01.12 木 _ 다형성

⭐ `참조 변수`가 사용되는 이유?

생성된 객체를 가르키고, 그 객체의 멤버들에 접근하기 위해서

`참조 변수`는 자기 자신의 자료형과 동일한 객체를 가르킬 수 있다 (기본 원칙)

`참조 변수` + `다형성` ⇒ ⭐ 자식 객체도 가르킬 수 있다 ⭐

다형성 (多形性, polymorphism)

> 다양한 형태의 성질을 받아 들일 수 있다

⭐ `다형성` 이 사용되는 이유?

여러 종류의 class들의 객체를 **부모의 형으로 묶어서 관리**할 수 있기 때문이다.

「 다형성과 함께 따라오는 2가지 개념 」

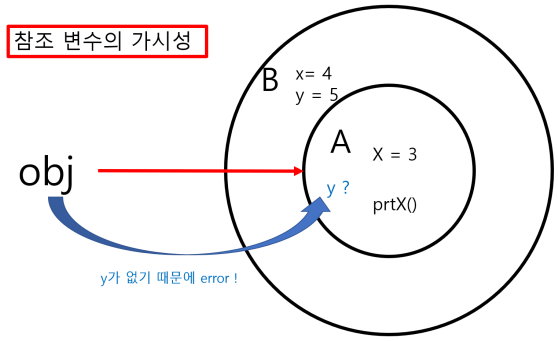
1. Visibility of a reference variable (참조 변수의 가시성)
2. Dynamic binding for an overrided method (Overriding된 메소드의 동적 바인딩)

---

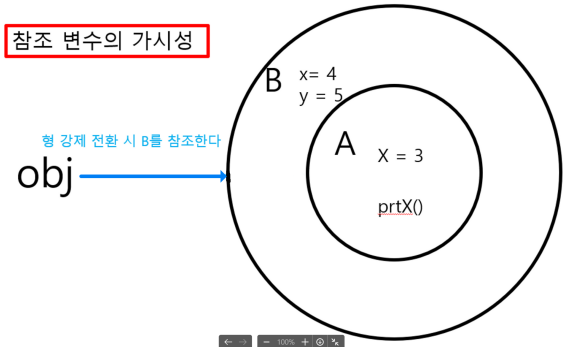
1. 참조 변수의 가시성 문제

```
package test;

class A {
  int x = 3;
```

```
  void prtX() {
    System.out.println(x);
  }
}

class B extends A {
  int x = 4;
  int y = 5;
}

public class TEST1 {

public static void main(String[] args) {


      「 다형성이 적용된 참조 변수는 A를 가르키고 있다 」

    A obj = new B();      // 다형성 : obj에 B 클래스의 객체 주소를 집어 넣는다
 부모 형       자식형       원래 B클래스의 객체를 가르키기 위한 자료형은 B 여야 한다


    System.out.println(obj.x);   //  결과 : 3
    System.out.println(((B)obj).y);    ☆참조 변수의 자료형을 강제적으로 형 변환 시킨다☆
  }
```



System.out.println(obj.y);



System.out.println(((B)obj).y);

객체가 실질적으로 **B** 까지 다 만들어졌다 하더라도
`참조 변수`의 자료형에 따라서 다형성이 적용되면
그 `참조 변수`의 자료형을 가르키게 된다 ⇒ ⭐ `참조 변수`의 자료형 까지만 볼 수 있다 ⭐

⭐ 참조 변수는 `메모리 주소 값` , `자료형` 을 가진다

⭐ **다형성**의 장단점 ⭐

부모의 형으로 다 묶어서 다른 자료형의 객체를 하나의 자료형으로 관리를 하지만 (장점)
그 참조 변수를 이용해서 **접근할 수 있는 멤버들은 부모 형**밖에 안 보인다는 것! (단점)

➡️ 해결 방법 : ⭐ 참조 변수의 자료형을 **강제적으로 형 변환** 시킨다!! ⭐

instanceof (이항 연산자)

> 현재 내가 가르키고 있는 객체가 무슨 class의 형인가?
> 참조 변수가 참조하고 있는 인스턴스의 실제 타입을 알아보기 위해 사용
> 주로 조건문에 사용

- 형태 : 참조 변수 **instanceof** **타입 (class 이름)** → 다형성이 적용 되어야 사용 가능
- 해석 : 참조 변수**가** **가르키는 객체가 class 이름**의 **객체니**?

