Java Web Programming 입문 07

(Java #07)

오늘의 키워드

- 상속 (Inheritance)
  - super, super()
- 패키지 (Package)
  - import
- 캡슐화 (Encapsulation)
  - 제어자(Modifier)
  - 접근 속성(Access Attribute) 혹은 접근 제어자(Access Modifier) 혹은 접근 지정자(Access Identifier)
- 다형성 (Polymorphism)
  - 다형성 (Polymorphism)
  - 형변환, Upcasting, Downcasting

super

- Definition
  - Variable for Parent Class Instance

- Why?
  - Contact Parent Class's member
    - Call parent constructor
      - Child constructor 1st line
      - compiler
    - Call parent member
      - super.variable; super.method();

super()

- Parent Class's Constructor
  - this() 는 동일 클래스내 다른 생성자 호출
  - super()는 조상 클래스의 생성자 호출
  - 상속이 여러 번일 경우 최상위 조상 클래스까지 올라간다

  - Object 클래스를 제외한 모든 클래스의 생성자 첫 줄에는 생성자 (같은 클래스의 다른 생성자 또는 조상의 생성자)를 호출해야함
    - 그렇지 않으면 컴파일러가 자동으로 super(); 를 생성자의 첫 줄에 삽입

1. **클래스** - 어떤 클래스의 인스턴스를 생성할 것인가?
2. **생성자** - 선택한 클래스의 어떤 생성자를 이용해서 인스턴스를 생성할 것인가?

super

```
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    );
    }
}
```

```
x=10
this.x=10
super.x=10
```

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

class Parent {
    int x = 10;
}

class Child extends Parent {
    int x = 20;
    void method() {
        System.out.println("x = " + x);
        System.out.println("this.x = " + this.x);
        System.out.println("super.x = " + super.x);
    }
}
```

```
x=20
this.x=20
super.x=10
```

super()

```
class PointTest {
    public static void main(String args[]{
        Point3D p3 = new Point3D( 1, 2, 3);
    }
}

class Point {
    int x;
    int y;

    Point ( int x, int y ) {
        this.x = x;
        this.y = y;
    }

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

class Point3D extends Point {
    int z;
    Point3D ( int x, int y, int z ) {
        /* ######################################## */
        // Conpilier가 super(); 삽입
        /* ######################################## */
        this.x = x;
        this.y = y;
        this.z = z;
    }

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

```
Point3D (int x, int y, int z) {
    super();
    this.x = x;
    this.y = y;
    this.z = z;
}
```

```
Point3D (int x, int y, int z) {
    // 조상 클래스의 생성자 Point(int x, int y) 호출
    super(x, y);
    this.z = z;
}
```

패키지 (Package)

- Package의 사전적 의미

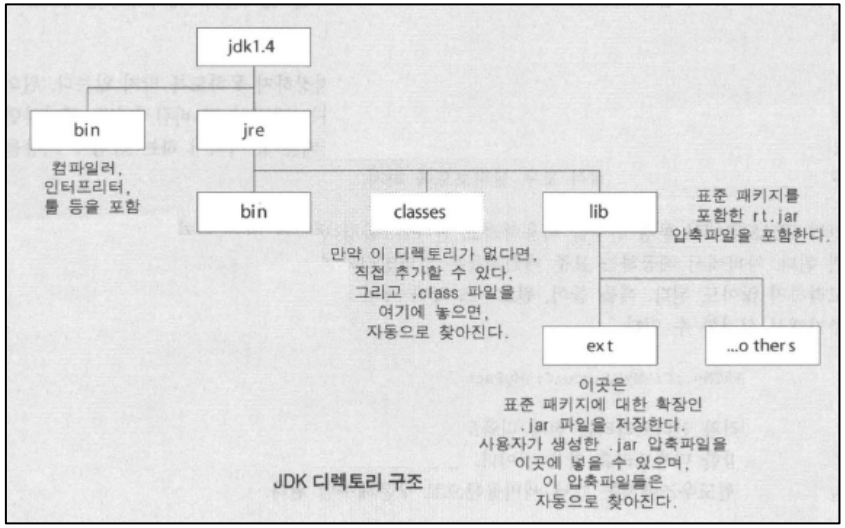
package [ˈpækɪdʒ]

[동의어] parcel

1. [명사](특히 美)
   A large package has arrived for you. 당신 앞으로 큰 소포가 왔어요.
2. [명사](美) (포장용) 상자[봉지 등]; 포장물
   Check the list of ingredients on the side of the package. 포장 상자[봉지] 옆에 적힌 성분 목록을 확인하라.
   a package of hamburger buns 햄버거 빵 한 봉지
3. [명사](=package deal) 일괄 (거래·처리 등)
   a benefits package 일괄 수당
   an aid package 일괄 지원
4. [명사](=software package)(또한 컴퓨터) 패키지, 일괄 프로그램
   The system came with a database software package. 그 시스템은 데이터베이스 소프트웨어 패키지와 함께 나왔다.
5. [동사]~ sth (up) 포장하다
   packaged food / goods 포장된 식품/상품
   We package our products in recyclable materials. 우리는 상품을 재활용 가능 재료들로 포장한다.

