제 8 장 상속 (Inheritance) Part-2

객체의 타입 변환

- 자바의 Type checking은 엄격한 편이다
  - 클래스 A의 참조 변수로 클래스 B의 객체를 참조할 수는 없다

```
class A {
        A() { }
}

class B {
        B() { }
}

public class TypeTest1 {
        public static void main(String args[]) {
                A a = new B(); // NO!
        }
}
```

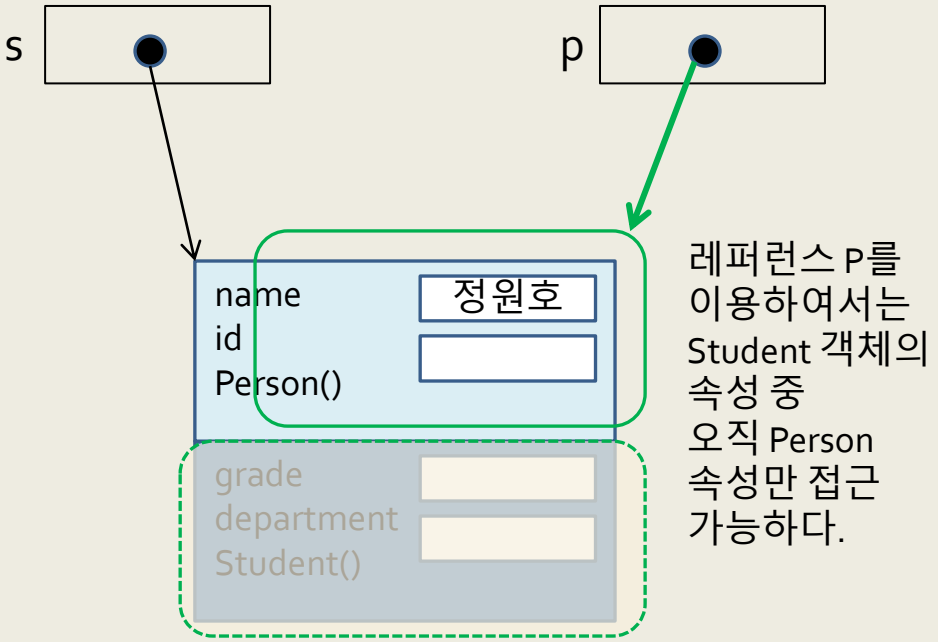
  - **그러나** 부모 클래스의 참조 변수는 자식 클래스의 객체를 참조할 수 있다
  - 이러한 현상은 타입변환(type casting)을 통해 가능하다
    - 타입 변환은, 상속 관계 혹은 (추후 다룰 인터페이스인 경우) 구현 관계가 성립하는 객체 간에 있어서, 어떤 타입의 위치에 다른 타입의 객체를 사용하는 것에 대한 자바의 규칙이다.

- 업캐스팅(upcasting) – 상향 형변환, 자동변환
  - 컴파일러에 의해 이루어지는 자동 타입 변환
  - 서브 클래스의 레퍼런스 → 슈퍼 클래스 레퍼런스에 대입
    - 상위클래스-레퍼런스 = 하위클래스-레퍼런스
    - 상위 클래스 레퍼런스가 하위 클래스 객체를 가리키게 되는 현상
    - 객체 내에 있는 모든 멤버를 접근할 수 없고 상위 클래스의 멤버만 접근 가능

```
class Person {
}

class Student extends Person {
}

Student s = new Student();
Person p = s;          // 업캐스팅, 자동타입변환
```

업캐스팅 예

```
class  Person {
    String name;
    String id;

    public Person(String name) {
        this.name = name;
     }
}

class  Student extends Person {
    String grade;
    String department;

    public Student(String name) {
        super(name);
     }
}

public class UpcastingEx {
    public static void main(String[] args) {
        Person  p;
        Student s = new Student("정원호");
        p = s; // 업캐스팅 발생

        System.out.println(p.name); // 오류 없음

        p.grade = "A"; // 컴파일 오류
        p.department = "Com"; // 컴파일 오류
    }
}
```



레퍼런스 P를 이용하여서는 Student 객체의 속성 중 오직 Person 속성만 접근 가능하다.

