getter, setter 구현
...
public int getCustomerID() {
    return customerID;
}

public void setCustomerID(int customerID) {
    this.customerID = customerID;
}
  
```

```

public String getCustomerName() {
    return customerName;
}

public void setCustomerName(String customerName) {
    this.customerName = customerName;
}

public String getCustomerGrade() {
    return customerGrade;
}

public void setCustomerGrade(String customerGrade) {
    this.customerGrade = customerGrade;
}

```

Customer와 VIPCustomer 테스트하기

```

public class CustomerTest {

    public static void main(String[] args) {
        Customer customerLee = new Customer();
        customerLee.setCustomerName("이순신");
        customerLee.setCustomerID(10010);
        customerLee.bonusPoint = 1000;
        System.out.println(customerLee.showCustomerInfo());

        VIPCustomer customerKim = new VIPCustomer();
        customerKim.setCustomerName("김유신");
        customerKim.setCustomerID(10020);
        customerKim.bonusPoint = 10000;
        System.out.println(customerKim.showCustomerInfo());
    }
}

```

접근 제한자(access modifier)의 가시성

	외부 클래스	하위 클래스	동일 패키지	내부 클래스
public	O	O	O	O
protected	X	O	O	O
선언되지 않음 (default)	X	X	O	O
private	X	X	X	O

3. 상속에서 클래스 생성 과정과 형 변환

하위 클래스가 생성되는 과정

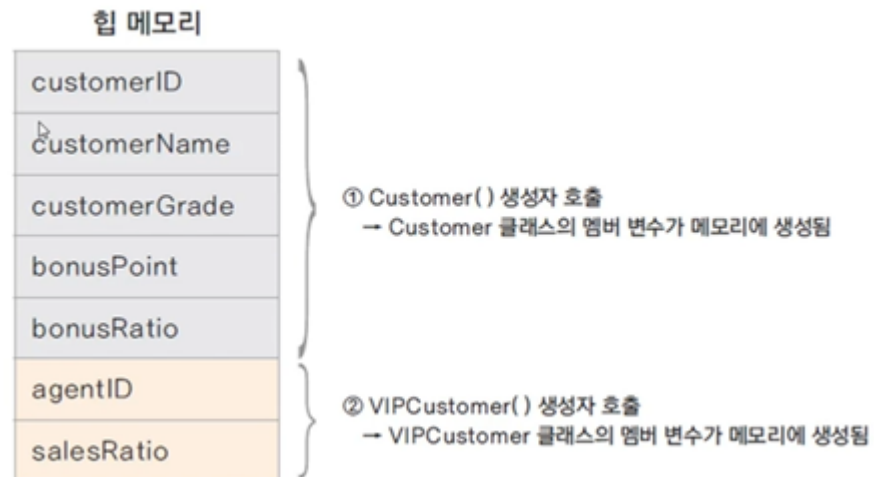
- 하위 클래스가 생성 될 때 **상위 클래스가 먼저 생성된다.**
- 상위 클래스의 생성자가 호출되고, 하위 클래스의 생성자가 호출된다.
- 하위 클래스의 생성자에서는 무조건 **상위 클래스의 생성자가 호출되어야 한다.**
- 하위 클래스에서 상위 클래스의 생성자를 호출하는 코드가 없는 경우, 컴파일러는 상위 클래스 기본 생성자를 호출하기 위한 `super()`를 추가한다.
- `super()`로 호출되는 **생성자는 상위 클래스의 기본 생성자(default constructor)이다.**
 - `super()`는 상위 클래스의 메모리 위치/참조 값을 가지고 있다.
- 만약 상위 클래스의 , (매개변수가 있는 생성자만 존재하는 경우) 하위 클래스는 **명시적으로 상위 클래스의 생성자를 호출**해야 한다.

기본 생성자가 없는 경우

- 예시) `super(customerID, customerName)`

상속에서의 메모리 상태

- 상위 클래스의 인스턴스가 먼저 생성이 되고,
- 하위 클래스의 인스턴스가 생성됨



예시

- 상속을 사용하여 고객관리 프로그램 구현하기
 - 고객의 등급에 따라 차별화된 서비스 제공 (다른 할인율과 적립금)
 - SilverCustomer
 - VIPCustomer - 제품 구매시 10% 할인, 보너스 포인트 5% 적립, 담당 상담원 배정

상속에서의 형 변환

상위 클래스로의 묵시적 형변환 (업캐스팅)

- 상위 클래스 형으로 변수를 선언하고, 하위 클래스의 생성자로 인스턴스를 생성할 수 있음

```
Customer customerLee = new VIPCustomer();
```

- 상위 클래스 타입의 변수에 하위 클래스 변수가 대입;

```
VIPCustomer vCustomer = new VIPCustomer();

addCustomer(vCustomer);

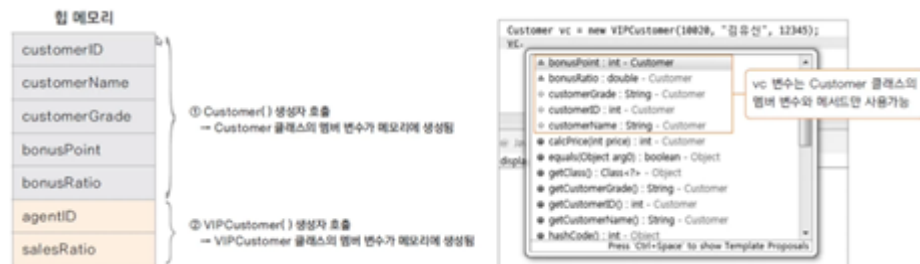
int add Customer(Customer customer) {
}
```

- 하위 클래스는 상위 클래스의 타입을 **내포**하고 있으므로 상위 클래스로 **묵시적 형변환**이 가능함
- 상속관계에서 모든 하위 클래스는 상위 클래스로 묵시적 형변환이 됨
 - 그 역은 성립하지 않음
- 예시



형변환에서의 메모리

- `Customer vc = new VIPCustomer()` 에서 vc가 가리키는 것은?



- VIPCustomer() 생성자의 호출로 인스턴스는 모두 생성되었지만, 타입이 Customer으로 형변환 됐으므로 **접근할 수 있는 변수나 메서드는 Customer의 변수와 메서드**임
 - 상위 클래스의 생성자 호출로 인스턴스가 생성되었으므로, 접근이 가능한 변수나 메서드는 상위 클래스의 변수와 메서드이다.

클래스 계층구조가 여러 단계인 경우



- Human은 내부적으로 Primate과 Mammal의 자료형을 모두 내포하고 있음
- `Primate aHuman = new Human();`
- `Mammal mHuman = new Human();`

4. 메서드 재정의하기 (Overriding)

하위 클래스에서 메서드 재정의하기

오버라이딩(overriding)

- 상위 클래스에 정의된 메서드의 구현 내용이 하위 클래스에서 구현할 내용과 맞지 않는 경우 하위 클래스에서 동일한 이름의 메서드를 재정의 할 수 있음
 - 이미 구현된 코드가 있지만, 그 코드와 다른 코드로 재구현하는 것이다.
- 예제의 Customer 클래스의 calcPrice()와 VIPCustomer의 calcPrice() 구현 내용은 할인율과 보너스 포인트 적립 내용 부분의 구현이 다름.
- 따라서 VIPCustomer 클래스는 calcPrice() 메서드를 재정의 해야 함
- `VipCustomer.java`

