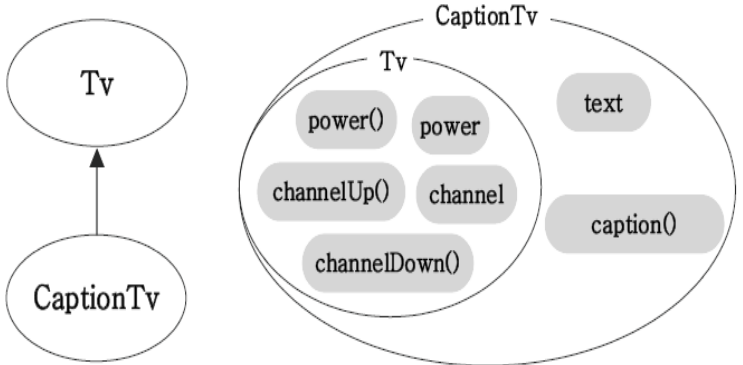
객체 지향 프로그래밍4_다형성 1

다형성(polymorphism) "여러 가지 형태를 가질 수 있는 능력"

- "하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것" 즉, 조상타입의 참조변수로 자손타입의 객체를 다룰 수 있다.
- "조상타입의 참조변수로 자손타입의 인스턴스를 참조할 수 있지만, 반대로 자손타입의 참조변수로 조상타입의 인스턴스를 참조할 수는 없다."

```
class Tv {
    boolean power;  // 전원상태(on/off)
    int channel;    // 채널

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

class CaptionTv extends Tv {
    String text;    // 캡션내용
    void caption() { /* 내용생략 */}
}
```

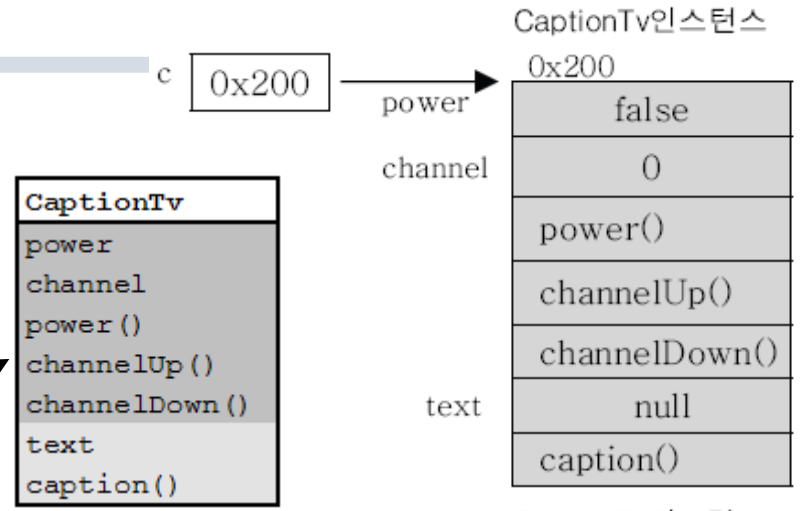


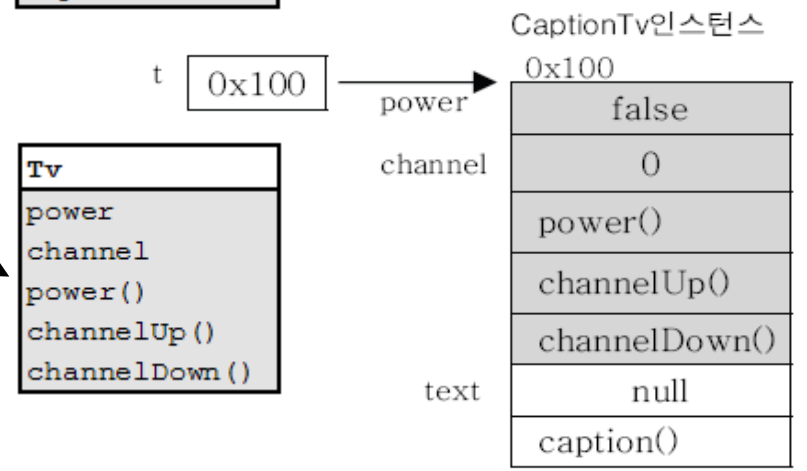

```
CaptionTv c = new CaptionTv();

Tv        t = new CaptionTv();
```

