

다형성 (Polymorphism)

▶ 다형성

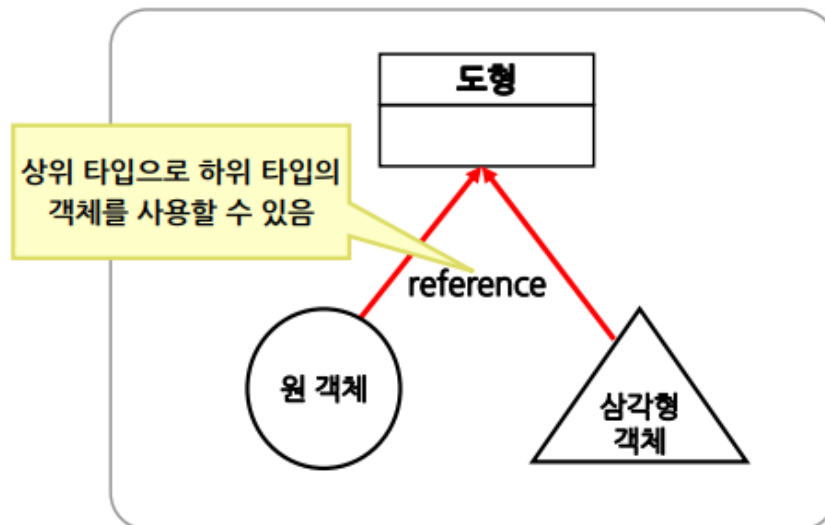
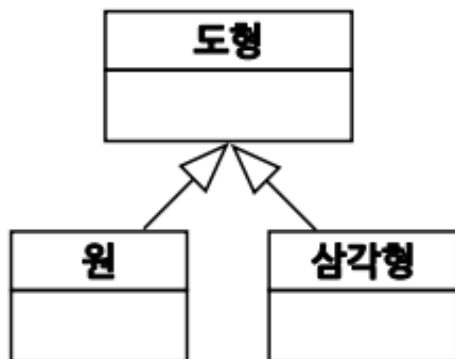
객체지향 프로그래밍의 3대 특징 중 하나

'여러 개의 형태를 갖는다'는 의미로

하나의 행동으로 여러 가지 일을 수행하는 개념

상속을 이용해 부모 타입으로부터 파생된 여러 가지 타입의 자식 객체를

부모 클래스 타입 하나로 다룰 수 있는 기술



▶ 클래스 형변환

✓ 업 캐스팅(Up Casting)

상속 관계에 있는 부모, 자식 클래스 간에 부모타입의 참조형 변수가 모든 자식 타입의 객체 주소를 받을 수 있음

```
//Sonata 클래스는 Car 클래스의 후손  
Car c = new Sonata();  
//Sonata클래스형에서 Car클래스형으로 바뀜
```

- * 자식 객체의 주소를 전달받은 부모타입의 참조변수를 통해서 사용할 수 있는 후손의 정보는 원래 부모타입이었던 멤버만 참조 가능

▶ 클래스 형변환

✓ 다운 캐스팅(Down Casting)

자식 객체의 주소를 받은 부모 참조형 변수를 가지고 자식의 멤버를 참조해야 할 경우, 부모 클래스 타입의 참조형 변수를 자식 클래스 타입으로 형 변환하는 것

자동으로 처리되지 않기 때문에 반드시 후손 타입 명시해서 형 변환

```
//Sonata 클래스는 Car 클래스의 후손  
Car c = new Sonata();  
((Sonata)c).moveSonata();
```

* 클래스 간의 형 변환은 반드시 상속 관계에 있는 클래스끼리만 가능

▶ 객체배열과 다형성

다형성을 이용하여 상속 관계에 있는 하나의 부모 클래스 타입의 배열 공간에 여러 종류의 자식 클래스 객체 저장 가능

```
Car[] carArr = new Car[5];
```

```
carArr[0] = new Sonata();
```

```
carArr[1] = new Avante();
```

```
carArr[2] = new Grandure();
```

```
carArr[3] = new Spark();
```

```
carArr[4] = new Morning();
```

▶ 매개변수와 다형성

다형성을 이용하여 메소드 호출 시 부모타입의 변수 하나만 사용해 자식 타입의 객체를 받을 수 있음

```
public void execute() {  
    driveCar(new Sonata());  
    driveCar(new Avante());  
    driveCar(new Grandure());  
}
```

```
public void driveCar(Car c) {}
```