패키지 (Package)

- 패키지는…
  - 고유한 이름을 가지는 Class들의 묶음
  - Package에 속한 Class들은 Package로 이름 구분 가능
    - Class 이름간의 충돌 방지
  - 관련된 Class를 그룹 단위로 묶어서 사용
    - 효율적인 관리
    - Project 구조의 가독성



패키지 (Package)

- Package 선언과 사용
  - 하나의 Source File에는
    - 첫 번째 문장으로 단 한 번의 Package 선언만을 허용
  - 모든 Class는
    - 반드시 하나의 Package에 속해야 함
  - Package는
    - 점(.)을 구분자로 하여 계층구조로 구성할 수 있음
    - 물리적으로 클래스 파일(.class)을 포함하는 하나의 디렉토리

```
package Package.Name
```

패키지 (Package)

- import
  - 타 패키지 내 클래스를 사용하려면 풀네임을 써야 한다.
    - 귀찮다...
  - 미리 이 패키지에 어떤 클래스를 쓰겠다 명시
    - 현재 클래스 내에서 패키지명 생략하여 사용 가능
  - 컴파일시 컴파일러가 import문을 참고
    - 사용된 클래스 앞에 패키지명을 붙여준다

패키지 (Package)

- import 선언

```
import PackageName.ClassName;
```

```
import PackageName.*;
```

```
import java.util.Calendar;
import java.util.Date;
import java.util.ArrayList;
```

```
import java.util.*;
```

- 일반적인 소스파일(*.java) 구성 순서

```
package PackageName

import PackageName.*;

Class ClassName
```

- 주의!

```
import java.util.*;
import java.text.*;
```

```
import java.*;
```

패키지 (Package)

- import Example

```
import java.text.SimpleDateFormat;
import java.util.Date;

class ImportTest{
    public static void main(Strings[] args){
        Date today = new Date();
        SimpleDateFormat date = new SimpleDateFormat("yyyy/MM/dd");
        SimpleDateFormat time = new SimpleDateFormat("hh:mm:ss a");

        System.out.println("오늘 날짜는 " + date.format(today));
        System.out.println("오늘 시간은 " + date.format(today));
    }
}
```

```
오늘 날짜는 2003/01/07
현재 시간은 01:59:53 오후
```

```
java.util.Date today = new java.util.Data();
java.text.SimpleDateFormat date = new java.text.SimpleDateFormat("yyyy/MM/dd");
java.text.SimpleDateFormat time = new java.text.SimpleDateFormat("hh:mm:ss a");
```

캡슐화 (Encapsulation)

- Encapsulation의 사전적 의미

encapsulation

1. [명사](의학) 피포(被包), 피막형성(被膜形成)
2. [명사](의학-용어) 피막형성
3. [명사](의학-북한) 피막형성, 피포화

encapsulation

1. [명사]캡슐에 넣음[넣기]; 소중히 보호함[하기].

캡슐화 (Encapsulation)

- 제어자(modifier)
  - 접근 제어자

```
public  protected
default private
```

  - 그 외

```
static      final          abstract    native
transient   synchronized   volatile    strictfp
```

  - 클래스, 변수, 메소드 선언시 사용
  - 이름 그대로 클래스, 변수, 메소드를 정의할 때 제어
  - public, private, static, final, abstract, synchronized
  - 조합도 가능함
  - 접근 제어자는 딱 하나만 사용

캡슐화 (Encapsulation)

- static
  - 멤버변수, 메소드, 초기화 블록

제어자	대상	의 미
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.

- final
  - 클래스, 메소드, 멤버변수, 지역변수

제어자	대상	의 미
final	클래스	변경될 수 없는 클래스, 확장될 수 없는 클래스가 된다. 그래서 final로 지정된 클래스는 다른 클래스의 조상이 될 수 없다.
	메서드	변경될 수 없는 메서드, final로 지정된 메서드는 오버라이딩을 통해 재정의 될 수 없다.
	멤버변수	변수 앞에 final이 붙으면, 값을 변경할 수 없는 상수가 된다.
	지역변수	

캡슐화 (Encapsulation)

- Example

```
class Card{
    final   int     NUMBER; // 상수지만 선언과 함께 초기화 하지 않고
    final   String  KIND;   // 생성자에서 단 한번 초기화할 수 있다.
    static  int     width   = 100;
    static  int     height  = 250;

    Card(String kind, int num){
        KIND    = kind;
        NUMBER  = num;
    }

    Card() {
        this("HEART", 1);
    }

    public String toString(){
        return "" + KIND + " " + NUMBER;
    }
}

class FinalCardtest{
    public static void main(String args[]){
        Card c = new Card("HEART", 10);
//      c.NUMBER = 5;
        System.out.println(c.KIND);
        System.out.println(c.NUMBER);
    }
}
```

```
HEART
10
```

캡슐화 (Encapsulation)

- abstract
  - abstract가 사용될 수 있는 곳
    - 클래스, 메소드

제어자	대상	의 미
abstract	클래스	클래스 내에 추상메서드가 선언되어 있음을 의미한다.
	메서드	선언부만 작성하고 구현부는 작성하지 않은 추상메서드임을 알린다.