정원호

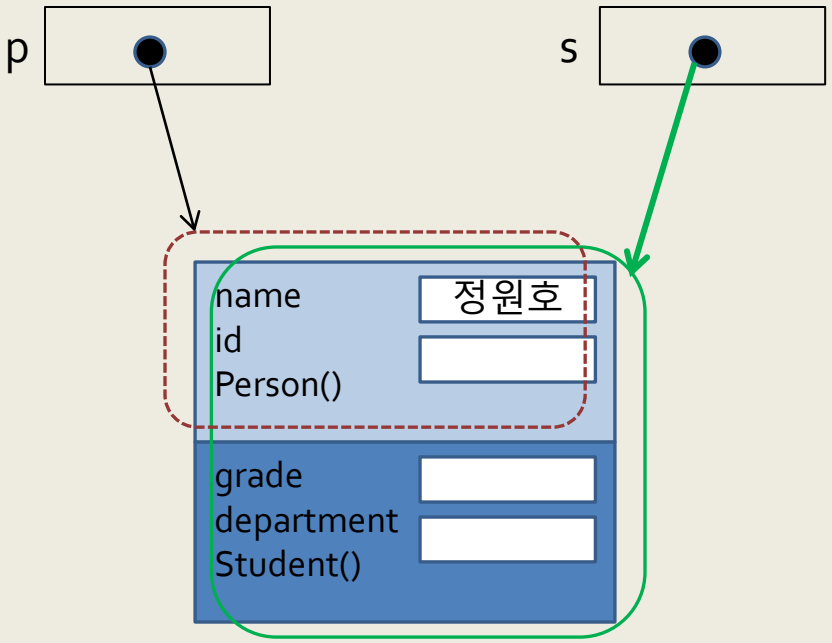
- 다운캐스팅(downcasting)
  - 슈퍼 클래스 레퍼런스 → 서브 클래스 레퍼런스에 대입
  - 업캐스팅 된 것을 다시 원래대로 되돌리는 것
  - 명시적으로 타입 지정

```
class Person {
}
class Student extends Person {
}

Student s = (Student) p;          // 다운캐스팅, 강제타입변환
```

다운캐스팅 사례

```
public class DowncastingEx {
    public static void main(String[] args) {
        Person  p = new Student("정원호"); //
업캐스팅 발생
        Student s;

        s  = (Student) p; // 다운캐스팅

        System.out.println(s.name); // 오류 없음
        s.grade = "A"; // 오류 없음
    }
}
```