▶ instanceof 연산자

현재 참조형 변수가 어떤 클래스 형의 객체 주소를 참조하고 있는지 확인 할 때 사용하는 연산자로 클래스 타입이 맞으면 true, 맞지 않으면 false 반환

✓ 표현식

```
if(레퍼런스 instanceof 클래스타입) {  
    //true일때 처리할 내용, 해당 클래스 타입으로 down casting  
}
```

```
if(c instanceof Sonata) {  
    ((Sonata)c).moveSonata();  
} else if (c instanceof Avante){  
    ((Avante)c).moveAvante();  
} else if (c instanceof Grandure){  
    ((Grandure)c).moveGrandure();  
}
```

▶ 바인딩

실제 실행할 메소드 코드와 호출하는 코드를 연결 시키는 것
프로그램이 실행되기 전에 컴파일이 되면서 모든 메소드는 정적 바인딩 됨

✓ 동적 바인딩

컴파일 시 정적 바인딩된 메소드를
실행할 당시의 객체 타입을 기준으로 바인딩 되는 것

✓ 동적 바인딩 성립 요건

상속 관계로 이루어져 다형성이 적용된 경우, 메소드 오버라이딩이
되어 있으면 정적으로 바인딩 된 메소드 코드보다
오버라이딩 된 메소드 코드를 우선적으로 수행

▶ 추상 클래스

✓ 추상 클래스(abstract class)

몸체 없는 메소드를 포함한 클래스

추상 클래스일 경우 클래스 선언부에 abstract 키워드 사용

[접근제한자] abstract class 클래스명 {}

✓ 추상 메소드(abstract method)

몸체 없는 메소드

추상 메소드의 선언부에 abstract 키워드 사용

상속 시 반드시 구현해야 하는, 오버라이딩이 강제화되는 메소드

[접근제한자] abstract 반환형 메소드명(자료형 변수명);

▶ 추상 클래스

✓ 특징

1. 미완성 클래스(`abstract` 키워드 사용)

자체적으로 객체 생성 불가 → 반드시 상속하여 객체 생성

2. `abstract` 메소드가 포함된 클래스는 반드시 `abstract` 클래스

`abstract` 메소드가 없어도 `abstract` 클래스 선언 가능

3. 클래스 내에 일반 변수, 메소드 포함 가능

4. 객체 생성은 안되지만 참조형 변수 타입으로는 사용 가능

✓ 장점

일관된 인터페이스 제공

꼭 필요한 기능 강제화(공통적이거나 자식클래스에서 특수화 되는 기능)

▶ 인터페이스

상수형 필드와 추상 메소드만을 작성할 수 있는 추상 클래스의 변형체
메소드 통일성을 부여하기 위해 추상 메소드만 따로 모아놓은 것으로
상속 시 인터페이스 내에 정의된 모든 추상메소드 구현해야 함

[접근제한자] **interface** 인터페이스명 {

// 상수도 멤버로 포함할 수 있음

public static final 자료형 변수명 = 초기값;

// 추상 메소드만 선언 가능

[public abstract] 반환자료형 메소드명([자료형 매개변수]);

// public abstract가 생략되기 때문에

// 오버라이딩 시 반드시 public 표기해야 함

}

▶ 인터페이스

✓ 특징

1. 모든 인터페이스의 메소드는 묵시적으로 public이고 abstract
2. 변수는 묵시적으로 public static final,
따라서 인터페이스 변수의 값 변경 시도 시 컴파일 시 에러 발생
3. 객체 생성은 안되나 참조형 변수로는 가능

✓ 장점

- 상위 타입 역할로 다형성을 지원하여 연결
- 해당 객체가 다양한 기능 제공 시에도 인터페이스에 해당하는 기능만을 사용하게 제한 가능
- 공통 기능 상의 일관성 제공
- 공동 작업을 위한 인터페이스 제공

▶ 추상클래스와 인터페이스

| 구분 | 추상 클래스 | 인터페이스 |
|----------|--------------------|------------------|
| 상속 | 단일 상속 | 다중 상속 |
| 구현 | extends 사용 | implements 사용 |
| 추상 메소드 | abstract 메소드 0개 이상 | 모든 메소드는 abstract |
| abstract | 명시적 사용 | 묵시적으로 abstract |
| 객체 | 객체 생성 불가 | 객체 생성 불가 |
| 용도 | 참조 타입 | 참조 타입 |