- 접근(Access) 속성 (Attribute), 제어자 (Modifier), 지정자 (Identifier)
  - 객체 내 멤버변수에 값을 할당할 때
    - 값을 직접 할당할 수 있는지 여부를 결정
  - 멤버변수의 접근 및 부모자식 클래스간의 접근제어
  - 이러한 접근 방식의 제어를 캡슐화, 은폐화라고 표현

캡슐화 (Encapsulation)

```java
/**
private 멤버 변수를 포함한 클래스
**/
public class Person {
    public int age; //public 멤버 변수 선언
    public float height; //public 멤버 변수 선언
    private float weight; //private 멤버 변수 선언
}
```

```java
/**
private에 직접 접근하기 때문에 에러가 발생하는 예
**/
public class PrivateAccessMain{
    public static void main(String[] args) {
        Person brother = new Person(); //객체 생성
        brother.age =100; //public 멤버 접근
        brother.height = 170.0F; //public 멤버 접근
        brother.weight = 67.0F; //private 멤버 접근 - 에러
        System.out.println("age:" + brother.age); //public 멤버 접근
        System.out.println("height:" + brother.height); //public 멤버 접근
        System.out.println("weight:" + brother.weight); //private 멤버 접근 - 에러
    }
}
```

캡슐화 (Encapsulation)

- 접근(Access) 속성 (Attribute), 제어자 (Modifier), 지정자 (Identifier)
  - 접근 제어자가 사용될 수 있는 곳
    - 클래스, 멤버변수, 메소드, 생성자

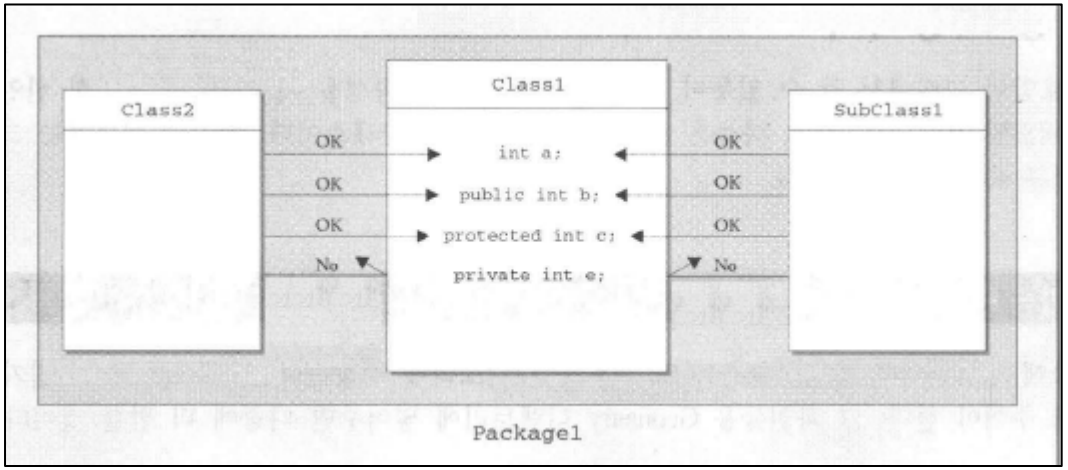
```
private     : 같은 클래스 내에서 접근 가능
default     : 같은 패키지 내에서 접근 가능
protected   : 같은 패키지 내, 다른 패키지 자손 클래스에서 접근 가능
public      : 접근 제한 없음
```

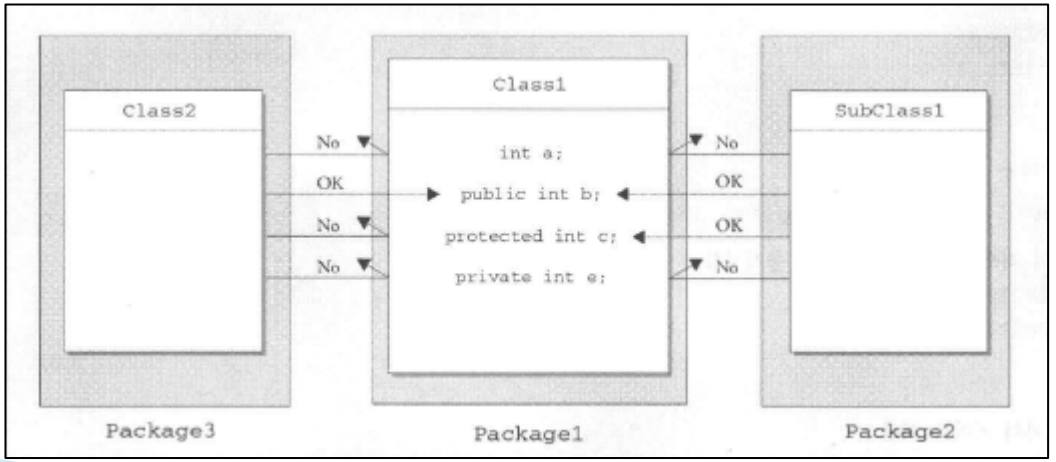
```
public > protected > default > private
```

제어자	같은 클래스	같은 패키지	자손클래스	전 체
public				
protected				
default				
private				

캡슐화 (Encapsulation)

- 접근(Access) 속성 (Attribute), 제어자 (Modifier), 지정자 (Identifier)





캡슐화 (Encapsulation)

- 접근(Access) 속성 (Attribute), 제어자 (Modifier), 지정자 (Identifier)

속성	허용되는 접근
접근 속성이 없는 경우	같은 패키지 안의 모든 클래스의 메소드에서 접근할 수 있다.
public	모든 패키지의 모든 클래스의 메소드에서 접근할 수 있다.
private	같은 클래스 안의 메소드에서만 접근할 수 있다. 클래스 외부에서는 접근할 수 없다.
protected	같은 패키지 안의 모든 클래스의 메소드와 모든 서브클래스에서 접근할 수 있다.