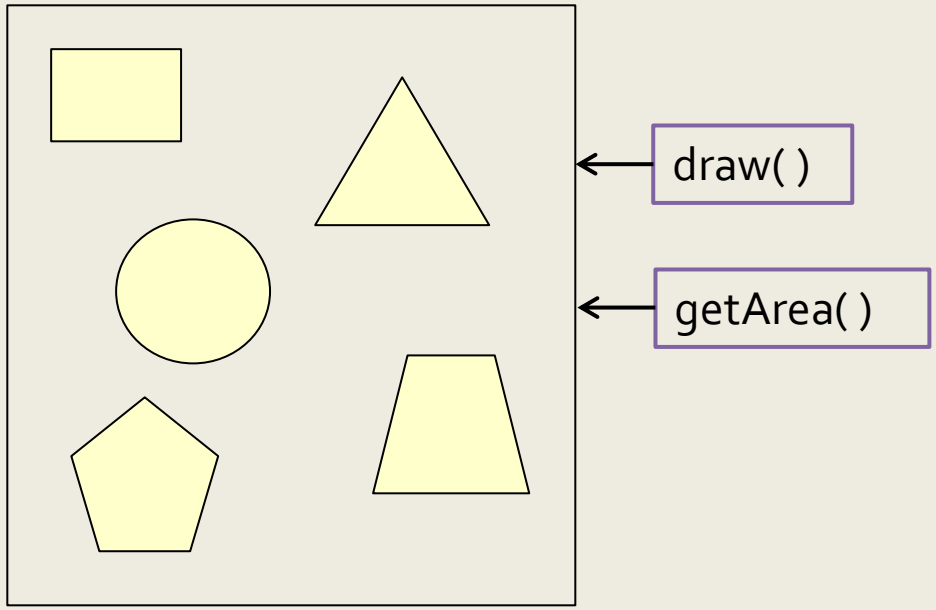
정원호



upcast 레퍼런스가 존재하는 경우, 새로 객체를 생성하지 않고 upcast 레퍼런스를 downcast하여 그대로 사용할 수 있다

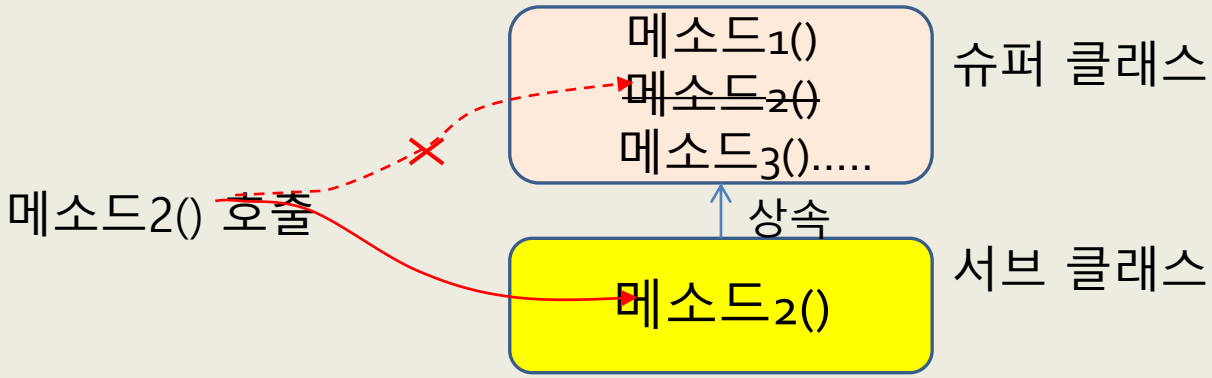
상속과 다형성

- 다형성(polymorphism)이란 하나의 동일한 메시지(시그너쳐+리턴 타입)에 대해, 서로 다른 유형의 객체들이 서로 다른 동작을 수행하는 개념
- 하나의 코드로 다양한 유형의 객체를 처리할 수 있도록 하는 유용한 기술 → 다형성은 ***상속 (메소드 오버라이딩)*** 을 통해서 구현할 수 있다



메소드 오버라이딩

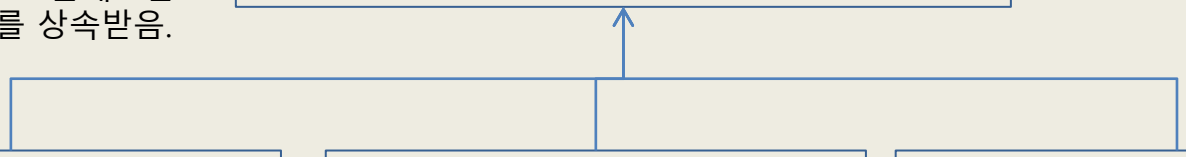
- 메소드 오버라이딩(Method Overriding)
  - 자바의 다형성을 보여주는 특성
  - 슈퍼 클래스의 메소드를 서브 클래스에서 재정의하는 것
    - 슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수(메소드 시그너쳐), 리턴 타입 등 모든 것 동일하게 정의
      - 이 중 하나라도 다르면 메소드 오버라이딩 실패
  - 슈퍼 클래스의 "메소드 재정의"로 번역되기도 함
    - 동적 바인딩 발생
    - 오버라이딩 된 메소드가 실행되도록 **동적 바인딩(dynamic binding)** 됨



메소드 오버라이딩 예

```
class DObject {
    public DObject next;

    public DObject() { next = null;}
    public void draw() {
        System.out.println("DObject draw");
    }
}
```

Line, Rect, Circle 클래스는 모두 DObject를 상속받음.



```
class Line extends DObject {
    public void draw() {
        System.out.println("Line");
    }
}
```

```
class Rect extends DObject {
    public void draw() {
        System.out.println("Rect");
    }
}
```

```
class Circle extends DObject {
    public void draw() {
        System.out.println("Circle");
    }
}
```

메소드 오버라이딩 조건

1. 반드시 슈퍼 클래스 메소드와 동일한 이름, 동일한 호출 인자, 반환 타입을 가져야 한다.
2. 오버라이딩된 메소드의 접근 지정자는 슈퍼 클래스의 메소드의 접근 지정자 보다 좁아질 수 없다.

