Java의 정석

Chapter 07

1. 상속
1.1 상속의 정의와 장점
1.2 클래스간의 관계 – 포함관계
1.3 클래스간의 관계 결정하기

2. 오버라이딩
2.1 오버라이딩이란?
2.2 오버라이딩의 조건
2.3 오버로딩 vs 오버라이딩
2.4 super

3. Package와 import
3.1 패키지
3.2 패키지의 선언
3.3 import문
3.4 import문의 선언
3.5 static import문

4. 제어자
4.1 제어자란?
4.2 static – 클래스의, 공통적인
4.3 final – 마지막의, 변경될 수 없는
4.4 abstract – 추상의, 미완성의
4.5 접근 제어자
4.6. 제어자의 조합

5. 다형성
5.1 다형성이란?
5.2 참조변수의 형변환
5.3 inctanceof 연산자
5.5 매개변수의 다형성
5.6 여러 종류의 객체를 배열로 다루기

6. 추상클래스
6.1 추상클래스란?
6.2 추상메서드
6.3 추상클래스의 작성

7. 인터페이스
7.1 인터페이스란?
7.2 인터페이스의 작성
7.3 인터페이스의 상속
7.4 인터페이스의 구현

심화. 디자인패턴
- 디자인 패턴이란
- 디자인 패턴의 종류

상속

1.1상속의 정의와 장점

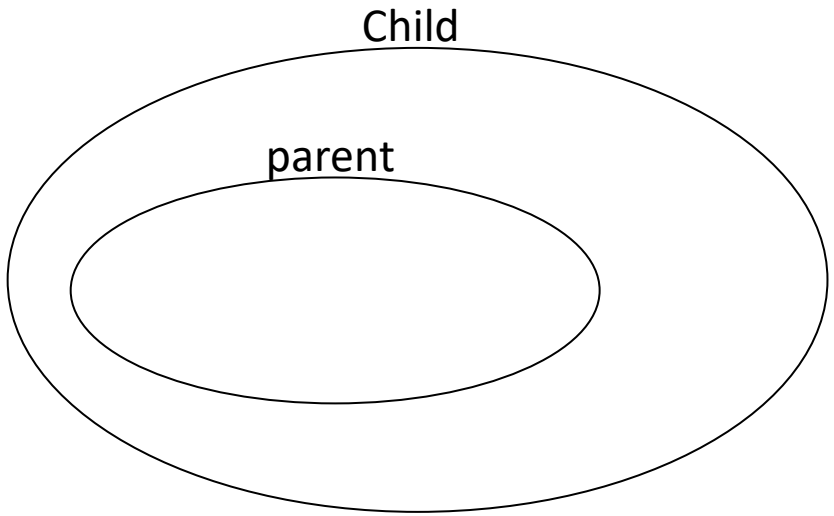
□상속이란

-기존 클래스를 재사용 하여 새로운 클래스를 작성 하는 것
- 보다 적은 양의 코드로 새로운 클래스 작성 가능
- 코드를 공통적으로 관리할 수 있어 유지/보수 용이

```
Class child extend parent{
    ...
}
```