대 상	사용가능한 접근 제어자
클래스	public, (default)
메서드	public, protected, (default), private
멤버변수	
지역변수	없 음

캡슐화 (Encapsulation)

- 접근(Access) 속성 (Attribute), 제어자 (Modifier), 지정자 (Identifier)

```java
/**
public 메서드를 이용한 private 멤버 변수의 접근
**/
public class TopSecret{
    private int secret; //private 멤버 변수 선언
    //private 멤버에 값 할당하기
    public void setSecret(int x){ //private에 접근하는 public 멤버 메서드
        secret = x;
    }
    //private 멤버의 값을 외부로 내보내기
    public int getSecret(){ //private에 접근하는 public 멤버 메서드
        return secret;
    }
}
```

```java
public void setSecret(int x){
        secret = x;
 }
```

```java
public int getSecret(){
        return secret;
 }
```

```java
/**
TopSecret 클래스를 테스트하는 예
**/
public class TopSecretMain {
    public static void main(String[] args) {
        TopSecret t = new TopSecret();
        t.setSecret(1000); //private 멤버 변수에 값을 할당하는 메서드
        int s = t.getSecret(); //private 멤버 변수의 값을 얻어오는 메서드

        System.out.println("s의 값은: " + s); //s의 값 출력
        System.out.println("t.getSecret(): " + t.getSecret()); //t.getSecret()의 값 출력
    }
}
```

캡슐화 (Encapsulation)

- 접근 제어자 사용 이유
  - 외부로부터 Data를 보호하기 위해
  - 외부에는 불필요한, 내부적으로만 사용되는 부분을 감추기 위해

```
public class Time{
    private int hour;
    private int minute;
    private int second;

    public int getHour()    {   return hour;    }
    public int getMinute()  {   return minute;  }
    public int getSecond()  {   return second;  }
    public void setHour(int hour){
        if ( hour < 0 || hour > 23)
            return;
        this.hour = hour;
    }
    public void setMinute(int minute){
        if ( minute < 0 || minute > 59 )
            return;
        this.minute = minute;
    }
    public void setSecond (int second) {
        if ( second < 0 || second > 59 )
            return;
        this.second = second;
    }
}
```

```
public class Time {
    public int hour;
    public int minute;
    public int second;
}
```

```
Time t = new Time();
t.hour = 25;
```

캡슐화 (Encapsulation)

```
public class Time {
    private int hour;   private int minute; private int second;

    Time (int hour, int minute, int second) {
        setHour(hour);      setMinute(minute);      setSecond(second);
    }

    public int getHour()    {   return hour;    }
    public int getMinute()  {   return minute;  }
    public int getSecond()  {   return second;  }
    public void setHour(int hour){
        if ( hour < 0 || hour > 23)          return;
        this.hour = hour;
    }
    public void setMinute(int minute){
        if ( minute < 0 || minute > 59 )     return;
        this.minute = minute;
    }
    public void setSecond (int second) {
        if ( second < 0 || second > 59 )     return;
        this.second = second;
    }
    public String toString(){
        return hour + ":" + minute + ":" + second;
    }
}
```

```
12:35:30
13:35:30
```

```
public class TimeTest {
    public static void main(String[] args){
        Time t = new Time (12, 35, 30);
        System.out.println(t);
//      t.hour = 13;
        t.setHour(t.getHour() + 1);
        System.out.prinlnt(t);
    }
}
```

캡슐화 (Encapsulation)

- 생성자의 접근 제어자

```
class Singleton{
    //...
    private static Singleton = new Singleton();

    private Singleton(){
    //...
    }

    // 인스턴스를 생성하지 않고도 호출할 수 있어야하므로
    // static 이어야함
    public static Singleton getInstance(){
        return s;
    }
    //...
}
```

```
class Singleton{
    private Singleton(){
        //...
    }
    //...
}
```

캡슐화 (Encapsulation)

- 생성자의 접근 제어자

```
final class Singleton{
    private static Singleton = new Singleton();

    private Singleton(){
        //...
    }

    public static Singleton getInstance(){
        if ( s == null ){
            s = new Singleton();
        }
        return s;
    }
    //...
}

class SingletonTest{
    public static void main(String args[]){
//      Singleton s new Singleton();
        Singleton s1 = Singleton.getInstance();
    }
}
```

캡슐화 (Encapsulation)

- 제어자의 사용 범위 및 주의점

대 상	사용가능한 제어자
클래스	public, (default), final, abstract
메서드	모든 접근 제어자, final, abstract, static
멤버변수	모든 접근 제어자, final, static
지역변수	final

Method 에 static 과 abstract 함께 사용 불가
- static Method 는 몸통이 있는 Method 에만 사용 가능

Class 에 abstract 와 final 동시 사용 불가
- Class 에 사용되는 final 은 Class 를 확장할 수 없다는 의미
  abstract 는 상속을 통해 완성되어야한다는 의미로 상반

abstract Method 의 접근 제어자는 private 불가
- abstract Method 는 자식 Class 에서 구현해주어야 하므로
 접근 제어자가 private 가 되면 자식 Class 접근 불가

Method 에 private 과 final 동시 사용 불가
- 접근 제어자가 private 인 Method 는 Overriding 이 불가하므로
 둘 중 하나만 사용해도 충분

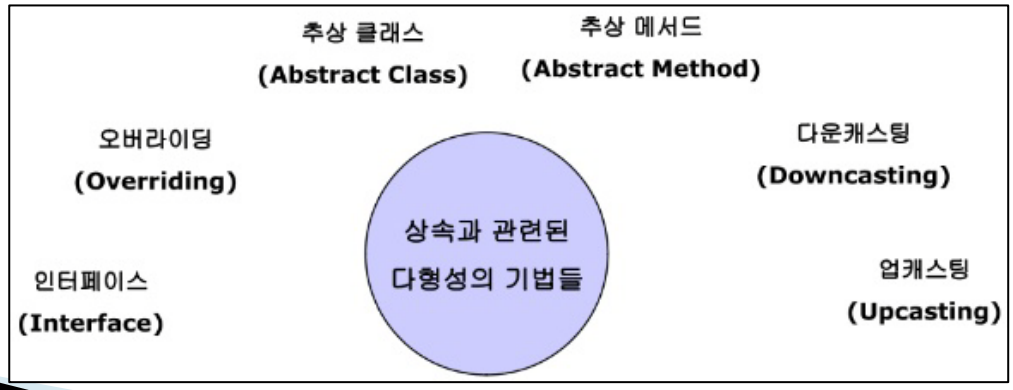
다형성 (Polymorphism)