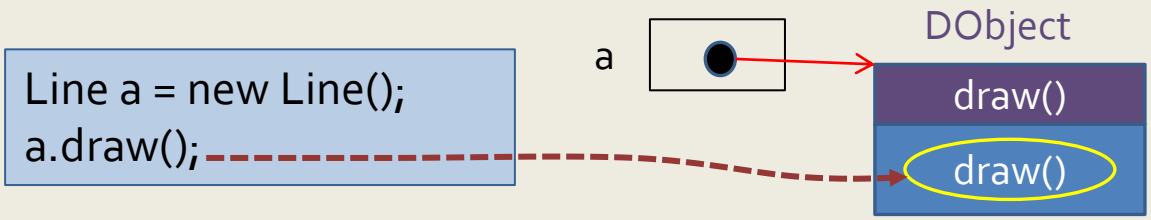
   public > protected > private 순으로 지정 범위가 좁아진다.
3. 반환 타입만 다르면 오류
4. static, private, 또는 final 메소드는 오버라이딩 될 수 없다.

   (static 메소드의 오버라이딩은 메소드 " 은폐 " 라고 하며 오버라이딩과 다른 개념이다)

```
class Person {
        String name;
        String phone;
        static int ID;

        public void setName(String s) {
                name = s;
        }
        public String getPhone() {
                return phone;
        }
        public static int getID() {
                return ID;
        }
}
class Professor extends Person {
        protected void setName(String s) { // 2번 조건위배
        }
        public String getPhone() {              // 1번 조건 성공
                return phone;
        }
        public void getPhone(){                 // 3번 조건 위배
        }
        public int getID() {// 4번 조건 위배
        }
}
```

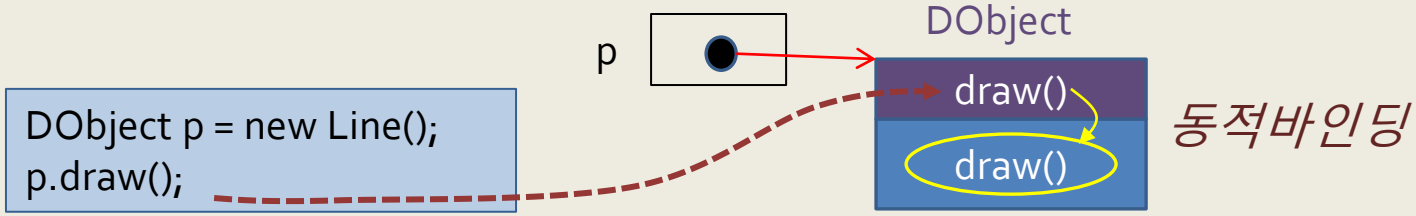
서브 클래스 객체와 오버라이딩된 메소드 호출

(1) 서브 클래스 레퍼런스로 오버라이딩된 메소드 호출



실행 결과 : Line이 출력됨

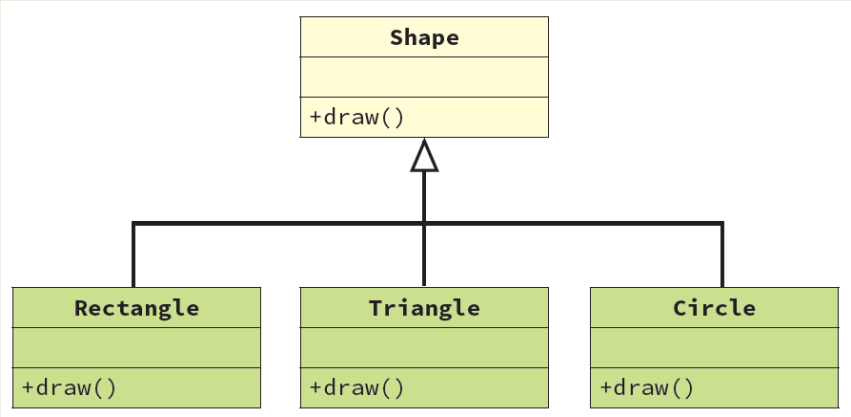
(2) 업캐스팅에 의해 슈퍼클래스 레퍼런스로 오버라이딩된 메소드 호출(동적 바인딩)