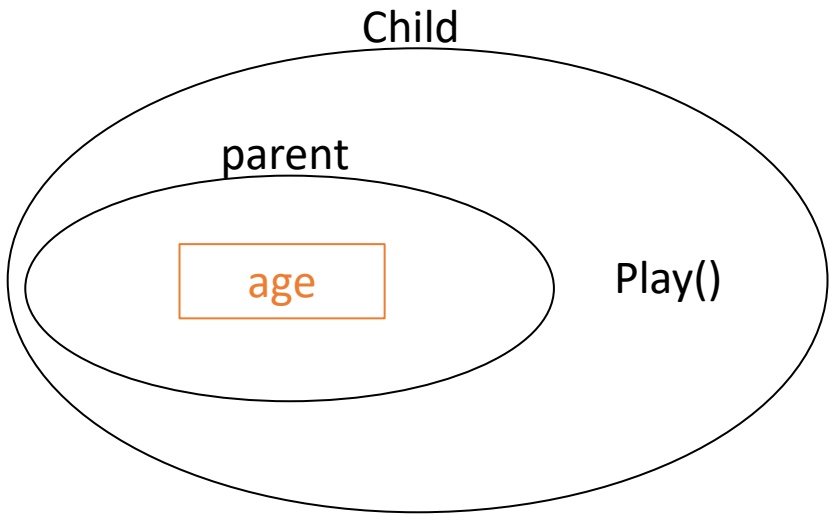
자손 클래스

조상 클래스

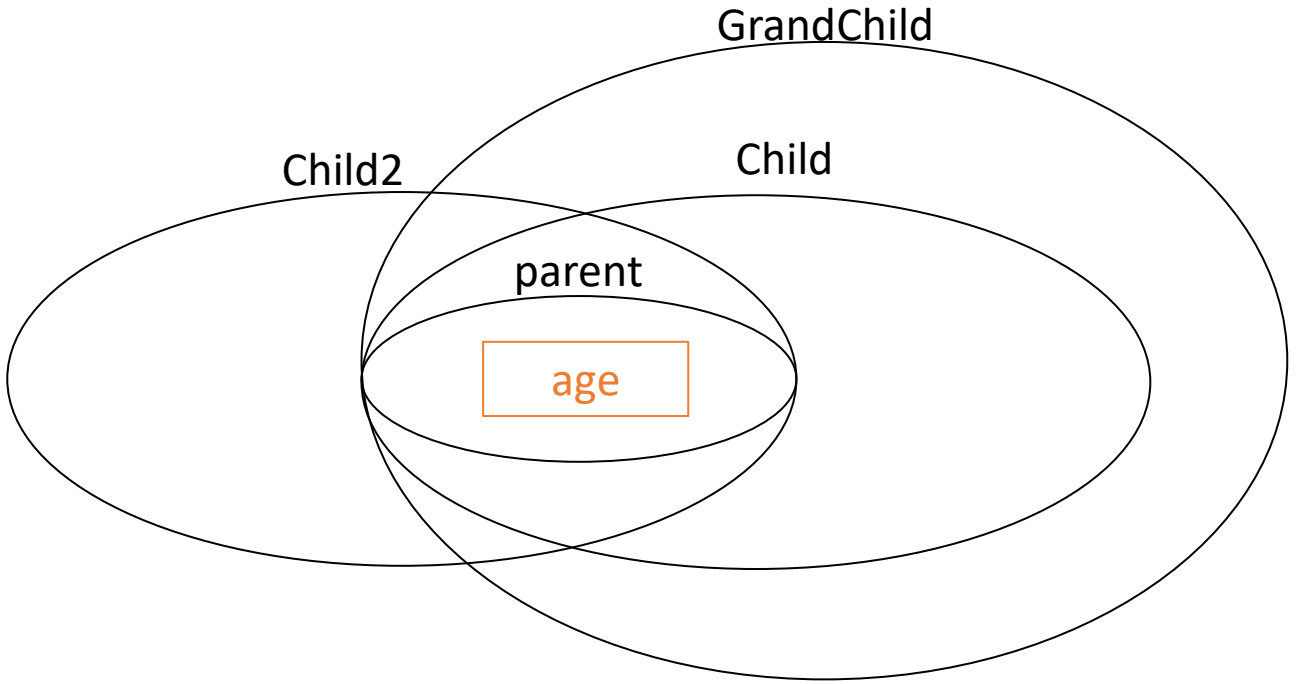


1.1상속의 정의와 장점

```java
class parent {
    int age;
}

class child extends parent {
    void play() {
        System.out.println("놀자~");
    }
}
```

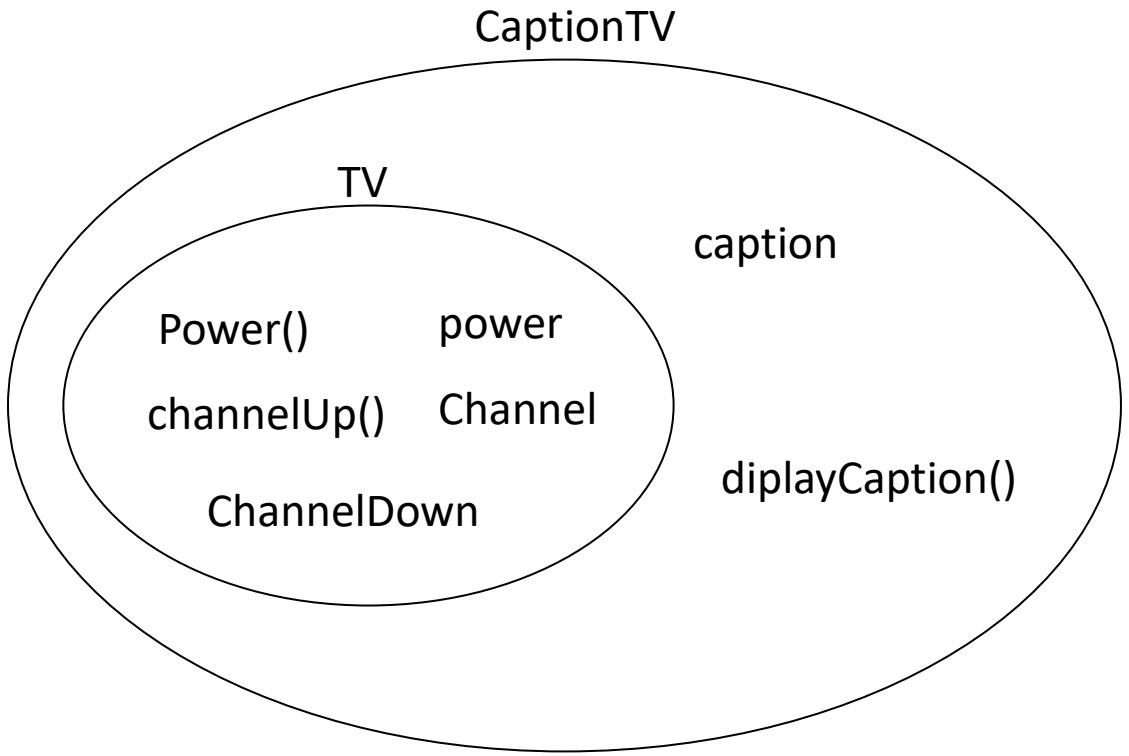



```java
class parent {
    int age;
}
class child extends parent {}
class child2 extends parent {}
class GrandChild extends child {}
```