- Polymorphism의 사전적 의미

polymorphism [pàlimɔ́:rfizm, pɔ̀l-]

1. [명사](결정) 동질 이상(同質異像)
2. [명사] [생물] 다형(多形)(현상), 다형성; [유전] 다형 현상 ((동종 집단 가운데에서 2개 이상의 대립 형질이 뚜렷이 구별되는 것; 사람의 ABO식 혈액형 등))

  - “여러 가지 형태를 가질 수 있는”



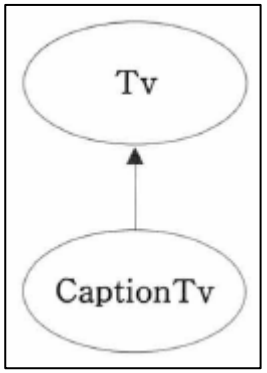
다형성 (Polymorphism)

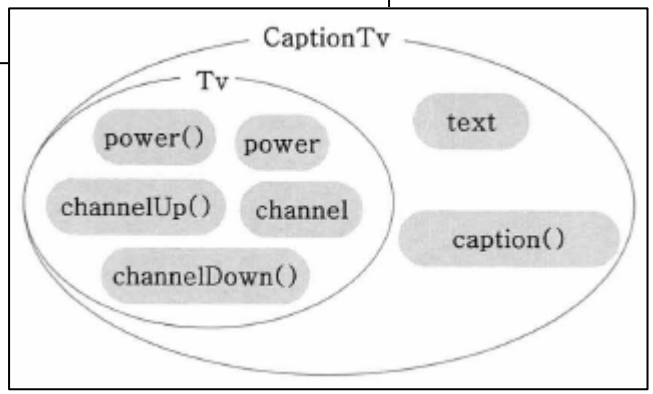
- 주어진 타입의 변수 하나가 여러 타입의 객체를 참조
  - 부모 클래스 타입의 참조변수로 자식 클래스의 인스턴스 참조가 가능
- 변수가 참조하는 객체의 타입에 맞는 메소드를 자동 호출
- 메소드 호출 하나가 호출이 적용되는 객체의 타입에 따라 서로 다르게 작동하게 할 수 있다

다형성 (Polymorphism)

- 다형성은 상속에서 시작

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

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

class CaptionTv extends Tv {
    String text;     // 캡션을 보여주기 위한 문자열
    void caption { /* . . . */}
}
```





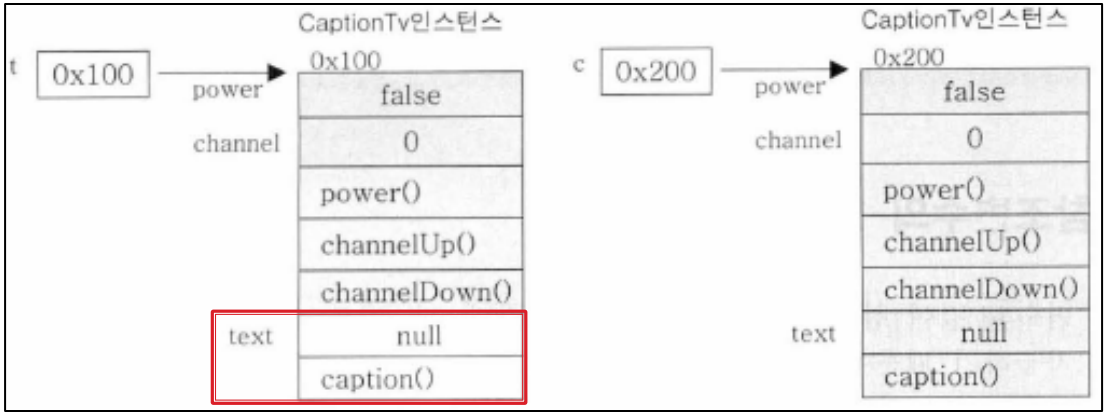
다형성 (Polymorphism)

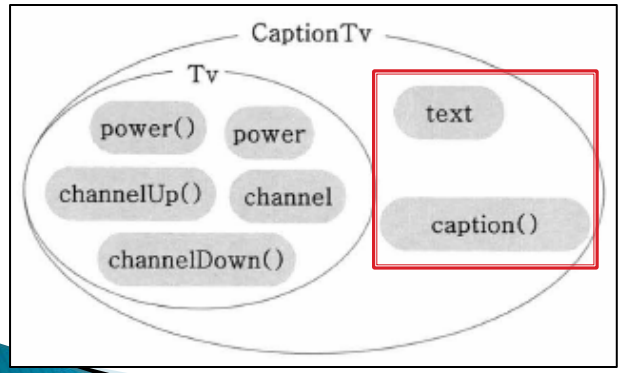
- 부모 클래스의 참조변수를 자식 클래스의 인스턴스로..

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

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

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





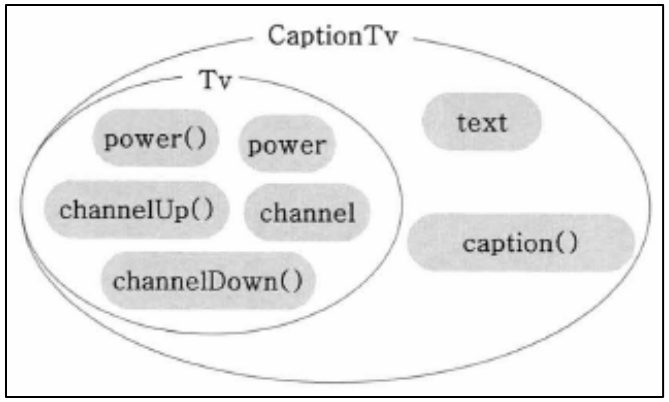