실행 결과 : Line이 출력됨

동적 바인딩(dynamic binding)

- 다형성은 객체들이 동일한 메시지를 받더라도 각 객체의 타입에 따라서 서로 다른 동작을 하는 성질
- 다형성을 나타낼 수 있는 것은 자바의 동적 바인딩 특성 때문..
- 동적바인딩 :
  - 메소드 호출을 호출될 실제 메소드와 연결하는 것을 바인딩 **(binding)**이라고 한다.
- 자바 가상 머신(JVM)은 실행 단계에서 객체의 타입을 보고 적절한 메소드를 호출하게 된다. 이것을 동적 바인딩**(dynamic binding)** 또는 가상 메소드 호출**(virtual method invocation)**이라고 한다.



동적 바인딩 예-1

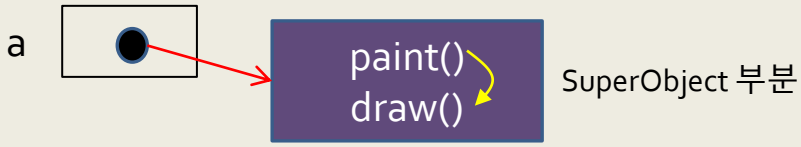
보통의 경우

```
public class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
    public static void main(String [] args) {
        SuperObject a = new SuperObject();
        a.paint();
    }
}
```

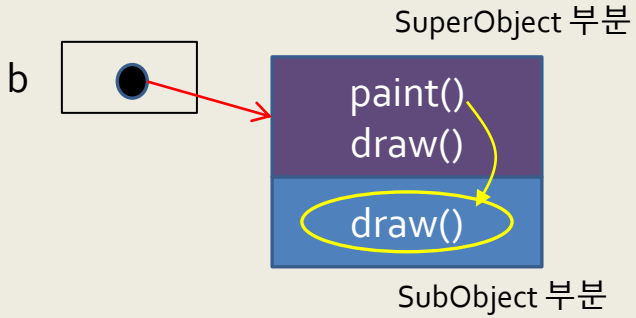
```
class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
}
public class SubObject extends SuperObject {
    public void draw() {
        System.out.println("Sub Object");
    }
    public static void main(String [] args) {
        SuperObject b = new SubObject();
        b.paint();
    }
}
```

동적바인딩

Super Object



Sub Object



동적 바인딩 예-2

```
class Shape {
   protected int x, y;
   public void draw() {
            System.out.println("Shape Draw");
   }
}

class Rectangle extends Shape {
   private int width, height;
   public void draw() {
            System.out.println("Rectangle Draw");
   }
```

```
class Triangle extends Shape {
   private int base, height;
   public void draw() {
            System.out.println("Triangle Draw");
   }
}

class Circle extends Shape {
   private int radius;

   public void draw() {
            System.out.println("Circle Draw");
   }
}
```

```
public class ShapeTest {
  public static void main(String arg[]) {
  Shape s1, s2, s3, s4;

  s1 = new Shape();
  s2 = new Rectangle();
  s3 = new Triangle();
  s4 = new Circle();

  s1.draw();
  s2.draw();
  s3.draw();
  s4.draw();
  }
}
```

```
Shape Draw
Rectangle Draw
Triangle Draw
Circle Draw
```

ShapeTest만 수정한 예

```
public class ShapeTest {
    private static Shape arrayOfShapes[];

    public static void main(String arg[]) {
            init();
            drawAll();
    }
    public static void init() {
            arrayOfShapes = new Shape[3];
            arrayOfShapes[0] = new Rectangle();
            arrayOfShapes[1] = new Triangle();
            arrayOfShapes[2] = new Circle();
    }
    public static void drawAll() {
            for (int i = 0; i < arrayOfShapes.length; i++) {
                    arrayOfShapes[i].draw();
            }
    }
}
```