1.1상속의 정의와 장점

```java
public class TV {

    boolean power; // 전원상태(on/off)
    int channel; //채널

    void power() { power = !power; }
    void channelUp() { ++channel; }
    void channelDown() { --channel; }

}

class CaptionTv extends TV{
    boolean Caption; // 캡션상태(on/off)
    void displayCaption (String text) {
        if (Caption) {
            System.out.println(text);
        }
    }
}

class CaptionTvTest {
    public static void main (String args[]) {
        CaptionTv ctv = new CaptionTv();
        ctv.channel = 10;
        ctv.channelUp();
        System.out.println(ctv.channel);
        ctv.displayCaption("Hello, World");
        ctv.Caption = true;
        ctv.displayCaption("Hello, Wolrd");
    }
}
```



1.2 클래스간의 관계 - 포함관계

□ 클래스간 포함 관계

- 한 클래스의 멤버변수로 다른 클래스 타입의 참조변수를 선언하는 것.

```
class Circle {
    int x;          // 원점의 x좌표
    int y;          // 원점의 y좌표
    int r;          // 반지름
}

class Point {
    int x;
    int y;
}
```

```
class Circle {
    Point c = new point();
    int r;
}
```

오버라이딩

2.1 오버라이딩이란?

□ 오버라이딩이란?

- 조상 클래스로부터 상속받은 메서드의 내용을 변경하는 것

```
class point {
    int x;
    int y;

    String getLocation() {
        return "x :" + x+",y:"+y;
    }

}

class point3D extends point{
    int z;

    String getLocation() {
        return "x :" + x+",y:"+y+",z :"+z;
    }

}
```

2.2 오버라이딩의 조건

자손 클래스에서 오버라이딩하는 메서드는 조상 클래스의 메서드와

- 이름이 같아야 한다
- 매개변수가 같아야 한다 → 선언부가 서로 일치해야 한다.
- 반환타입이 같아야 한다

조상 클래스의 메서드를 자손 클래스에서 오버라이딩할 때

- 접근 제어자를 조상 클래스의 메서드보다 좁은 범위로 변경할 수 없다
- 예외는 조상 클래스의 메서드보다 많이 선언 할 수 없다.
- 인스턴스 메서드를 static메서드로 또는 그 반대로 변경할 수 없다.

2.3 오버로딩 vs 오버라이딩

□ 오버로딩 (overloading) – 기존에 없는 새로운 메서드를 정의하는 것 (new)

□ 오버라이딩 (overriding) – 상속받은 메서드의 내용을 변경하는 것 (change, modify)

```
class parent {

    void parentMethod() {}
}

class child extends parent{
    void parentMethod() {}       // 오버라이딩
    void parentMethod(int i) {} // 오버로딩

    void childMethod() {}        // 오버라이딩
    void childMethod(int i) {}  // 오버로딩

}
```

2.4 super

☐ super

- 상속받은 멤버와 자신의 클래스에 정의된 멤버의 이름이 같을 때 super를 붙여서 구별
- this 를 사용할 수 있지만 조상, 자손 클래스의 멤버가 중복되어 서로 구별해야 할 땐 super를 사용

```
class SuperTest {
    public static void main(String args[]) {
        Child c = new Child();
        c.method();
    }
}

class Parent {
    int x = 10;
}

class Child extends Parent {
    void method() {
        System.out.println("x=" + x);
        System.out.println("this.x=" + this.x);
        System.out.println("super.x=" + super.x);
    }
}
```

```
class SuperTest2 {
    public static void main(String args[]) {
        Child c = new Child();
        c.method();
    }
}

class Parent2 {
    int x =10 ;
}

class Child1 extends Parent2 {
    int x = 20;

    void method() {
        System.out.println("x=" + x);
        System.out.println("this.x=" + this.x);
        System.out.println("super.x=" + super.x);
    }
}
```

※ 조상 클래스의 메서드를 자손 클래스에서 오버라이딩한 경우에 super를 사용한다

패키지

3.1 패키지

□ 패키지란

- 클래스의 묶음
- 패키지에는 클래스 또는 인터페이스를 포함 시킬 수 있음
- 서로 관련된 클래스들끼리 그룹 단위로 묶어 효율적으로 관리할 수 있음

→ 물리적으로 하나의 디렉토리

- 하나의 소스파일에는 첫 번째 문장으로 단 한번의 패키지 선언만 허용
- 모든 클래스는 반드시 하나의 패키지에 속해야 한다.
- 패지키는 점(.)을 구분자로 하여 계층구조로 구성할 수 있다.
- 패키지는 클래스 파일(.class)을 포함하는 하나의 디렉토리이다.