  ⇒ `true` or `false` 을 return

```
package test

class Terran { int hp; }
class Scv extends Terran { int mineral;}
class Tank extends Terran { boolean sMode; }
class Marine extends Terran { boolean sPack; }

public class TEST2 {
                                                    참조 변수 , 매개 변수 , 메소드의 반환형
  static void prtStatusOfSMode(Terran argObj) {     ☆ 매개 변수에 다형성을 적용! ☆

      // 각 클래스별 S mode값을 출력하라.     =>     S mode는 전부 자식들에 있어 못 본다!
        if(argObj instanceof Scv) {
```

```
            System.out.println(((Scv)argObj).mineral);

        } else if (argObj instanceof Tank) {          가시성은 전부 Terran  밖에 못 본다!

              System.out.println(((Tank)argObj).sMode);

        } else if (argObj instanceof Marine) {

              System.out.println(((Marine)argObj).sPack);
        }
  }

  public static void main(String[] args) {

        Scv o1 = new Scv();
        Marine o2 = new Marine();
        Tank o3 = new Tank();

        TEST2.prtStatusOfSMode(o1);
        TEST2.prtStatusOfSMode(o2);
        TEST2.prtStatusOfSMode(o3);
  }
}
```

override = “덮어 쓴다”

`overloading` - 기존에 없는 새로운 메서드를 정의하는 것

동일한 반환 형과 동일한 이름을 가지는 Method나 함수가 같은 영역에 여러 개 존재 할 수가 있고
이를 호출될 때 입력되는 인자의 유형에 따라서 호출

`overriding` - 상속 받은 메서드의 내용을 변경하는 것

- **overriding 은 상속을 반드시 가지고 나와야 한다.**
- **매개변수가 완전 똑같다**
- 부모에서 Method를 만들어 놓으면 내가 상속을 받으면서 그 Method를 다 사용할 수도 있지만
  아예 안 맞을 수도 있다 → **再정의**를 할 수 있어야 한다 ⇒ `overloading`

`overloading`의 2가지 방법

1. 완전 재정의 (부모 꺼 무시 하고 내가 새롭게 재정의)
2. 부분 재정의 (부모로부터 물려 받은 것 쓰긴 쓸꺼야.
   근데 내가 원하는 거 추가 할래) ⇒ 부모가 추가되고 이어서 내 것을 추가하는 것

```
class A {
  int x = 3;
  void prtX() {
    sout("prtX of A is invoked");
  }
}

class B extends A {
  int x = 5;
  void prtX() {
    sout("prtX of B is invoked");
  }
}

public class Main {
  public static void main(String args[]) {
    new B().prtX();  // prtX of B is invoked
  }
}
```

`super` (부분 재정의)

현재 부모의 객체를 가르킨다.

> 자손 class에서 조상 class로부터 상속 받은 멤버를 참조하는데 사용되는 참조 변수.
> 상속 받은 멤버와 자신의 멤버와 이름이 같을 때는 `super`를 붙여서 구별할 수 있다.

super = 부모 객체의 메모리 주소 값

```
package test;

class A {
  int x = 3;
```

```
  void prtX() {
    System.out.println("prtX of A is invoked");
  }
}
class B extends A {
  int x = 5;

  void prtX() {    // 부모 메서드에 정이되어진 메서드와 동일 !!
    super.prtX();  /*  부모로 부터 물려 받은 것 사용하고 */
    System.out.println("prtX of B is invoked"); /* 이어서 내것도 추가   */
  }
}


public class TEST2 {

  public static void main(String[] args) {

    new B().prtX(); // 참조 변수는 없지만 넘어오는 값은 B 객체를 가르킨다
                            // B객체에 와서 prtX()를 찾는다
  }
}
```

단순한 Overriding 예시

```
package test;

class A {
  void prtSomething() {
    System.out.println("A");
  }
}

class B extends A {
  void prtSomething() {
    System.out.println("B");
  }
}

public class TEST2 {
  public static void main(String[] args) {

    B obj = new B();
    obj.prtSomething();    // =>  B
  }
}
```

---

2. Overriding된 메소드의 동적 바인딩

동적 바인딩

- 프로그램이 실행되고 나서 결정되는 것

정적 바인딩

- 프로그램이 실행 전에 다 결정되어있는 것

오버 라이딩된 메소드에 동적 바인딩이 적용

> Overriding 되어있는 Method를 호출하면 참조 변수의 자료형과 상관 없이 자식 class의 멤버를 찾음

```
package test;

class A {
  void prtSomething() {
    System.out.println("A");
  }
}

class B extends A {
  void prtSomething() {
    System.out.println("B");
  }
}

public class TEST2 {

  public static void main(String[] args) {

    A obj = new B();  // 반대로 안에서 밖으로 찾아 나온다
    obj.prtSomething();
                                // 참조 변수의 자료형과 상관 없이
  }

}
```