Rectangle Draw
Triangle Draw
Circle Draw

동적 바인딩의 장점

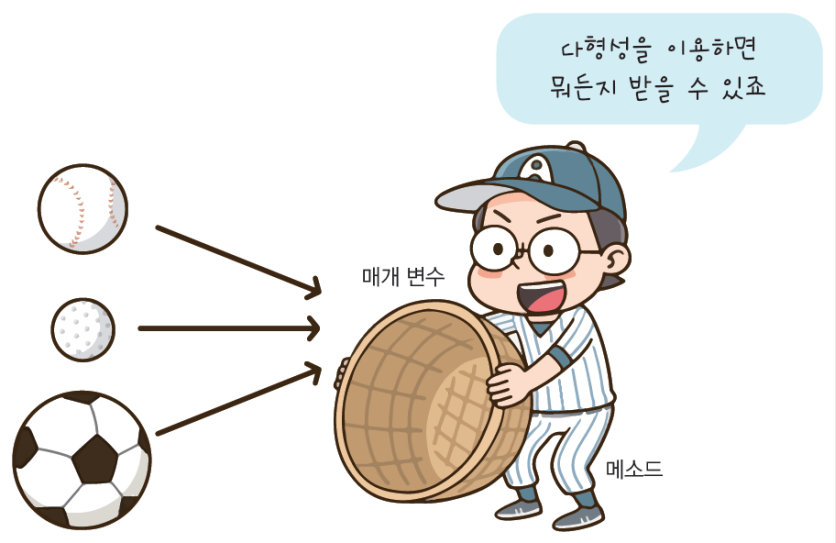
- 다음과 같은 새로운 클래스를 추가할 경우...

```
class Cylinder extends Shape {
  private int radius, height;

  public void draw(){
  System.out.println("Cylinder Draw");
  }
}
```

- 위와 같은 새로운 클래스가 추가되더라도 drawall() 코드는 최소한의 변경만 요한다. 즉, 배열의 추가

- 변수(혹은 메소드 매개변수)로 수퍼 클래스 참조 변수를 이용하면 동적 바인딩을 통해 수퍼 클래스 메소드를 사용하여
  - 모든 자식 클래스의 메소드를 수행시킬 수 있어, 다형성을 최대로 이용할 수 있다

  → 다형성을 이용하는 전형적인 방법



```
public class ShapeTest {
  public static void printLocation(Shape s) {
        System.out.println("x=" + s.x + " y=" + s.y);
  }
  public static void main(String arg[]) {
        Rectangle s1 = new Rectangle();
        Triangle s2 = new Triangle();
        Circle s3 = new Circle();

        printLocation(s1);
        printLocation(s2);
        printLocation(s3);
  }
}
```