3.2 패키지의 선언

□ package 패키지명;

- 소스파일에서 주석과 공백을 제외한 첫 번째 문장
- 하나의 소스파일에 단 한번만 선언될 수 있다.
- 클래스명과 쉽게 구분하기 위해 소문자로 하는 것을 원칙으로 한다.

Package 패키지명;

3.3 import문

- 다른 패키지의 클래스를 사용하려면 패키지명이 포함된 클래스 이름을 사용해야 한다.

- 사용하고자 하는 클래스의 패키지를 미리 명시해 주면 소스코드에 사용되는 클래스이름에서 패키지명을 생략 할 수 있다.

※ 이클립스 단축키 : ctrl + shift + o

※ 프로그램의 성능에는 영향 X, 단, 컴파일 시간이 소폭 증가.

3.4 import문의 선언

- Import문은 package문 다음에, 클래스 선언문 이전에 위치
- Import문은 한 소스파일에 여러 번 선언 가능

□ 소스파일의 구성

1. Package문
2. Import문
3. 클래스 선언

□ import문의 선언 방법

Import 패키지명. 클래스명;

또는

Import 패키지명.*;

Import java.util. Calendar;
Import java.util.Date; → Import java.util.*;
Import java.util.ArrayList

※ 성능상의 차이는 전혀 없다.

Static import문

- Static import문을 사용하면 static멤버를 호출할 때 **클래스 이름을 생략**할 수 있다

- 특정 클래스의 static멤버를 자주 사용할 때 용이

```
import static java.lang.System.out;
import static java.lang.Math.*;

class StaticImportEx1 {
    public static void main(String args[]) {
        //System.out.println(Math.random());
        out.println(random());
        //System.out.println("Math.PI :" +Math.PI);
        out.println("Math.PI :" + PI);
    }
}
```

제어자

4.1 제어자란?

- 클래스, 변수 또는 메서드의 선언부에 함께 사용되어 부가적인 의미를 부여

제어자 → 접근 제어자 : public, protected, default, private

제어자 → 그 외 : static, final, abstract, native, transient, synchronized, volatile, strictfp

- 하나의 대상에 대해서 여러 제어자를 조합하여 사용가능
- 단, 접근 제어자는 네 가지 중 하나만 선택해 사용 가능

4.2 static – 클래스의, 공통적인

- 클래스 변수(static멤버변수)는 인스턴스에 관계없이 같은 값을 갖는다.

→ Static이 사용될 수 있는 곳 – 멤버변수, 메서드, 초기화 블럭

static

멤버변수
- 모든 인스턴스에 공통적으로 사용되는 클래스변수가 된다
- 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다
- 클래스가 메모리에 로드될 때 생성된다.

메서드
- 인스턴스를 생성하지 않고도 호출이 가능한 static 메서드가 된다
- Static메서드 내에서는 인스턴스멤버들을 직접 사용할 수 없다.

4.3 final – 마지막의, 변경될 수 없는

- 거의 모든 대상에 사용 될 수 있다.
- 변수(멤버변수, 지역변수)에 사용되면 상수
- 메서드에 사용되면 오버라이딩을 할 수 없다.
- 클래스에 사용되면 자손 클래스를 정의하지 못한다.

```
final class FinalTest{                  //클래스
    final int MAX_SIZE = 10;            //멤버변수

    final void getMaxSize() {           //메서드
        final int LV = MAX_SIZE;        //지역변수
    }
}
```

4.4 abstract – 추상의, 미완성의

- 메서드의 선언부만 작성하고 실제 수행 내용은 구현하지 않은 추상 메서드를 선언하는데 사용

- 클래스 내에 추상메서드가 존재한다는 것을 알 수 있게 한다.

abstract

- 클래스 – 클래스 내에 추상 메서드가 선언되어 있음을 의미
- 메서드 – 선언부만 작성하고 구현부는 작성하지 않은 추상 메서드임을 알림.

※ 추상 클래스는 미완성 설계도이므로 인스턴스를 생성 할 수 없다.

4.5 접근제어자

- 멤버 또는 클래스에 사용되어, 해당하는 멤버 또는 클래스를 외부에서 접근하지 못하도록 제한하는 역할을 한다.

접근 제어자
- Private : 같은 클래스 내에서만 접근이 가능하다
- Default : 같은 패키지 내에서만 접근이 가능하다
- Protected : 같은 패키지 내에서, 다른 패키지의 자손클래스에서 접근이 가능하다
- Public : 접근 제한이 전혀 없다.

※ public > protected > (default) > private

클래스	→	Public, (default)
메서드 멤버변수	➡	Public, protected, (default), private
지역변수	→	없음

4.6 제어자의 조합

대상	사용가능한 제어자
클래스	Public, (default), final, abstract
메서드	모든 접근 제어자, final, abstract, static
멤버변수	모든 접근 제어자, final, static
지역변수	final

※ 제어자 조합시 주의사항

1. 메서드에 static과 abstract를 함께 사용 할 수 없다.
2. 클래스에 abstract와 final을 동시에 사용할 수 없다.
3. Abstract메서드의 전근 제어자가 private일 수 없다.
4. 메서드에 pirtvate와 final을 같이 사용할 필요는 없다.

다형성

5.1 다형성이란?