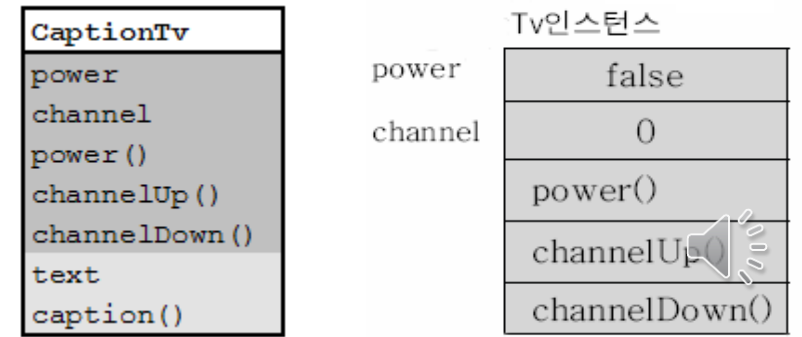
```
Tv         t = new Tv();
CaptionTv  c = new CaptionTv();
```

```
Tv         t   = new CaptionTv();
```

? `CaptionTv c = new Tv();`







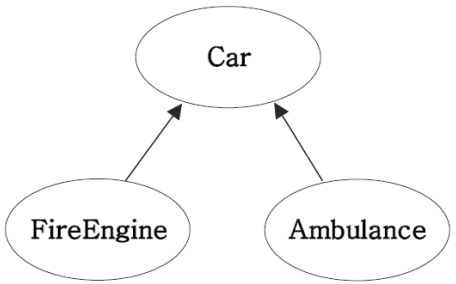
다형성 - 참조변수의 형변환

- 서로 상속관계에 있는 타입간의 형변환만 가능하다.
- 자손 타입에서 조상타입으로 형변환하는 경우, 형변환 생략가능.

자손타입 → 조상타입 (Up-casting) : 형변환 생략가능
자손타입 ← 조상타입 (Down-casting) : 형변환 생략불가

```
class Car {
    String color;
    int door;

    void drive() {  // 운전하는 기능
        System.out.println("drive, Brrrr~");
    }

    void stop() {   // 멈추는 기능
        System.out.println("stop!!!");
    }
}

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

class Ambulance extends Car {   // 구급차
    void siren() {  // 사이렌을 울리는 기능
        System.out.println("siren~~~");
    }
}
```

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

    fe.water();
    car = fe;   // car = (Car)fe; 조상 <- 자손
//  car.water();
    fe2 = (FireEngine)car; // 자손 <- 조상
    fe2.water();
}
```



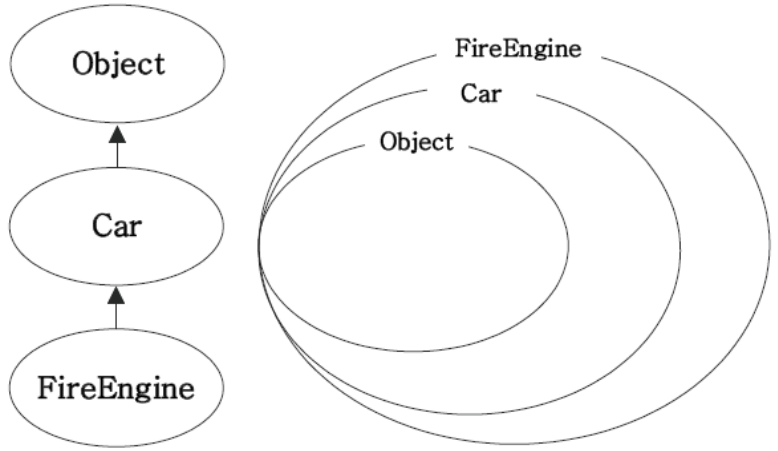

```
FireEngine f
Ambulance a;

a = (Ambulance)f;
f = (FireEngine)a;
```

instanceof연산자

- 참조변수가 참조하는 인스턴스의 실제 타입을 체크하는데 사용.
- 이항연산자이며 피연산자는 참조형 변수와 타입. 연산결과는 true, false.
- instanceof의 연산결과가 true이면, 해당 타입으로 형변환이 가능하다.