```

@Override
public int calcPrice(int price) {
    bonusPoint += price * bonusRatio;
    return price - (int)(price * salesRatio);
}
  
```

@Override 애노테이션 (Annotation)

- 재정의된 메서드라는 의미로 선언부가 기존의 메서드와 다른 경우 에러 발생
- 애노테이션은 **컴파일러에게 특정한 정보를 제공해주는 역할**
- 주로 사용되는 자바에서 제공되는 애노테이션

애노테이션	설명
@Override	재정의된 메서드라는 정보 제공
@FunctionalInterface	함수형 인터페이스라는 정보 제공
@Deprecated	이후 버전에서 사용되지 않을 수 있는 변수, 메서드에 사용됨
@SuppressWarnings	특정 경고가 나타나지 않도록 함 (예) @SuppressWarnings("deprecation")는 @Deprecated가 나타나지 않도록 함

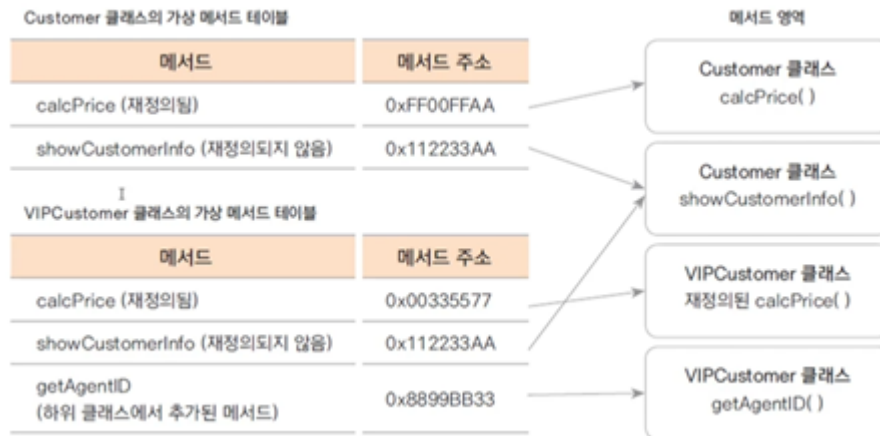
형변환과 오버라이딩 메서드 추출

```
Customer vc = new VIPCustomer();
vc.calcPrice(10000);
```

- 위 코드에서 calcPrice() 메서드는 어느 메서드가 호출 될 것인가?
 - 자바에서는 **항상 인스턴스 (여기서는 VIP Customer)의 메서드가 호출됨**
 - 이를 **가상함수**라고 한다.
 - vc 변수 타입은 Customer지만 인스턴스의 타입은 VIPCustomer이다.

가상 메서드 (virtual method)

- 메서드의 이름과 메서드 주소를 가진 가상 메서드 테이블에서 호출될 메서드의 주소를 참조함
- 메서드에 대한 매핑되는 주소값이 따로 있다. 메서드가 수행되는 수행 코드의 위치가 있는데, 수행코드로 매개변수도 들어가고 명령이 수행되려면 제어가 넘어가야 한다. 함수를 호출한다는 것은, 그 함수 부분으로 제어가 넘어간다는 뜻인데, 그 주소로 넘어가게 되면 그 주소에 대한 부분을 매핑하고 있는 테이블이 있다.



5. 메서드 재정의와 가상 메서드 원리

메서드는 어떻게 호출되고 실행 되는가?

- 메서드(함수)의 이름은 주소값을 나타냄.
 - 따로 주소를 mapping 시키는 것이 아니라, 함수를 호출하면 그 함수 이름에 mapping에 따른 주소가 호출된다.
- 메서드는 명령어의 set 이고 프로그램이 로드되면 **메서드 영역(코드 영역)에 명령어 set이 위치**
- 해당 메서드가 호출 되면 명령어 set 이 있는 주소를 찾아 명령어가 실행됨
- 이때 **메서드에서 사용하는 변수들은 스택 메모리에 위치** 하게됨
- 따라서 다른 인스턴스라도 같은 메서드의 코드는 같으므로 같은 메서드가 호출됨
- 인스턴스가 생성되면 변수는 힙 메모리에 따로 생성되지만, 메서드 명령어 set은 처음 한 번만 로드 됨**

```
public class TestMethod {

    int num;

    void aaa() {
        System.out.println("aaa() 호출");
    }

    public static void main(String[] args) {

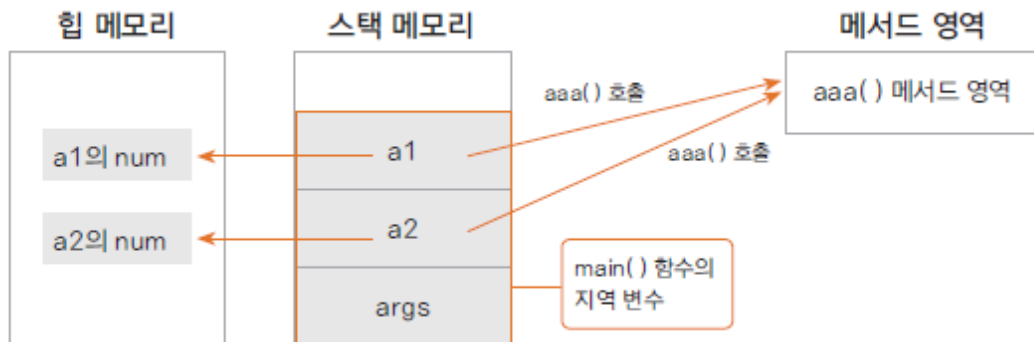
        TestMethod a1 = new TestMethod();
        a1.aaa();

        TestMethod a2 = new TestMethod();
```

```

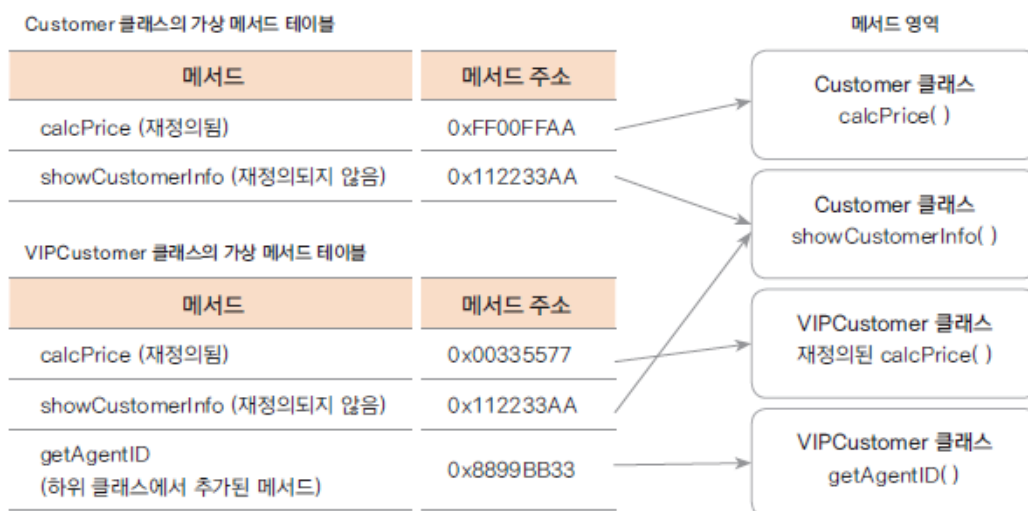
    a2.aaa();
}
}

```



가상 메서드의 원리

- 가상 메서드 테이블(virtual method table)에서 해당 메서드에 대한 address를 가지고 있음
- 재정의된 경우는 재정의 된 메서드의 주소를 가리킴



재정의된 메서드의 호출 과정

- `Customer vc = new VIPCustomer();` 일 때



- 재정의된 경우에 VipCustomer 클래스의 calcPrice가 불린다. 이것이 가상함수이다.

다형성 (polymorphism)

다형성과 다형성을 사용하는 이유

다형성이란

- 하나의 코드가 여러 자료형으로 구현되어 실행되는 것
- 같은 코드에서 여러 실행 결과가 나옴
- 정보은닉, 상속과 더불어 객체지향 프로그래밍의 가장 큰 특징 중 하나
- 다형성을 잘 활용하면 유연하고 확장성 있고, 유지보수가 편리한 프로그램을 만들 수 있음

다형성의 예