원래 자기 영역을 찾아서 없으면 부모 쪽(안 쪽) 으로 파고 들어간다

Overriding 이되면 반대로 안에서 부터 바깥쪽으로 찾아가게 실행한다

동적 바인딩을 사용한 Overriding

```
class Terran {
    void prtSMode() {}       // 동적바인딩을 하기 위해 선언해놓은 메소드
}

class Scv extends Terran {
    int mineral = 20;
    void prtSMode() {
        System.out.println(mineral);
    }

}

class Marine extends Terran {
    boolean sPack = false;
    void prtSMode() {
        System.out.println(sPack);
    }
}

public class Starcraft {
    public static void main(String[] args) {
        Terran [] unitList = new Terran [200];

        unitList[0] = new Scv();
        unitList[1] = new Marine();

        unitList[0].prtSMode();
        unitList[1].prtSMode();
    }
}
```

제일 마지막에 overriding 되어있는 것을 찾는다

```
package Test;

class A {
  void prtX() {System.out.println("A");}
}

class B extends A {

}

class C extends B {
  void prtX() {System.out.println("C");}
}

public class MyProject {
  public static void main(String args[]) {
```

```
package test;

class A {
  int x = 2;

  void prtX() {
    System.out.println("A");
  }
}

class B extends A {
  int x = 4;
}

class C extends B {
  void prtX() {
```

```
    A obj = new C();

    obj.prtX();  // 결과 :  C
  }
}
```

중간에서 끊겼다고 해서 끊기는게 아니다.

일단은 계속 제일 마지막 상속을 받은 놈 까지 간다.

그리고 제일 마지막에 overriding것을 찾는다.

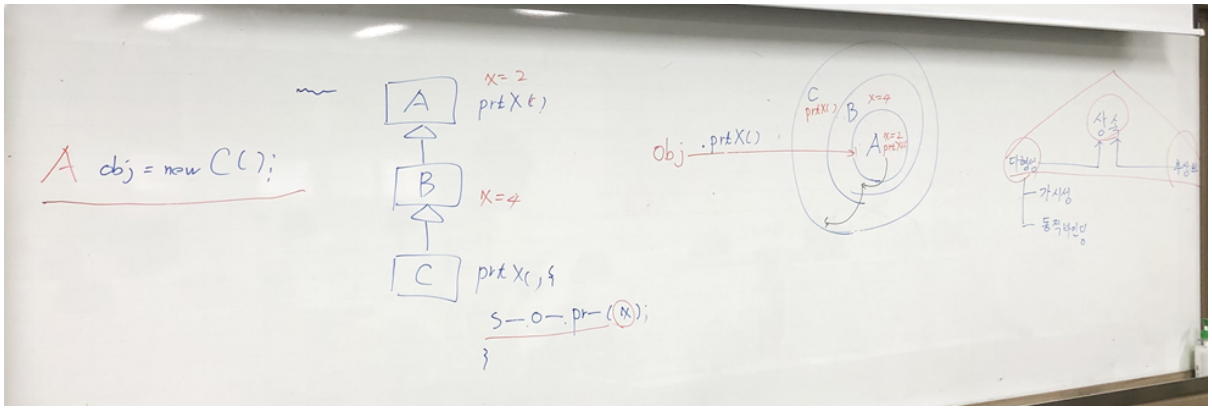
overriding 된 메소드인 경우 반대로 내려간다

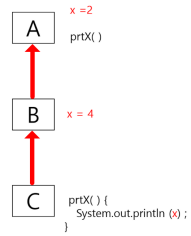
```
    System.out.println(x);
  }
}

public class TEST5 {

  public static void main(String[] args) {
    A obj = new C();

    obj.prtX();   // 4

  }
}
```

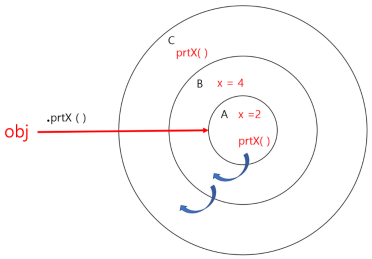
Overriding된 다형성

- 상속
- 메소드

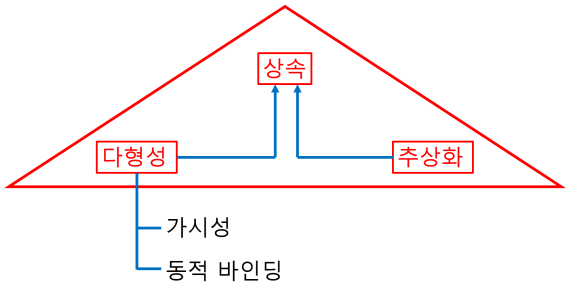
→ 다형성의 단점을 보완한다




A
x =2
prtX( )
B
x = 4
C
prtX( ) {
System.out.println (x) ;
}


obj
.prtX ( )
C
prtX( )
B
x = 4
A
x =2
prtX( )

A obj = new C ( ) ;


상속
다형성
추상화
가시성
동적 바인딩