```
class InstanceofTest {
    public static void main(String args[]) {
        FireEngine fe = new FireEngine();

        if(fe instanceof FireEngine) {
            System.out.println("This is a FireEngine instance.");
        }

        if(fe instanceof Car) {
            System.out.println("This is a Car instance.");
        }

        if(fe instanceof Object) {
            System.out.println("This is an Object instance.");
        }
    }
}
```

```
---------- java ----------
This is a FireEngine instance.
This is a Car instance.
This is an Object instance.

출력 완료 (0초 경과)
```




```
void method(Object obj) {
    if(objinstanceof Car) {
        Car c = (Car)obj;
        c.drive();
    } else if(objinstanceof FireEngine) {
        FireEngine fe = (FireEngine)obj;
        fe.water();
    }
}
```

참조변수와 인스턴스변수의 연결

- 멤버변수가 중복정의된 경우, 참조변수의 타입에 따라 연결되는 멤버변수가 달라진다. (참조변수타입에 영향받음)
- 메서드가 중복정의된 경우, 참조변수의 타입에 관계없이 항상 실제 인스턴스의 타입에 정의된 메서드가 호출된다.(참조변수타입에 영향받지 않음)

```
class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    int x = 200;

    void method() {
        System.out.println("Child Method");
    }
}
```

```
p.x = 100
Child Method
c.x = 200
Child Method
```

```
class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent { }


    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();

        System.out.println("c.x = " + c.x);
        c.method();
    }
```

```
p.x = 100
Parent Method
c.x = 100
Parent Method
```

매개변수의 다형성

- 참조형 매개변수는 메서드 호출시, 자신과 같은 타입 또는 자손타입의 인스턴스를 넘겨줄 수 있다.

```
class Product {
    int price;      // 제품가격
    int bonusPoint; // 보너스점수
}

class Tv extends Product {}
class Computer extends Product {}
class Audio extends Product {}

class Buyer { // 물건사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수
}
```

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

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

```
Buyer b = new Buyer();

Tv tv = new Tv();
Computer com = new Computer();

b.buy(tv);
b.buy(com);
```

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

여러 종류의 객체를 하나의 배열로 다루기(1/3)

- 조상타입의 배열에 자손들의 객체를 담을 수 있다.

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

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

```
class Buyer { // 물건사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수

    Product[] cart = new Product[10]; // 구입한 물건을 담을 배열

    int i=0;

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액부족");
            return;
        }

        money -= p.price;
        bonusPoint += p.bonusPoint;
        cart[i++] = p;
    }
}
```

여러 종류의 객체를 하나의 배열로 다루기(2/3)

- ▶ java.util.Vector – 모든 종류의 객체들을 저장할 수 있는 클래스

메서드 / 생성자	설 명
`Vector()`	10개의 객체를 저장할 수 있는 Vector인스턴스를 생성한다. 10개 이상의 인스턴스가 저장되면, 자동적으로 크기가 증가된다.
`boolean add(Object o)`	Vector에 객체를 추가한다. 추가에 성공하면 결과값으로 true, 실패하면 false를 반환한다.
`boolean remove(Object o)`	Vector에 저장되어 있는 객체를 제거한다. 제거에 성공하면 true, 실패하면 false를 반환한다.
`boolean isEmpty()`	Vector가 비어있는지 검사한다. 비어있으면 true, 비어있지 않으면 false를 반환한다.
`Object get(int index)`	지정된 위치(index)의 객체를 반환한다. 반환타입이 Object타입이므로 적절한 타입으로의 형변환이 필요하다.
`int size()`	Vector에 저장된 객체의 개수를 반환한다.