```
class Animal{  
  
    public void move() {  
        System.out.println("동물이 움직입니다.");  
    }  
  
    public void eating() {  
  
    }  
}
```

```

class Human extends Animal{
    public void move() {
        System.out.println("사람이 두발로 걷습니다.");
    }

    public void readBooks() {
        System.out.println("사람이 책을 읽습니다.");
    }
}

class Tiger extends Animal{

    public void move() {
        System.out.println("호랑이가 네 발로 뛴니다.");
    }

    public void hunting() {
        System.out.println("호랑이가 사냥을 합니다.");
    }
}

class Eagle extends Animal{
    public void move() {
        System.out.println("독수리가 하늘을 날아갑니다.");
    }

    public void flying() {
        System.out.println("독수리가 날개를 쭉 펴고 멀리 날아갑니다");
    }
}

public class AnimalTest {

    public static void main(String[] args) {

        Animal hAnimal = new Human();
        Animal tAnimal = new Tiger();
        Animal eAnimal = new Eagle();

        AnimalTest test = new AnimalTest();
        test.moveAnimal(hAnimal);
        test.moveAnimal(tAnimal);
        test.moveAnimal(eAnimal);

        ArrayList<Animal> animalList = new ArrayList<Animal>();
        animalList.add(hAnimal);
        animalList.add(tAnimal);
        animalList.add(eAnimal);

        for(Animal animal : animalList) {
            animal.move();
        }
    }
}

```



```
public void moveAnimal(Animal animal) {
    animal.move();

}
}
```

다형성을 사용하는 이유?

- 다른 동물을 추가하는 경우
- 상속과 메서드 재정의의 활용하여 확장성 있는 프로그램을 만들 수 있음
- 그렇지 않는 경우 **if-else if문**이 구현되고 코드의 유지보수가 어려워짐

```
if(customerGrade == "VIP") { //할인해 주고, 적립도 많이 해주고
}
else if(customerGrade == "GOLD") { //할인해 주고, 적립은 적당히
}
else if(customerGrade == "SILVER") { //적립만 해준다
}
```

- 상위클래스에서는 공통적인 부분을 제공하고 하위 클래스에서는 각 클래스에 맞는 기능 구현
- 여러 클래스를 하나의 타입(상위 클래스)으로 핸들링 할 수 있음

다형성을 활용한 멤버십 프로그램 확장

- 일반 고객과 VIP 고객 중간 멤버십 만들기

고객이 늘어 일반 고객보다는 많이 구매하고 VIP보다는 적게 구매하는 고객에게도 혜택을 주기로 했다.

GOLD 고객 등급을 만들고 혜택은 다음과 같다

1. 제품을 살때는 10프로를 할인해준다
2. 보너스 포인트는 2%를 적립해준다

GoldCustomer.java

```

public class GoldCustomer extends Customer{

    double saleRatio;

    public GoldCustomer(int customerID, String customerName){
        super(customerID, customerName);

        customerGrade = "GOLD";
        bonusRatio = 0.02;
        saleRatio = 0.1;

    }

    public int calcPrice(int price){
        bonusPoint += price * bonusRatio;
        return price - (int)(price * saleRatio);
    }
}

```

VIPCustomer.java

```

//showCustomerInfo() 재정의
public String showCustomerInfo(){
    return super.showCustomerInfo() + " 담당 상담원 번호는 " + agentID + "입니다";
}

```

CustomerTest.java

```

public class CustomerTest {

    public static void main(String[] args) {

        ArrayList<Customer> customerList = new ArrayList<Customer>();

        Customer customerLee = new Customer(10010, "이순신");
        Customer customerShin = new Customer(10020, "신사임당");
        Customer customerHong = new GoldCustomer(10030, "홍길동");
        Customer customerYul = new GoldCustomer(10040, "이율곡");
        Customer customerKim = new VIPCustomer(10050, "김유신", 12345);

        customerList.add(customerLee);
        customerList.add(customerShin);
        customerList.add(customerHong);
        customerList.add(customerYul);
        customerList.add(customerKim);

        System.out.println("===== 고객 정보 출력 =====");

        for( Customer customer : customerList){
            System.out.println(customer.showCustomerInfo());
        }
    }
}

```

```

        System.out.println("===== 할인율과 보너스 포인트 계산 =====");

        int price = 10000;
        for( Customer customer : customerList){
            int cost = customer.calcPrice(price);
            System.out.println(customer.getCustomerName() + " 님이 " + cost + "원 지불하셨습니다.");
            System.out.println(customer.getCustomerName() + " 님의 현재 보너스 포인트는 " + customer.bonusPoint + "점입니다.");
        }
    }
}

```

• 결과

```

<terminated> CustomerTest (3) [Java Application] C:\Users\Administrator\p2\pool\plugins\org.eclipse.
===== 고객 정보 출력 =====
이순신 님의 등급은 SILVER이며, 보너스 포인트는 0입니다.
신사임당 님의 등급은 SILVER이며, 보너스 포인트는 0입니다.
홍길동 님의 등급은 GOLD이며, 보너스 포인트는 0입니다.
이율곡 님의 등급은 GOLD이며, 보너스 포인트는 0입니다.
김유신 님의 등급은 VIP이며, 보너스 포인트는 0입니다. 담당 상담원 번호는 12345입니다
===== 할인율과 보너스 포인트 계산 =====
이순신 님이 10000원 지불하셨습니다.
이순신 님의 현재 보너스 포인트는 100점입니다.
신사임당 님이 10000원 지불하셨습니다.
신사임당 님의 현재 보너스 포인트는 100점입니다.
홍길동 님이 9000원 지불하셨습니다.
홍길동 님의 현재 보너스 포인트는 200점입니다.
이율곡 님이 9000원 지불하셨습니다.
이율곡 님의 현재 보너스 포인트는 200점입니다.
김유신 님이 9000원 지불하셨습니다.
김유신 님의 현재 보너스 포인트는 500점입니다.

```

다형성을 사용함으로써 갖는 장점?

- 다양한 여러 클래스를 하나의 자료형 (상위 클래스)으로 선언하거나 형변환하여 각 클래스가 동일한 메서드를 오버라이딩 한 경우, 하나의 코드가 다양한 구현을 실행할 수 있음
- 유사한 클래스가 추가되는 경우, 유지보수에 용이하고 각 자료형마다 다른 메서드를 호출하지 않으므로 코드에서 많은 if문이 사라짐

7. 상속은 언제 사용할까?

- **IS-A 관계 (is a relationship: inheritance)**

- 일반적인 (general) 개념과 구체적인 (specific) 개념과의 관계
- 상위 클래스: 일반적인 개념 클래스 (예: 포유류)
- 하위 클래스: 구체적인 개념 클래스 (예: 사람, 원숭이, 고래...)
- 상속은 클래스간의 결합도가 높은 설계
- 상위 클래스의 수정이 많은 하위 클래스에 영향을 미칠 수 있음
- 계층구조가 복잡하거나 hierarchy가 높으면 좋지 않음

- **HAS-A 관계 (composition)**

- 한 클래스가 다른 클래스를 소유한 관계
- 코드 재사용의 한 방법
- Student가 Subject를 포함한 관계
- Library를 구현할 때 ArrayList 생성하여 사용
- 상속하지 않음

8. 다운 캐스팅과 instanceof

하위클래스로 형 변환, 다운 캐스팅

- 묵시적으로 상위클래스 형변환된 인스턴스가 원래 자료형(하위클래스)으로 변환되어야 할 때 다운캐스팅이라 함
- 하위 클래스로의 형 변환은 명시적으로 되어야 함
- 업캐스팅된 클래스를 다시 원래의 타입으로 형변환