```
class Tv {
    boolean power   ;
    int     channel ;

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

class CaptionTv extends Tv {
    String text;     // 캡션을 보여주기 위한 문자열
    void caption { /* . . . */}
}
```

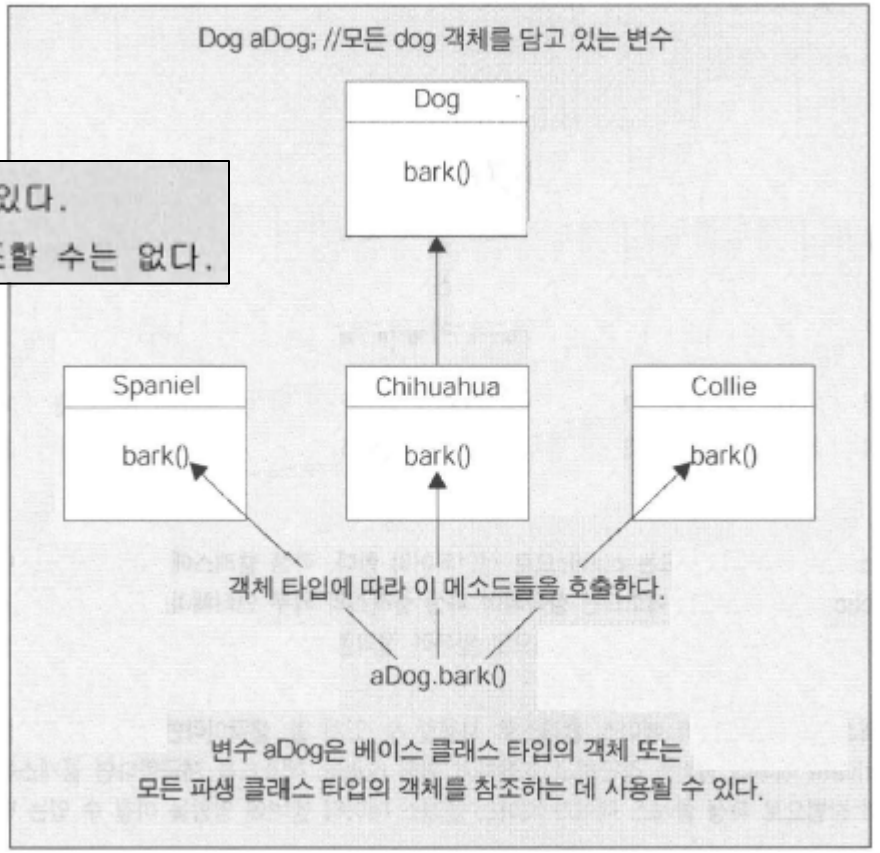
다형성 (Polymorphism)

- 거꾸로는? 안된다

```
CaptionTv c = new Tv(); // Compile Error!
```

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

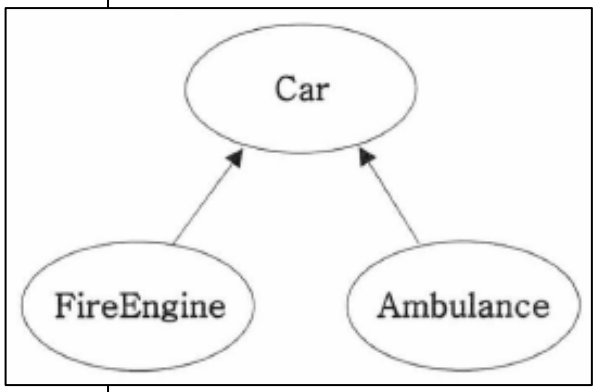




다형성 (Polymorphism)

- 참조변수에서의 형변환 (더 크다의 의미를 정확히)

```
class Car {
    String color;
    int door;
    void drive() {  // 운전
        System.out.println("drive, Brrr~");
    }
    void stop() {   // 정지
        System.out.println(("stop!!!");
    }
}

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

class Ambulance extends Car {    // 엠뷸런스
    void siren() {               // 사이렌 울리기
        System.out.println("siren~~~");
    }
}
```

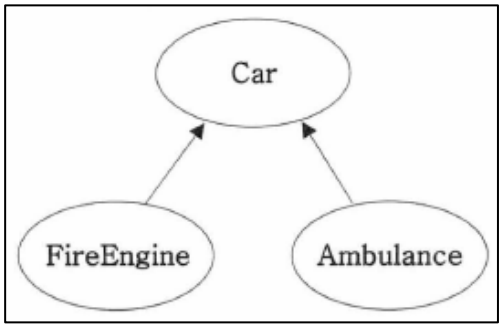


자손타입 → 조상타입 (Up-casting) : 형변환 생략가능
자손타입 ← 조상타입 (Down-casting) : 형변환 생략불가

다형성 (Polymorphism)

- 참조변수에서의 형변환