```
public class Vector extends AbstractList implements List, Cloneable,
                    java.io.Serializable {
    protected Object elementData[];
    ...
}
```

여러 종류의 객체를 하나의 배열로 다루기(3/3)

```
Product[] cart = new Product[10];
//...

void buy(Product p) {
    //...
    cart[i++] = p;
}
```



```
Vector cart = new Vector();
//...

void buy(Product p) {
    //...
    cart.add(p);
}
```

메서드 / 생성자
Vector()
boolean add(Object o)
boolean remove(Object o)
boolean isEmpty()
Object get(int index)
int size()

```
void summary() {             // 구매한 물품에 대한 정보를 요약해서 보여준다.
    int sum = 0;             // 구입한 물품의 가격합계
    String cartList ="";     // 구입한 물품목록

    if(cart.isEmpty()) {     // Vector가 비어있는지 확인한다.
        System.out.println("구입하신 제품이 없습니다.");
        return;
    }

    // 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.
    for(int i=0; i<cart.size();i++) {
        Product p = (Product)cart.get(i);
        sum += p.price;
        cartList += (i==0) ? "" + p : ", " + p;
    }
    System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
    System.out.println("구입하신 제품은 " + cartList + "입니다.");
}
```

```
class Tv extends Product {
    Tv() { super(100); }
    public String toString() { return "Tv"; }
}
```

```
Object obj = cart.get(i);
sum += obj.price; // 에러
```

추상클래스(abstract class)

- 클래스가 설계도라면 추상클래스는 '미완성 설계도'
- 추상메서드(미완성 메서드)를 포함하고 있는 클래스

  * 추상메서드 : 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
abstract class Player {
     int currentPos;          // 현재 Play되고 있는 위치를 저장하기 위한 변수

     Player() {               // 추상클래스도 생성자가 있어야 한다.
          currentPos = 0;
     }

     abstract void play(int pos);      // 추상메서드
     abstract void stop();             // 추상메서드

     void play() {
          play(currentPos);            // 추상메서드를 사용할 수 있다.
     }
    ...
}
```

- 일반메서드가 추상메서드를 호출할 수 있다.(호출할 때 필요한 건 선언부)
- 완성된 설계도가 아니므로 인스턴스를 생성할 수 없다.
- 다른 클래스를 작성하는 데 도움을 줄 목적으로 작성된다.

추상메서드(abstract method)

- 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
/* 주석을 통해 어떤 기능을 수행할 목적으로 작성하였는지 설명한다. */
abstract 리턴타입 메서드이름();

Ex)
/* 지정된 위치(pos)에서 재생을 시작하는 기능이 수행되도록 작성한다.*/
abstract void play(int pos);
```

- 꼭 필요하지만 자손마다 다르게 구현될 것으로 예상되는 경우에 사용
- 추상클래스를 상속받는 자손클래스에서 추상메서드의 구현부를 완성해야 함.

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

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

abstract class AbstractPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
}
```

추상클래스의 작성

- 여러 클래스에 공통적으로 사용될 수 있는 추상클래스를 바로 작성하거나 기존클래스의 공통 부분을 뽑아서 추상클래스를 만든다.

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

class Tank {      // 탱크
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void changeMode()       { /* 공격모드를 변환한다. */}
}

class Dropship {  // 수송선
    int x, y;     // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()             { /* 현재 위치에 정지 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
}
```

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

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

class Tank extends Unit {    // 탱크
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode()       { /* 공격모드를 변환한다. */}
}

class Dropship extends Unit { // 수송선
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
}
```

```
Unit[] group = new Unit[4];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();

for(int i=0;i< group.length;i++) {
    group[i].move(100, 200);
}
```

추상메서드가 호출되는 것이 아니라 각 자손들에 실제로 구현된 move(int x, int y)가 호출된다.