```
Customer vc = new VIPCustomer(); //묵시적
VIPCustomer vCustomer = (VIPCustomer)vc; //명시적
```

instanceof

- 클래스가 정말 속해있는지 확인하며, 속해있다면 true 값을, 그렇지 않다면 false를 반환한다.
- 원래 인스턴스의 형이 맞는지 여부를 체크하는 키워드이다.
 - 맞다면 true 값, 틀리면 false 값을 리턴한다.

예시

AnimalTest.java

```
public void testDownCasting(ArrayList<Animal> list) {

    for(int i =0; i<list.size(); i++) {
        Animal animal = list.get(i);

        if ( animal instanceof Human) {
            Human human = (Human)animal;
            human.readBooks();
        }
        else if( animal instanceof Tiger) {
            Tiger tiger = (Tiger)animal;
            tiger.hunting();
        }
        else if( animal instanceof Eagle) {
            Eagle eagle = (Eagle)animal;
            eagle.flying();
        }
        else {
            System.out.println("error");
        }
    }
}
```

추상 클래스

9. 추상 클래스의 의미와 구현하는 방법

추상 클래스란

- 구현 코드 없이 메서드의 선언만 있는 추상 메서드를 포함한 메서드
- 메서드 선언 (declaration): 반환 타입, 메서드 이름, 매개 변수로 구성

- 메서드 정의 (definition): 메서드 구현 (implementation)과 동일한 의미 구현부 (body)를 가짐 `{ }`
- 예시

```
int add(int x, int y); //선언
int add(int x, int y){}; //구현부가 있음, 추상 메서드가 아님
```

- abstract 예약어 사용
- 추상 클래스는 new (인스턴스화) 할 수 없음
 - method에 body가 없으니까 불러질 수 없기 때문이다.
- cf) concrete class

주의하기

```
public void display() {}
```

- 위 코드는 {} 가 있으므로 body 가 있다고 판단한다.
- 그렇기 때문에 추상 메서드라고 할 수 없다.

예시

```
package Chapter8.abstractex;

public abstract class Computer {

    public abstract void display();
    public abstract void typing();

    public void turnOn() {
        System.out.println("전원을 켭니다.");
    }

    public void turnOff() {
        System.out.println("전원을 끕니다.");
    }
}
```

추상 클래스 구현하기

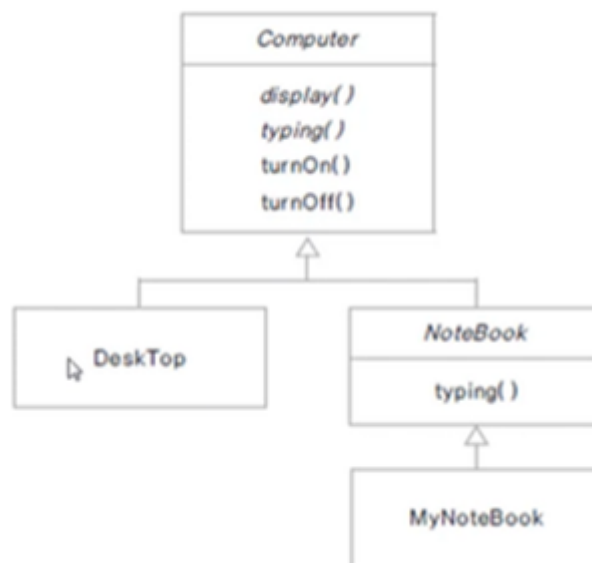
- 메서드와 클래스에 `abstract` 키워드 사용
- 추상 메서드가 포함된 클래스는 추상 클래스로 선언
- 모든 메서드가 구현되었다고 해도 클래스에 `abstract` 키워드를 사용하면 추상 클래스
- box를 class로 표현
- Computer - 클래스 이름
- display, typing, turnOn, turnOff - method
- 이탤릭체 → 추상메서드 및 클래스

```
package abstractex;

public abstract class Computer {
    public abstract void display( );
    public abstract void typing( );
    ...
}
```

abstract 예약어 추가

더 이상 오류 없음



- Computer, Notebook은 클래스 이름이다.
- 박스 자체는 클래스를 뜻한다.
- italic체 (클래스 이름, 메소드) → 추상 클래스 및 추상 메서드를 뜻한다.
- DeskTop과 Notebook은 상속을 받은 것이다.

Computer.java

```
public abstract class Computer {

    //현재는 구현하지 않을테니 상속받는 클래스들이 구현하라!
    abstract void display();
    abstract void typing();

    public void turnOn() {
        System.out.println("전원을 켭니다.");
    }

    public void turnOff() {
        System.out.println("전원을 끕니다.");
    }
}
```

DeskTop.java

```
public class DeskTop extends Computer{

    @Override
    public void display() {
        System.out.println("DeskTop display");
    }

    @Override
    public void typing() {
        System.out.println("DeskTop typing");
    }

    @Override
    public void turnOff() {
        System.out.println("Desktop turnoff");
    }
}
```

NoteBook.java

```
public abstract class NoteBook extends Computer{
    @Override
    public void typing() {
        System.out.println("NoteBook typing");
    }
}
```

- 아직도 구현 안한 메소드 (typing)가 있으므로 **abstract** 클래스로 선언한다.

MyNoteBook.java


```
public class MyNoteBook extends Notebook{

    @Override
    public void display() {
        System.out.println("MyNoteBook display");
    }
}
```

ComputerTest.java

```
public class ComputerTest {

    public static void main(String[] args) {
        Computer computer = new DeskTop();
        computer.display();
        computer.turnOff();

        Notebook myNote = new MyNoteBook();
    }
}
```

추상클래스 사용하기

- 추상 클래스는 주로 상속의 상위클래스로 사용됨
 - 추상 메서드: 하위 클래스가 구현해야 하는 메서드
 - 구현된 메서드: 하위 클래스가 공통으로 사용하는 기능의 메서드 + 하위 클래스에 따라 재정의할 수 있음

10. 추상 클래스를 활용한 템플릿 메서드 패턴

템플릿 메서드

프레임워크를 설계를 할 때 자주 사용한다.

- 템플릿: 틀이나 견본을 의미
- template method
 - 추상 메서드나 구현된 메서드를 활용하여 전체의 흐름 (시나리오)를 정의해놓은 메서드

- 클래스를 **final**로 선언하여 하위 클래스에서 재정의 할 수 없게 함
- 추상 클래스로 선언된 상위 클래스에서 템플릿 메서드를 활용하여 전체적인 흐름을 정의하고, 하위 클래스에서 다르게 구현되어야 하는 부분은 추상 메서드로 선언하여 하위 클래스에서 구현하도록 함
- template method pattern: 디자인 패턴의 일종
 - 프레임워크에서 많이 사용되는 설계 패턴
 - 추상 클래스로 선언된 상위 클래스에서 추상 메서드를 이용하여 전체 구현의 흐름을 정의하고, 구체적인 각 메서드 구현은 하위 클래스에 위임함
 - 하위 클래스가 다른 구현을 했다고 해서 템플릿 메서드에 정의된 시나리오 대로 수행됨

템플릿 메서드 구현하기 예제