```
FireEngine f;
Ambulance a;

a = (Ambulance) f;  // Compile Error!
f = (FireEngine) a; // Compile Error!
```




```
Car car = null;
FireEngine fe = new FireEngine();
FireEngine fe2 = null;

car = fe;   // car = (Car) fe; 에서 형변환이 생략
fe2 = (FireEngine) car; // 형변환 생략 불가
```

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

다형성 (Polymorphism)

- Example

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

        fe.water();
        car = fe; // (Car) fe 형변환 생략
//      car.water();    //  Compile Error!
        fe2 = (FireEngine) car; // 자손타입 <- 조상타입
        fe2.water();
    }
}
```

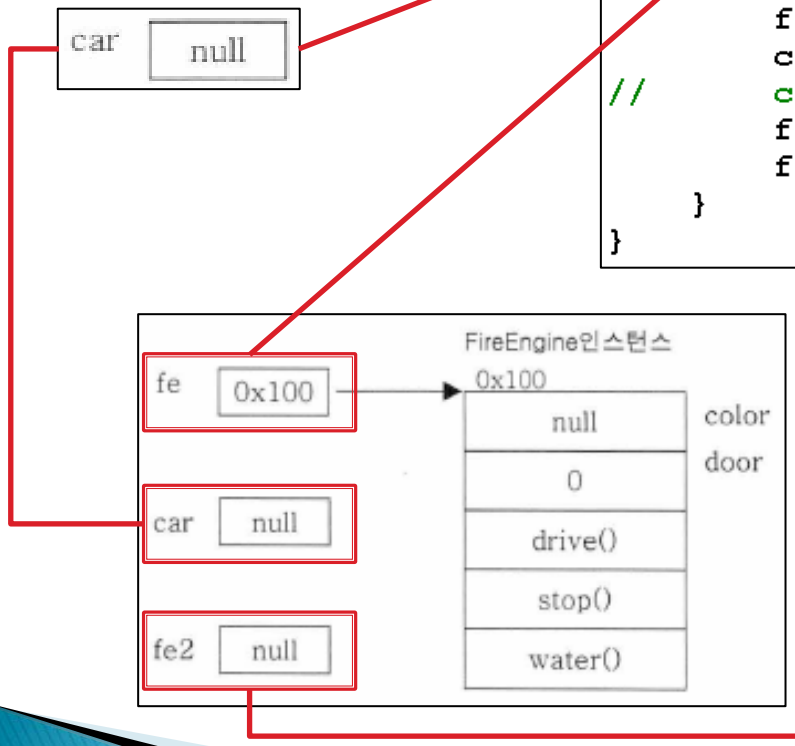
```
water!!!
water!!!
```

```
class Car {
    String color;
    int door;
    void drive() {  // 운전
        System.out.println("drive, Brrr~");
    }
    void stop() {   // 정지
        System.out.println(("stop!!!");
    }
}

class FireEngine extends Car {   // 소방차
    void water() {                // 물 뿌리기
        System.out.println("water!!!");
    }
}
```

다형성 (Polymorphism)

- Example

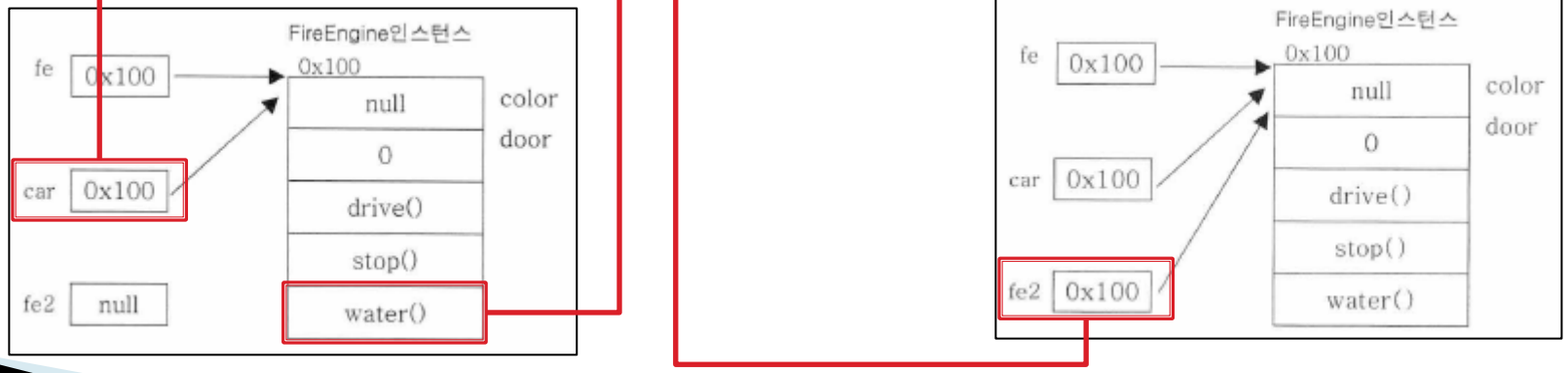
```
class CastingTest1 {
    public static void main(String args[]){
        Car car = null;
        FireEngine fe = new FireEngine();
        FireEngine fe2 = null;

        fe.water();
        car = fe; // (Car) fe 형변환 생략
//      car.water();    //  Compile Error!
        fe2 = (FireEngine) car; // 자손타입 <- 조상타입
        fe2.water();
    }
}
```



다형성 (Polymorphism)