- 여러가지 형태를 가질 수 있는 능력
- 한 타입의 참조변수로 여러 타입의 객체를 참조할 수 있도록 함.
- 조상클래스의 타입의 참조변수로 자손클래스의 인스턴스를 참조할 수 있도록 함.

```
class TV {
    boolean power;
    int channel;

    void power()      { power = !power;}
    void chnnelUp()   { ++channel;     }
    void chnnelDown() { --channel;     }
}

class CaptionTv extends TV {
    String text;
    void caption() {}
}
```

```
TV t = new TV();
CaptionTv c = new CaptionTv();
```

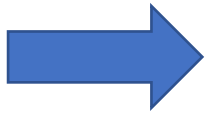
- 인스턴스 타입과 일치하는 참조 변수 사용

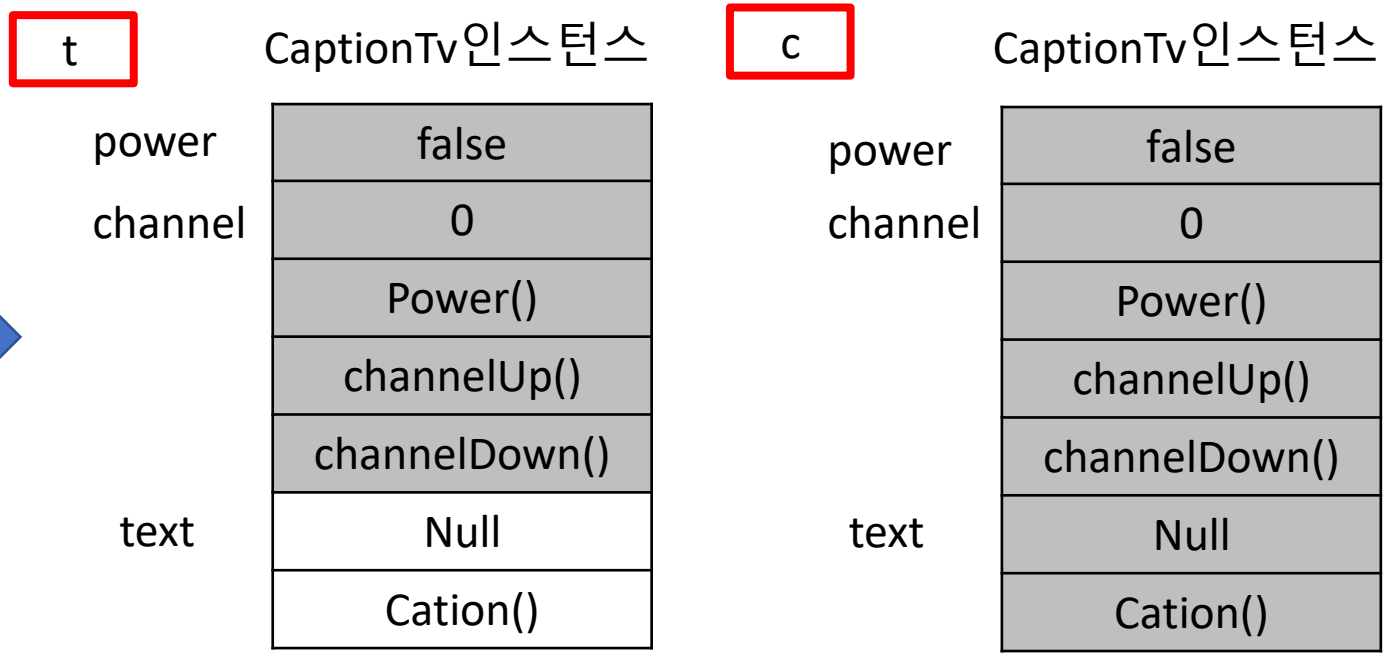
```
CaptionTv c = new CaptionTv();
TV t = new CaptionTv();
```

- 상속관계에 있을 경우 조상 클래스 타입의 참조변수로 자손 클래스의 인스턴스 참조 가능

5.1 다형성이란?

```
CaptionTv c = new CaptionTv();
TV t = new CaptionTv();
```





☐ `CaptionTv c = new TV();`

※ 조상타입의 참조변수로 자손타입의 인스턴스를 참조할 수 있다
반대로 자손타입의 참조변수로 조상타입의 인스턴스를 참조할 수는 없다.