```

public abstract class Car {
    public abstract void drive( );
    public abstract void stop( );

    public void startCar( ) {
        System.out.println("시동을 켭니다.");
    }

    public void turnOff( ) {
        System.out.println("시동을 끕니다.");
    }

    final public void run( ) {
        startCar( );
        drive( );
        stop( );
        turnOff( );
    }
}
  
```

템플릿 메서드

Car.java

```

public abstract class Car {

    public abstract void drive();
    public abstract void stop();

    public void startCar() {
        System.out.println("시동을 켭니다.");
    }

    public void turnOff() {
  
```

```

        System.out.println("시동을 끕니다.");
    }

    public void washCar(){} //구현부가 있는데 statement가 없음 그래서 추상 메서드가 아니다. -> 필요한 경우에 재정의의를 한다.

    final public void run() {
        startCar();
        drive();
        washCar();
        stop();
        turnOff();
    }
}

```

ManualCar.java

```

public class ManualCar extends Car{

    @Override
    public void drive() {
        System.out.println("사람이 운전합니다.");
        System.out.println("사람이 핸들을 조작합니다.");
    }

    @Override
    public void stop() {
        System.out.println("브레이크를 밟아서 정지합니다.");
    }

}

```

AICar.java

```

public class AICar extends Car{

    @Override
    public void drive() {
        System.out.println("자율 주행합니다.");
        System.out.println("자동차가 스스로 방향을 바꿉니다.");
    }

    @Override
    public void stop() {
        System.out.println("스스로 멈춥니다.");
    }

    @Override
    public void washCar() {
        System.out.println("자동 세차를 합니다.");
    }

}

```

CarTest.java

```
public class CarTest {  
  
    public static void main(String[] args) {  
        Car aiCar = new AICar();  
        aiCar.run();  
        System.out.println("=====");  
        Car manualCar = new ManualCar();  
        manualCar.run();  
    }  
}
```

final 예약어

- final 변수는 값이 변경될 수 없는 상수임
 - `public static final double PI = 3.14;`
- 오직 한 번만 값을 할당할 수 있음
- final 메서드는 하위 클래스에서 재정의 (overriding) 할 수 없음
- final 클래스는 더 이상 상속되지 않음
 - 예) java의 String 클래스

public static final 상수값 정의하여 사용하기

- 프로젝트 구현 시 여러 파일에서 공유해야 하는 상수 값은 하나의 파일에 선언하여 사용하면 편리함

```
public class Define {
    public static final int MIN = 1;
    public static final int MAX = 99999;
    public static final int ENG = 1001;
    public static final int MATH = 2001;
    public static final double PI = 3.14;
    public static final String GOOD_MORNING = "Good Morning!";
}
```

```
public class UsingDefine {
    public static void main(String[] args) {
        System.out.println(Define.GOOD_MORNING);
        System.out.println("최솟값은 " + Define.MIN + "입니다.");
        System.out.println("최댓값은 " + Define.MAX + "입니다.");
        System.out.println("수학 과목 코드 값은 " + Define.MATH + "입니다.");
        System.out.println("영어 과목 코드 값은 " + Define.ENG + "입니다.");
    }
}
```

static으로 선언했으므로 인스턴스를 생성
하지 않고 클래스 이름으로 참조 가능

Define.java

```
public class Define {

    public static final int MIN = 1;
    public static final int MAX = 999999;
    public static final double PI = 3.14;
    public static final String GREETING = "Good Morning!";
    public static final int MATH_CODE = 1001;
    public static final int CHEMISTRY_CODE = 1002;

}
```

UsingDefine.java

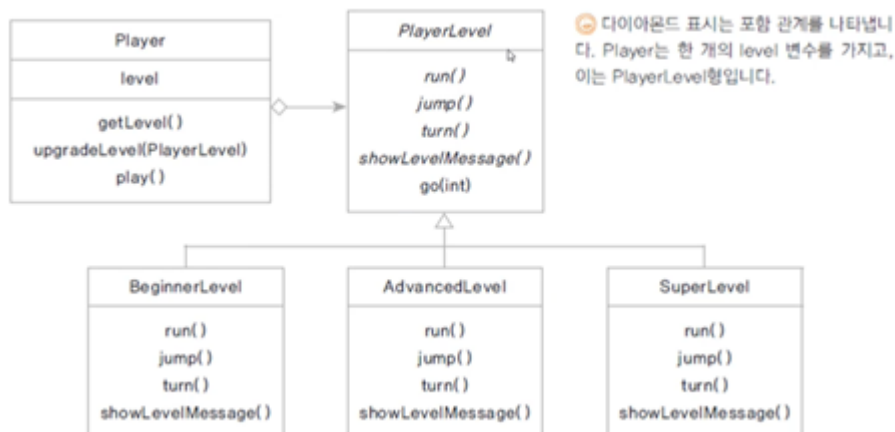
```
public class UsingDefine {

    public static void main(String[] args) {

        System.out.println(Define.GREETING);
        System.out.println(Define.MIN);
        System.out.println(Define.MAX);
        System.out.println(Define.MATH_CODE);
        System.out.println(Define.CHEMISTRY_CODE);
        System.out.println("원주율은" + Define.PI + "입니다.");
    }

}
```

템플릿 메서드 활용하기 (예제)



인터페이스

11. 구현 코드가 없는 인터페이스

인터페이스란

- 모두 메소드가 abstract method로 이루어져 있음 (public abstract)
- 인터페이스의 모든 변수는 상수로 선언됨 (public static final)

```
interface 인터페이스 이름{

    public static final float pi = 3.14F;
    public void makeSomething();
}
```

- 자바 8 부터 디폴트 메서드(default method)와 정적 메서드(static method) 기능의 제공으로 일부 구현 코드가 있음

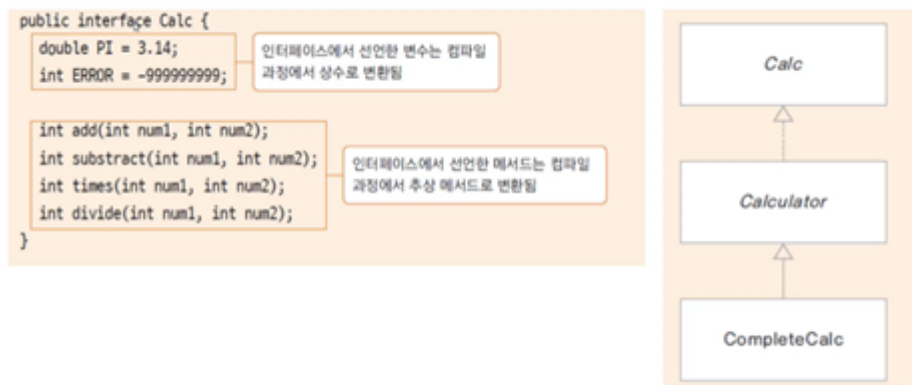
- 인터페이스란 어떤 object에 대한 명제이다. 이 객체가 어떤 method를 제공하고, 어떤 역할을 제공하는지 보여준다.

인터페이스의 요소

- 추상 메서드
- 상수
- 디폴트 메서드
- 정적 메서드
- private 메서드

인터페이스 선언과 구현

- class가 아닌 interface를 붙인다.
- abstract라고 method에 붙이지 않아도, 컴파일 과정에서 추상 메서드로 변환된다.
- 인터페이스는 점선으로 많이 표현한다.



예시

Calc.java