- Example

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

        fe.water();
        car = fe; // (Car) fe 형변환 생략
//      car.water();    //  Compile Error!
        fe2 = (FireEngine) car; // 자손타입 <- 조상타입
        fe2.water();
    }
}
```



다형성 (Polymorphism)

- Example

```
class CastingTest2 {
    public static void main(String args[]){
        Car car = new Car();
        Car car2 = null;
        FireEngine fe = null;

        car.drive();
        fe = (fireEngine) car; // 실행시 Error
        fe.drive();
        car2 = fe;
        car2.drive();
    }
}
```

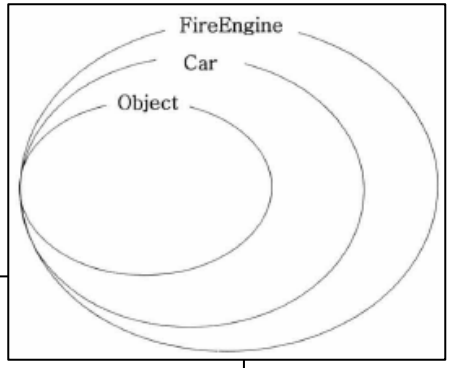
```
drive, Brrrr~
java.lang.ClassCastException: Car
    at CastingTest2.main(CastingTest2.java:8)
```

캐스트연산자를 사용하면 서로 상속관계에 있는 클래스 타입의 참조변수간의 형변환은 양방향으로 자유롭게 수행될 수 있다.
그러나 참조변수가 참조하고 있는 인스턴스의 자손타입으로 형변환을 하는 것은 허용되지 않는다.

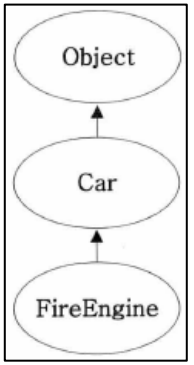
다형성 (Polymorphism)

- instanceof

```
if (c instanceof FireEngine){ // c는 Car타입 변수
    FireEngine fe = (FireEngine) c;
    fe.water();
    //...
}
```




```
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 a Object instance.");
        }
    }
}
```




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

어떤 타입에 대한 instanceof연산의 결과가 true라는 것은 검사한 타입으로 형변환이 가능하다는 것을 뜻한다.

다형성 (Polymorphism)

- 그럼 변수는?

```
class BindingTest {
    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();
    }
}

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 BindingTest2 {
    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();
    }
}

class Parent {
    int x = 100;
    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {    }
```

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

다형성 (Polymorphism)

- 그럼 변수는?

```
p.x = 100
x=200
super.x=100
this.x=200

c.x = 200
x=200
super.x=100
this.x=200
```

```
class BindingTest3 {
    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();
        System.out.println("c.x = " + c.x);
        c.method();
    }
}
```

```
class Parent {
    int x = 100;
    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    int x = 200;
    void method(){
        System.out.println("x = " + x); // this.x와 같다
        System.out.println("super.x = " + super.x);
        System.out.println("this.x = " + this.x);
    }
}
```

다형성 (Polymorphism)

- 매개변수에 매우 유용하다

```
class Product {
    int price;       // 제품의 가격
    int bonusPoint; // 제품구매 시 제공하는 보너스
}

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

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

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

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

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

```
void buy(Tv t){
    // Buyer 소지금에서 제품 가격을 뺌
    money = money - t.price

    // Buyer의 보너스에 제품의 보너스를 더함
    bonusPoint = bonusPoint + t.bonusPoint;
}
```

```
Buyer b = new Buyer();
Tv t = new Tv();
Computer c = new Computer();
b.buy(t);
b.buy(c);
```

다형성 (Polymorphism)

```
class Product {
    int price;       // 제품 가격
    int bonusPoint; // 보너스

    Product(int price){
        this.price = price;
        bonusPoint = (int) (price/10.0); // 10%
    }
}
```

```
class Tv extends Product {
    Tv() {
        // 부모클래스 생성자호출
        super(100); // Tv 가격 100
    }

    // Object Class의 toString() Overriding
    public String toString(){
        return "Tv";
    }
}
```

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

다형성 (Polymorphism)

```
class PolyArgumentTest {
    public static void main(String args[]){
        Buyer b = new Buyer();
        Tv tv = new Tv();
        Computer com = new Computer();

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

        System.out.println("현재 남은 돈은 " + b.money);
        System.out.println("현재 보너스는 " + b.bonusPoint);
    }
}
```

```
class Buyer {              // 물건 사기
    int money = 1000;      // 소유 금액
    int bonusPoint = 0; // 보너스

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

        money -= p.price;    // 소유 금액에서 물건가 빼기
        bonusPoint += p.bonusPoint; // 보너스 추가
        System.out.println(p + " 구입");
    }
}
```

```
Tv을/를 구입하셨습니다.
Computer을/를 구입하셨습니다.
현재 남은 돈은 700만원입니다.
현재 보너스점수는 30점입니다.
```

다형성 (Polymorphism)

- 상속관계 내 여러 종류의 객체를 배열에 집어넣자

```
class Buyer {
    int money = 1000;
    int bonusPoint = 0;
    Product[] item = new Product[10]; // 구입 제품 저장용 배열
    int i = 0; // Product배열에 사용될 index

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

        money -= p.price;    // 소유 금액에서 물건가 빼기
        bonusPoint += p.bonusPoint; // 보너스 추가
        item[i++] = p;       // 제품 객체를 저장
        System.out.println(p + " 구입");
    }
}
```

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

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

오늘 숙제

- 개념!