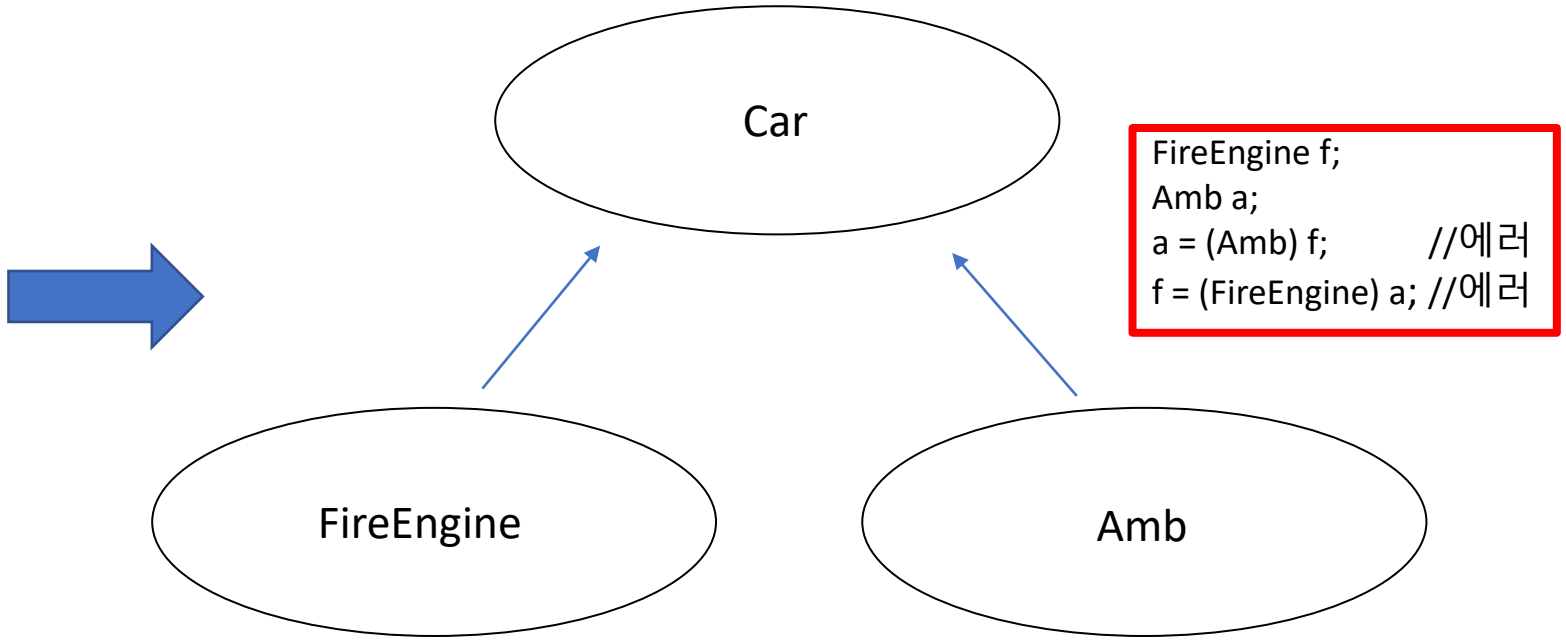
5.2 참조변수의 형변환

- 기본형 변수와 같이 참조변수도 형 변환이 가능 단, 서로 상속관계에 있는 클래스사이에서만 가능

자손타입 -> 조상타입(Up-casting) : 형변환 생략가능

자손타입 <- 조상타입 (Down-casting) : 형변환 생략불가

```
class Car {
    String color;
    int door;
    void drive() {                  // 운전기능
        System.out.println("부릉");
    }

    void stop() {                   // 제동기능
        System.out.println("멈춰!");
    }
}

class FireEngine extends Car {
    void water() {                  // 물뿌리는 기능
        System.out.println("물대포!");
    }
}

class Amb extends Car {
    void siren() {                  // 사이렌 울리는 기능
        System.out.println("삐뽀삐뽀");
    }
}
```



5.2 참조변수의 형변환

```
class CastingTest1 {
    public static void main(String args[]) {
        Car car = null;
        FireEngine fe = new FireEngine ();
        FireEngine fe2 = null;

        fe.water();
        car = fe;                   // (car)생략
        //car.water() Car타입의 참조변수로는 water()를 호출 할 수 없다.
        fe2= (FireEngine)car; // 자손타입 <- 조상타입 (Down)
        fe2.water();
    }
}
```

```
class Car {
    String color;
    int door;
    void drive() {                  // 운전기능
        System.out.println("부릉");
    }

    void stop() {                   // 제동기능
        System.out.println("멈춰!");
    }
}

class FireEngine extends Car {
    void water() {                  // 물뿌리는 기능
        System.out.println("물대포!");
    }
}

class Amb extends Car {
    void siren() {                  // 사이렌 울리는 기능
        System.out.println("삐뽀삐뽀");
    }
}
```

5.3 instanceof연산자

- 참조변수가 참조하고 있는 인스턴스의 실제 타입 식별을 위해 사용
- 주로 조건문에 사용
- 연산결과는 boolean값인 true와 false 중 하나를 반환.
- 연산결과로 true를 얻었다는 것은 참조변수가 검사한 타입으로 형변환이 가능하다는 뜻

참조변수 **instanceof** 클래스명

5.5 매개변수의 다형성

- 참조변수의 다형적인 특징은 메서드의 매개변수에도 적용된다.

1

```
class Product {
    int price;
    int bonusPoint;
}
class Tv       extends Product{}
class Computer extends Product{}
class Audio    extends Product{}

class Buyer {
    int money = 1000;
    int bonusPoint = 0;

    void buy(Tv t) {
        money = money - t.price;
        bonusPoint = bonusPoint + t.bonusPoint;
    }
```

물건(Tv)를 구입하는 메서드

2

```
void buy(Computer c) {
    money = money - c.price;
    bonusPoint = bonusPoint + c.bonusPoint;
}

void buy(Audio a) {
    money = money - a.price;
    bonusPoint = bonusPoint + a.bonusPoint;
}
```

- 구매 품목이 늘어날 때 마다 메서드를 추가해야 함.