```

public interface Calc {

    double PI = 3.14;
    int ERROR = -999999999;

    int add(int num1, int num2);
    int subtract(int num1, int num2);
    int times(int num1, int num2);
    int divide(int num1, int num2);

}

```

Calculator.java

```
public abstract class Calculator implements Calc{

    @Override
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    @Override
    public int subtract(int num1, int num2) {
        return num1 - num2;
    }
}
```

- times와 divide 와 같이 구현하지 않은 메서드를 포함하고 있으므로, abstract class 로 선언해야 한다.

CompleteCalc.java

```
public class CompleteCalc extends Calculator{

    @Override
    public int times(int num1, int num2) {
        return num1 * num2;
    }

    @Override
    public int divide(int num1, int num2) {
        if( num2 == 0 )
            return ERROR;
        else
            return num1 / num2;
    }

    public void showInfo() {
        System.out.println("모두 구현하였습니다.");
    }
}
```

CalculatorTest.java

```
public class CalculatorTest {

    public static void main(String[] args) {
        Calc calc = new CompleteCalc();
        int num1 = 10;
        int num2 = 2;

        System.out.println(num1 + "+" + num2 + "=" + calc.add(num1, num2));
        System.out.println(num1 + "-" + num2 + "=" + calc.subtract(num1, num2));
    }
}
```



```

        System.out.println(num1 + "*" + num2 + "=" + calc.times(num1, num2));
        System.out.println(num1 + "/" + num2 + "=" + calc.divide(num1, num2));
    }
}

```

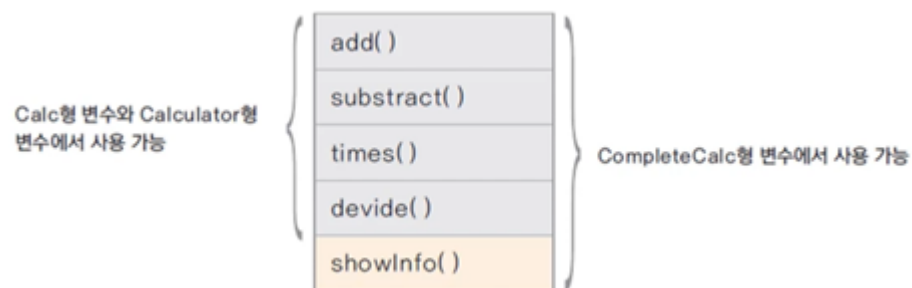
```

<terminated> CalculatorTest [Java Application] C:\Users\#Addr
10+2=12
10-2=8
10*2=20
10/2=5

```

타입 상속과 형 변환

- 인터페이스를 구현한 클래스는 인터페이스 타입으로 변수를 선언하여 형 변환 할 수 있음
 - `Calc calc = new CompleteCalc();`
- 상속에서의 형 변환과 동일한 의미
- 클래스 상속과 달리 인터페이스는 구현 코드가 없기 때문에 여러 인터페이스를 구현할 수 있음 (cf extends). → 이를 타입 상속이라고도 함
 - 형 변환 되는 경우, 인터페이스에 선언된 메서드만을 사용 가능 함.



12. 인터페이스는 왜 쓰는가?

인터페이스를 활용한 다형성 구현

- 인터페이스는 보통 설계할 때 사용한다.
- 인터페이스가 정의되어 있고, 이를 구현하는 클래스를 활용한다.

인터페이스의 역할은? 인터페이스가 하는 일

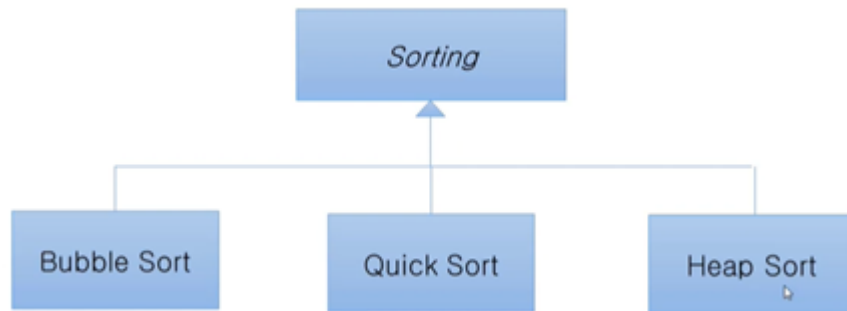
- 클래스나 프로그램이 제공하는 기능을 명시적으로 선언
- 인터페이스는 클라이언트 프로그램에서 어떤 메서드를 제공하는지 알려주는 명세 (specification) 또는 약속
- 한 객체가 어떤 인터페이스의 타입이라 함은 그 인터페이스의 메서드를 구현했다는 의미
- 클라이언트 프로그램은 실제 구현 내용을 몰라도 인터페이스의 정의만 알면서 그 객체를 사용할 수 있음
- 인터페이스를 구현해 놓은 다양한 객체를 사용함 - 다형성
 - JDBC를 구현한 오라클, MYSQL, MS-SQL 라이브러리 등

13. 프로그램에서 인터페이스의 역할과 다형성

인터페이스와 다형성 구현하기

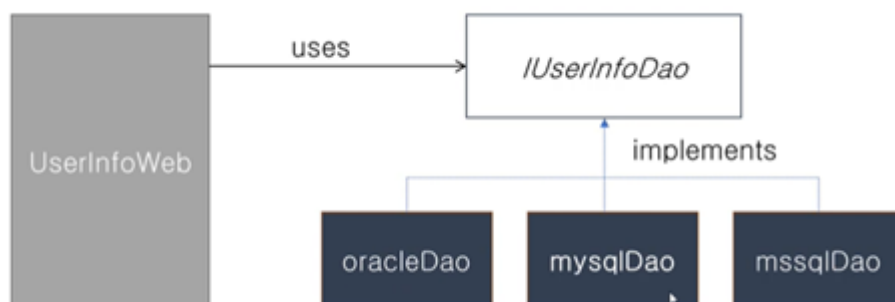
- 고객센터에 전화 상담을 하는 상담원 있음. 일단 고객 센터로 전화가 오면 대기열에 저장. 상담원이 지정되기 전까지 대기 상태가 된다. 각 전화가 상담원에게 배분되는 정책은 다음과 같이 여러 방식으로 구현될 수 있습니다.
 - 상담원 순서대로 배분
 - 대기가 짧은 상담원 먼저 배분하기
 - 우선순위가 높은 (숙련도가 높은) 상담원에게 먼저 배분하기
- 위와 같은 다양한 정책이 사용되는 경우, interface를 정의하고 다양한 정책을 구현하여 실행하세요.

인터페이스와 strategy pattern



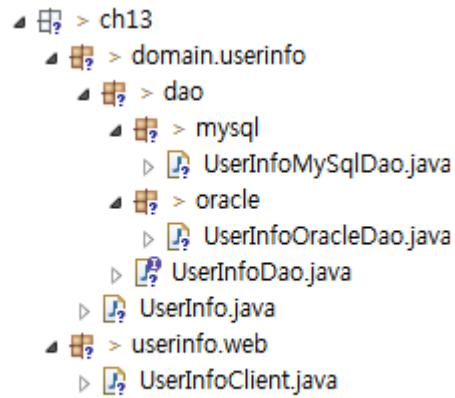
- 인터페이스를 활용하면 다양한 정책이나 알고리즘을 프로그램의 큰 수정 없이 적용, 확장할 수 있음

인터페이스의 사용 예



인터페이스를 활용한 dao 구현하기

- DB에 회원 정보를 넣는 dao(data access object)를 여러 DB 제품이 지원될 수 있게 구현함
- 환경파일(db.properties) 에서 database의 종류에 대한 정보를 읽고 그 정보에 맞게 dao 인스턴스를 생성하여 실행될 수 있게 함
- source hierachy



예시

UserInfo.java (사용자 정보 클래스)