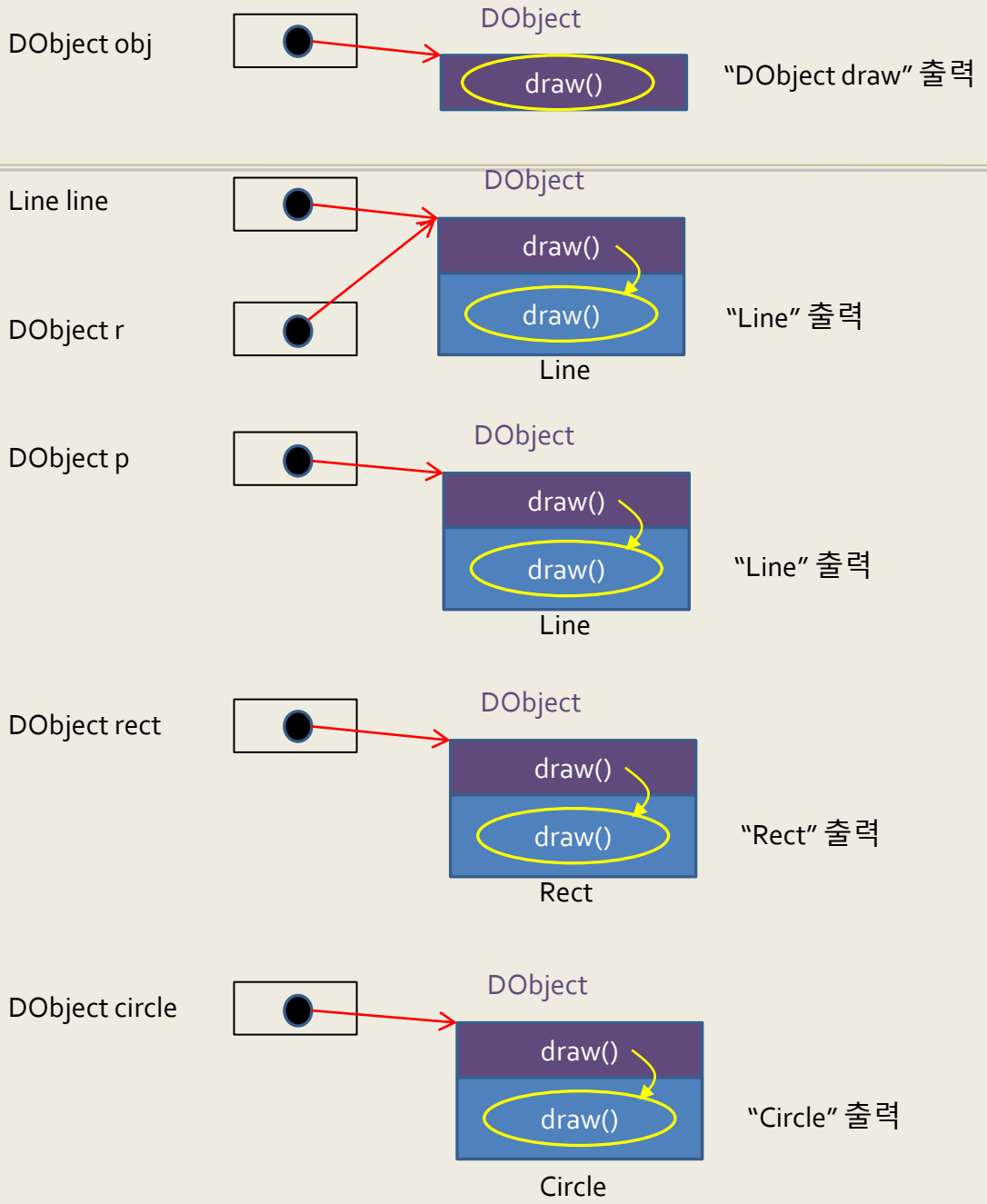
예제 : 메소드 오버라이딩 만들기

```
class DObject {
   public DObject next;

   public DObject() { next = null;}
    public void draw() {
      System.out.println("DObject
draw");
   }
}

class Line extends DObject {
   public void draw() { // 오버라이딩
       System.out.println("Line");
   }
}

class Rect extends DObject {
   public void draw() { //
오버라이딩
       System.out.println("Rect");
   }
}

class Circle extends DObject {
   public void draw() { // 오버라이딩
       System.out.println("Circle");
   }
}
```

```
public class MethodOverringEx {
   public static void main(String[] args) {
      DObject obj = new DObject();
      Line line = new Line();
      DObject p = new Line();
      DObject r = line;

      obj.draw(); // DObject.draw() 메소드 실행. "DObject draw" 출력
      line.draw(); // Line.draw() 메소드 실행. "Line" 출력
      p.draw(); // 오버라이딩된 메소드 Line.draw() 실행, "Line" 출력
      r.draw(); // 오버라이딩된 메소드 Line.draw() 실행, "Line" 출력

      DObject rect = new Rect();
      DObject circle = new Circle();
      rect.draw(); // 오버라이딩된 메소드 Rect.draw() 실행, "Rect" 출력

      circle.draw(); // 오버라이딩된 메소드 Circle.draw() 실행, "Circle"
출력
   }
}
```

```
DObject draw
Line
Line
Line
Rect
Circle
```

예제 실행 과정


DObject obj
DObject
draw()
"DObject draw" 출력
Line line
DObject r
DObject
draw()
draw()
Line
"Line" 출력
DObject p
DObject
draw()
draw()
Line
"Line" 출력
DObject rect
DObject
draw()
draw()
Rect
"Rect" 출력
DObject circle
DObject
draw()
draw()
Circle
"Circle" 출력

오버라이딩 활용

```
public static void main(String [] args) {
    DObject start, n, obj;

    // 링크드 리스트로 도형 생성하여 연결하기
    start = new Line(); //Line 객체 연결
    n = start;
    obj = new Rect();
    n.next = obj; //Rect객체 연결
    n = obj;
    obj = new Line(); // Line 객체 연결
    n.next = obj;
    n = obj;
    obj = new Circle(); // Circle 객체 연결
    n.next = obj;

    // 모든 도형 출력하기
    while(start != null) {
        start.draw();
        start = start.next;
    }
}
```

Line
Rect
Line
Circle