3

```
void buy(Product p) {
    money = money - p.price;
    bonusPoint = bonusPoint + p.bonusPoint;
}
```

- 매개변수에 다형성을 적용
- 매개변수가 product타입의 참조변수

5.5 매개변수의 다형성

```
class Product {
    int price;
    int bonusPoint;

    Product(int price) {
        this.price = price;
        bonusPoint = (int)(price/10.0);
    }
}

class Tv extends Product {
    Tv() {
        // 조상클래스의 생성자 product(int price)를 호출한다
        super(100);   //tv가격 100만원
    }

    public String toString() { return "Tv"; }
}

class Computer extends Product {
    Computer () { super(200); }
    public String toString() { return "Computer"; }
}
```

```
class Buyer {
    int money = 1000;
    int bonusPoint = 0;

    void buy (Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살 수 없습니다.");
            return;
        }
        money -= p.price;
        bonusPoint += p.bonusPoint;
        System.out.println (p + "을/를 구입하셨습니다.");
    }
}

class PolyArgumentTest {
    public static void main(String args[]) {
        Buyer b = new Buyer();

        b.buy(new Tv ());
        b.buy(new Computer());
    }
}
```

5.6 여러 종류의 객체를 배열로 다루기

- 조상타입의 참조변수 배열을 사용하면,
공통의 조상을 가진 서로 다른 종류의 객체를 배열로 묶어서 다룰 수 있다.

```
Product p1 = new Tv();
Product p2 = new Computer();
Product p3 = new Audio
```

Product타입의
참조변수 배열 처리

```
Product p[] = new Product [3]
P[0] = new Tv();
P[1] = new Computer();
P[2] = new Audio();
```

```java
class Buyer {
    int money = 1000;
    int bonusPoint = 0;
    Product[] item = new Product[10];
    int i = 0;

    void buy (Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살 수 없습니다.");
            return;
        }
        money -= p.price;
        bonusPoint += p.bonusPoint;
        item[i++] = p;
        System.out.println (p + "을/를 구입하셨습니다.");
    }
}
```

추상클래스

6.1 추상클래스란? (abstract class)

- 미완성 메서드(추상메서드)를 포함하고 있다는 의미
- 추상클래스로 인스턴스는 생성할 수 없다.
- 상속을 통해 자손클래스에 의해서만 완성될 수 있다.

```
abstract class 클래스이름 {
    ...
}
```

※ 추상메서드를 포함하지 않은 클래스에도 abstract를 붙여서 추상클래스로 지정 할 수 있다.
단, 추상클래스로 지정되면 클래스의 인스턴스를 생성할 수 없다.

6.2 추상메서드

- 메서드의 선언부만 작성하고 구현부는 작성하지 않은 채로 남겨둔 것

```
/* 어떤 기능을 수행 목적으로 작성했는지 작성 */
abstract 리턴타입 메서드이름();
```

```
abstract class Player{              //추상클래스
    abstract void play(int pos);            //추상메서드
    abstract void stop();                   //추상메서드
}

class AudioPlayer extends Player{
    void play (int pos) {/* 생략 */}        //추상메서드를 구현
    void stop () {/* 생략 */}               //추상메서드를 구현
}
```

6.3 추상클래스의 작성

- 추상화 : 클래스 간의 공통점을 찾아내서 공통의 조상을 만드는 작업
- 구체화 : 상속을 통해 클래스를 구현, 확장하는 작업

```
class Marine {
    int x,y;
    void move(int x, int y) {/* 지정된 위치로 이동 */}
    void stop()             {/* 현재 위치에 정지 */ }
    void stimPack()         {/* 스팀팩을 사용 */   }
}

class Tank {
    int x,y;
    void move(int x, int y) {/* 지정된 위치로 이동 */ }
    void stop()             {/* 현재 위치에 정지 */  }
    void mod()              {/* 공격모드를 변환 */   }

}

class DropShip{
    int x,y;
    void move(int x, int y) {/* 지정된 위치로 이동 */}
    void stop()             {/* 현재 위치에 정지 */ }
    void load()             {/* 선택된 대상 태우기 */}
    void unload()           {/* 선택된 대상 내리기 */}
}
```

```
abstract class unit {
    int x,y;
    abstract void move (int x, int y);
    void stop() { /* 현재 위치에 정지 */ }
}

class Marine extends Unit {
    void move (int x, int y) { /* 지정된 위치로 이동 */ }
    void stimpack() { /* 스팀팩을 사용한다 */ }
}

class Tank extends Unit {
    void move (int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode()        { /* 공격 모드로 변환 */ }
}

class DropShip extends Unit {
    void move (int x, int y) { /* 지정된 위치로 이동 */ }
    void load()              {/* 선택된 대상 태우기 */}
    void unload()            {/* 선택된 대상 내리기 */}
}
```

인터페이스

7.1 인터페이스란?