```
public class UserInfo {

    private String userId;
    private String passwd;
    private String userName;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public String getPasswd() {
        return passwd;
    }

    public void setPasswd(String passwd) {
        this.passwd = passwd;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

UserInfoDao.java (dao 에서 제공되어야 할 메서드를 선언한 인터페이스)

```
public interface UserInfoDao {

    void insertUserInfo(UserInfo userInfo);
    void updateUserInfo(UserInfo userInfo);
    void deleteUserInf(UserInfo userInfo);
}
```

UserInfoMySqlDao.java (UserInfoDao 인터페이스를 구현한 MySql 버전 dao)

```
public class UserInfoMySqlDao implements UserInfoDao{

    @Override
    public void insertUserInfo(UserInfo userInfo) {
        System.out.println("insert into MYSQL DB userId =" + userInfo.getUserId() );
    }

    @Override
    public void updateUserInfo(UserInfo userInfo) {
        System.out.println("update into MYSQL DB userId = " + userInfo.getUserId());
    }

    @Override
    public void deleteUserInf(UserInfo userInfo) {
        System.out.println("delete from MYSQL DB userId = " + userInfo.getUserId());
    }

}
```

UserInfoOracleDao.java (UserInfoDao 인터페이스를 구현한 Oracle 버전 dao)

```
public class UserInfoOracleDao implements UserInfoDao{

    public void insertUserInfo(UserInfo userInfo){
        System.out.println("insert into ORACLE DB userId =" + userInfo.getUserId() );
    }

    public void updateUserInfo(UserInfo userInfo){
        System.out.println("update into ORACLE DB userId = " + userInfo.getUserId());
    }

    public void deleteUserInf(UserInfo userInfo){
        System.out.println("delete from ORACLE DB userId = " + userInfo.getUserId());
    }

}
```

UserInfoClient.java (UserInfoDao 인터페이스를 활용하는 클라이언트 프로그램)

```

public class UserInfoClient {

    public static void main(String[] args) throws IOException {

        FileInputStream fis = new FileInputStream("db.properties");

        Properties prop = new Properties();
        prop.load(fis);

        String dbType = prop.getProperty("DBTYPE");

        UserInfo userInfo = new UserInfo();
        userInfo.setUserId("12345");
        userInfo.setPasswd("!@#$%");
        userInfo.setUserName("이순신");

        UserInfoDao userInfoDao = null;

        if(dbType.equals("ORACLE")){
            userInfoDao = new UserInfoOracleDao();
        }
        else if(dbType.endsWith("MYSQL")){
            userInfoDao = new UserInfoMySqlDao();
        }
        else{
            System.out.println("db support error");
            return;
        }

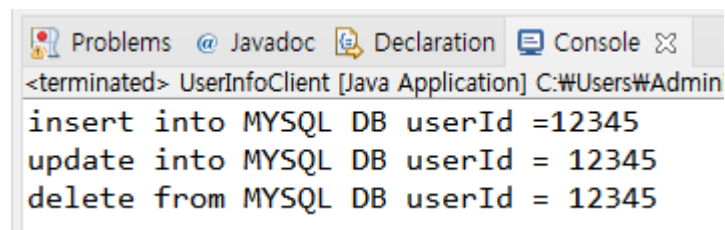
        userInfoDao.insertUserInfo(userInfo);
        userInfoDao.updateUserInfo(userInfo);
        userInfoDao.deleteUserInf(userInfo);
    }
}

```

db.properties 환경파일이 MYSQL 일때

DBTYPE=MYSQL

실행결과



```

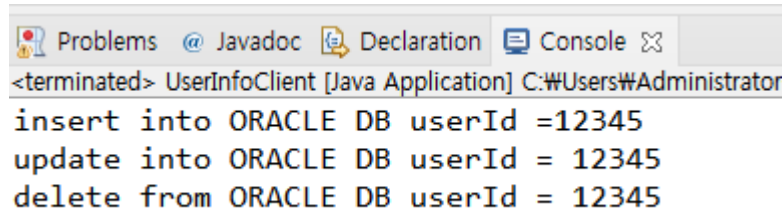
<terminated> UserInfoClient [Java Application] C:\Users\Admin\...
insert into MYSQL DB userId =12345
update into MYSQL DB userId = 12345
delete from MYSQL DB userId = 12345

```

db.properties 환경파일이 ORACLE 일때

```
DBTYPE=ORACLE
```

실행결과



```
<terminated> UserInfoClient [Java Application] C:\Users\Administrator
insert into ORACLE DB userId =12345
update into ORACLE DB userId = 12345
delete from ORACLE DB userId = 12345
```

14. 인터페이스의 요소들

• 상수

- 선언된 모든 변수는 상수로 처리 됨 public static final

```
double PI = 3.14;
int ERROR = -999999999;
```

• 메서드

- 모든 메서드는 추상 메서드

• 디폴트 메서드

- 기본 구현을 가지는 메서드로 구현하는 클래스에서 재정의 할 수 있음 (java 8)
- 구현을 가지는 메서드, 인터페이스를 구현하는 클래스들에서 공통으로 사용할 수 있는 기본 메서드
- default 키워드 사용

```
default void description() {
    System.out.println("정수 계산기를 구현합니다.");
    myMethod();
}
```

- 구현 하는 클래스에서 재정의 할 수 있음

```
@Override
public void description() {
    System.out.println("CompleteCalc에서 재정의한 default 메서드");
    //super.description();
}
```

- 인터페이스를 구현한 클래스의 인스턴스가 생성 되어야 사용 가능함
- **정적 메서드**
 - 인스턴스 생성과 상관없이 **인터페이스 타입으로 호출하는 메서드** (java 8)

```
static int total(int[] arr) {
    int total = 0;

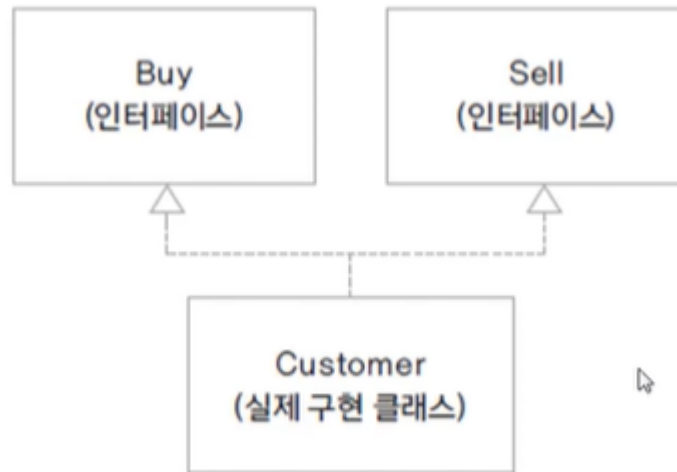
    for(int i: arr) {
        total += i;
    }
    mystaticMethod();
    return total;
}
```