- 일종의 추상클래스
- 추상클래스보다 추상화 정도가 높다
- 몸통을 갖춘 일반 메서드 또는 멤버변수를 구성원으로 가질 수 없다.
- 추상메서드와 상수만을 멤버로 가질 수 있으며, 그 외의 어떠한 요소도 허용하지 않는다.
- 다른 클래스를 작성하는데 도움 줄 목적으로 작성

7.2 인터페이스의 작성

- Class를 작성하는 것과 같으며, 키워드로 interface를 사용.

- 접근제어자로 public, default를 사용할 수 있다.

```
interface 인터페이스이름 {
        public static final 타입 상수이름 = 값;
        public abstract 메서드이름 (매개변수목록);
}
```

※ 인터페이스 멤버 제약사항

- 모든 멤버 변수는 public static final 이어야 하며, **이를 생략할 수 있다.**
- 모든 메서드는 public abstract 이어야 하며, **이를 생략할 수 있다.**

7.3 인터페이스의 상속

- 인터페이스는 인터페이스로만 상속받을 수 있으며,
- 클래스와 달리 다중상속이 가능하다.

```
interface Movable {
    /* (x,y)로 이동하는 기능의 메서드 */
    void move(int x, int y);
}

interface Attackable {
    /* 지정된 (u)를 공격하는 기능의 메서드 */
    void attack (Unit u);
}

interface Fightable extends Movable, Attackable{}
```

자손 인터페이스는 조상 인터페이스에 정의된 멤버를 모두 상속 받는다.

Fightable은 move, attack 메서드를 멤버로 갖는다.

7.4 인터페이스의 구현

- 인터페이스도 추상클래스처럼 그 자체로는 인스턴스를 생성 할 수 없다.
- 따라서, 자신에 정의된 추상메서드의 몸통을 만들어주는 클래스를 작성해야 한다.
- 추상클래스가 자신을 상속받는 클래스를 정의하는 것과 다르지 않다.
- 구현한다는 의미의 키워드 implements 사용.

```
Class 클래스이름 implements 인터페이스이름 {
        // 인터페이스에 정의된 추상메서드 구현
}
```

```
class Fighter implements Fightable {
    public void move(int x, int y) {}
    public void attack (Unit u) {}
}
```

※ Fightable 인터페이스 구현

```
abstract class Fighter implements Fightable {
    public void move(int x, int y) {}
}
```

※ 메소드 중 일부만 구현 (추상클래스로 선언)

```
interface Fightable extends Movable, Attackable{}

class Fighter extends Unit implements Fightable {
    public void move(int x, int y) {}
    public void attack(Unit u) {}
}
```

※ 상속과 동시에 구현

심화. 디자인패턴이란?

- 소프트웨어 디자인에서 자주 발생하는 고질적인 문제들이 발생했을 때 재사용 할 수 있는 해결책
- 완성된 디자인은 아니며 다른 여러 상황에 맞게 사용되는 일종의 템플릿.

※ 패턴 : 다양한 응용 소프트웨어 시스템을 개발할 때 공통되는 설계 문제가 존재하며 이를 처리하는 해결책 사이에도 공통점이 있다. 이러한 유사점을 패턴이라 한다.

□ 디자인 패턴 구조

- Context : 문제가 발생하는 여러 상황 기술, 즉 패턴이 적용될 수 있는 상황
- Problem : 패턴의 적용으로 해결될 필요가 있는 디자인 이슈들을 기술
- Solution : 요소들 사이의 관계, 책임, 협력 등을 기술.

디자인 패턴의 종류

□ GOF 디자인 패턴
- 23가지의 디자인 패턴을 정리하고 **생산, 구조, 행위** 3가지로 분류

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
• 추상 팩토리(Abstract Factory) • 빌더(Builder) • 팩토리 메서드(Factory Methos) • 프로토타입(Prototype) • 싱글턴(Singleton)	• 어댑터(Adapter) • 브리지(Bridge) • 컴퍼지트(Composite) • 데커레이터(Decorator) • 퍼사드(Facade) • 플라이웨이트(Flyweight) • 프록시(Proxy)	• 책임 연쇄(Chain of Responsibility) • 커맨드(Command) • 인터프리터(Interpreter) • 이터레이터(Iterator) • 미디에이터(Mediator) • 메멘토(Memento) • 옵서버(Observer) • 테이트(State) • 스트래티지(Strategy) • 템플릿 메서드(Template Method) • 비지터(Visitor)

디자인 패턴의 종류

□ 생성 패턴

- 객체 생성에 관련된 패턴
- 객체가 생성되는 과정의 유연성을 높이고 코드의 유지를 쉽게 함

□ 구조 패턴

- 프로그램 구조에 관련된 패턴
- 프로그램의 구조를 설계하는데 활용할 수 있는 패턴

□ 행위 패턴

- 반복적으로 사용되는 객체들의 상호작용을 패턴화 해놓은 것들
- 여러 객체가 수행하는 작업을 어떻게 분배하는지, 객체 사이의 결합도를 최소화 하는것에 중점을 둠.