- **private 메서드**
 - 인터페이스 내에서 사용하기 위해 구현한 메서드로 구현하는 클래스에서 재정의 할 수 없음 (java 9)
 - 인터페이스 내부에서만 사용하기 위해 구현하는 메서드
 - default 메서드나 static 메서드에서 사용함

```
private void myMethod() {
    System.out.println("private method");
}

private static void mystaticMethod() {
    System.out.println("private static method");
}
```

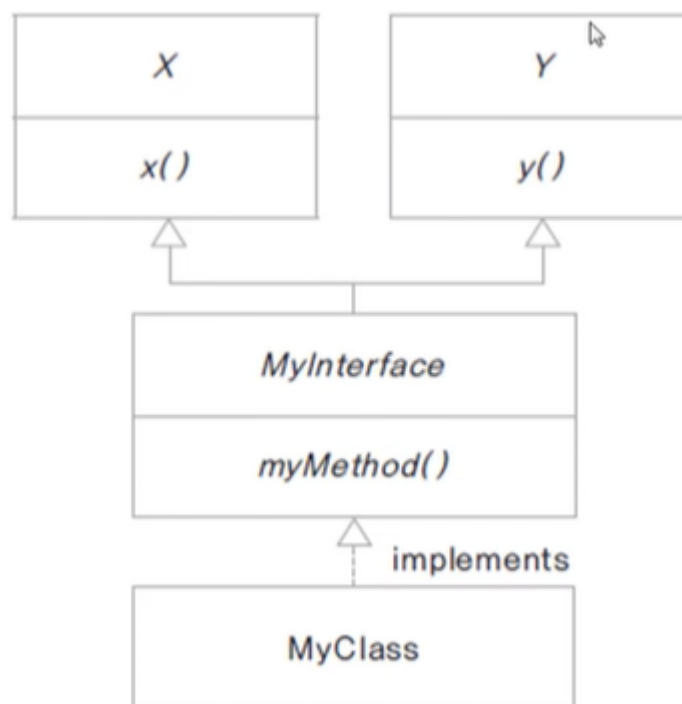
여러 개의 인터페이스 구현하기



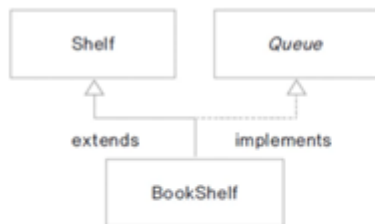
- 인터페이스는 구현 코드가 없으므로 **하나의 클래스가 여러 인터페이스를 구현할 수 있음**
- 디폴트 메서드의 이름이 중복되는 경우에는 재정의 함 - overriding을 통해 재정의

인터페이스 상속

- 인터페이스 간에도 상속이 가능함
- 구현이 없으므로 extends 뒤에 여러 인터페이스를 상속받을 수 있음
- 구현 내용이 없으므로 타입 상속 (type inheritance)라고 함



인터페이스 구현과 클래스 상속 함께 사용하기



```
public class BookShelf extends Shelf implements Queue {
    @Override
    public void enqueue(String title) {
        shelf.add(title);
    }
    @Override
    public String dequeue() {
        return shelf.remove(0);
    }
    @Override
    public int getCount() {
        return getCount();
    }
}
```

Annotations in the code are linked to Korean descriptions:

- `enqueue`: 배열에 요소 추가
- `dequeue`: 맨 처음 요소를 배열에서 삭제하고 반환
- `getCount`: 배열 요소 개수 반환

- Shelf Class

```
package Chapter9.bookshelf;

import java.util.ArrayList;

public class Shelf {
    protected ArrayList<String> shelf;

    public Shelf() {
        shelf = new ArrayList<String>();
    }

    public ArrayList<String> getShelf(){
        return shelf;
    }

    public int getCount() {
        return shelf.size();
    }
}
```

- Queue Interface

```
package Chapter9.bookshelf;
```

```
public interface Queue {
    void enqueue(String title); //집어넣는 method)
    String dequeue();

    int getSize();
}
```

- Bookshelf class

```
package Chapter9.bookshelf;

public class BookShelf extends Shelf implements Queue{
    @Override
    public void enqueue(String title) {
        shelf.add(title);
    }

    @Override
    public String dequeue() {
        return shelf.remove(0);
    }

    @Override
    public int getSize() {
        return getCount();
    }
}
```

- BookshelfTest

```
package Chapter9.bookshelf;

public class BookShelfTest {
    public static void main(String[] args) {
        Queue bookQueue = new BookShelf();

        bookQueue.enqueue("태백산맥1");
        bookQueue.enqueue("태백산맥2");
        bookQueue.enqueue("태백산맥3");

        System.out.println(bookQueue.dequeue());
        System.out.println(bookQueue.dequeue());
        System.out.println(bookQueue.dequeue());
    }
}
```

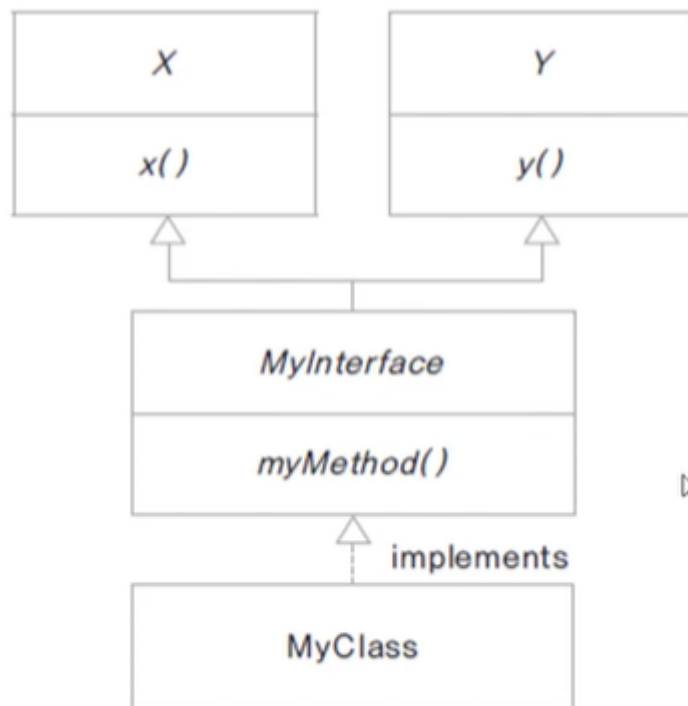
15. 여러 인터페이스 구현하기, 인터페이스의 상속

여러 개의 인터페이스 구현하기

- 인터페이스는 구현 코드가 없으므로 하나의 클래스가 여러 인터페이스를 구현할 수 있음
- 디폴트 메서드의 이름이 중복 되는 경우에는 재정의함

인터페이스 상속

- 인터페이스 간에도 상속이 가능함
- 구현이 없으므로 extends 뒤에 여러 인터페이스를 상속받을 수 있음
- 구현 내용이 없으므로 타입 상속 (type inheritance)라고 함



- 가상 메서드 테이블(virtual method table)에서 해당 메서드에 대한 address를 가지고 있음
- 재정의된 경우는 재정의 된 메서드의 주소를 가리킴