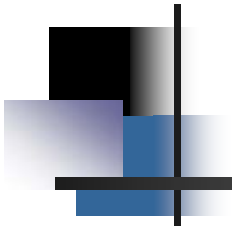


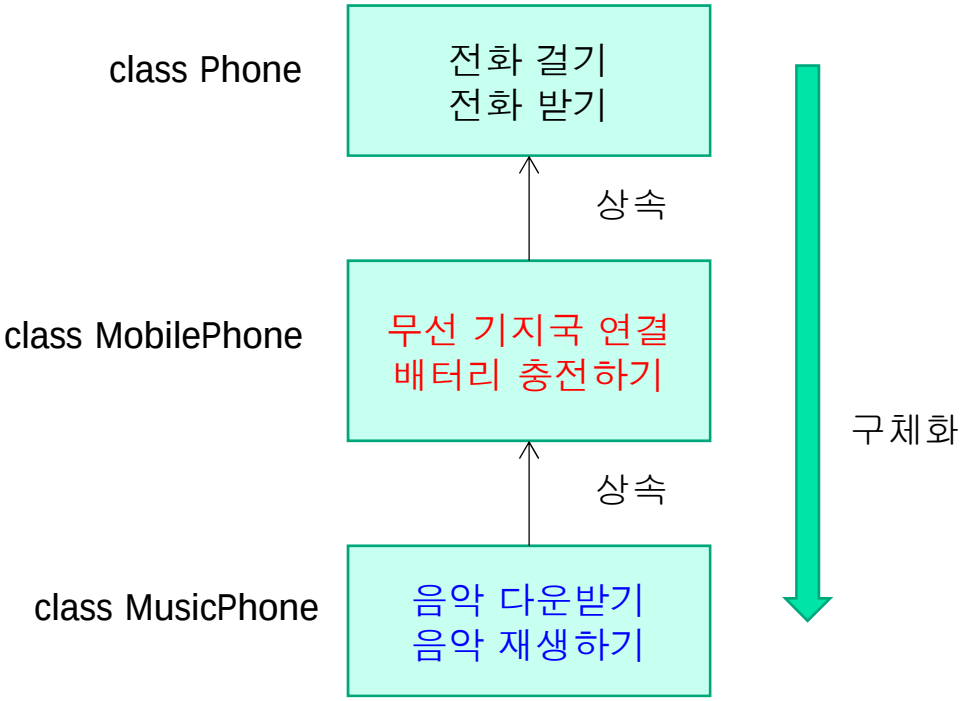
객체와 클래스 III
Advanced Classes

남 광 우

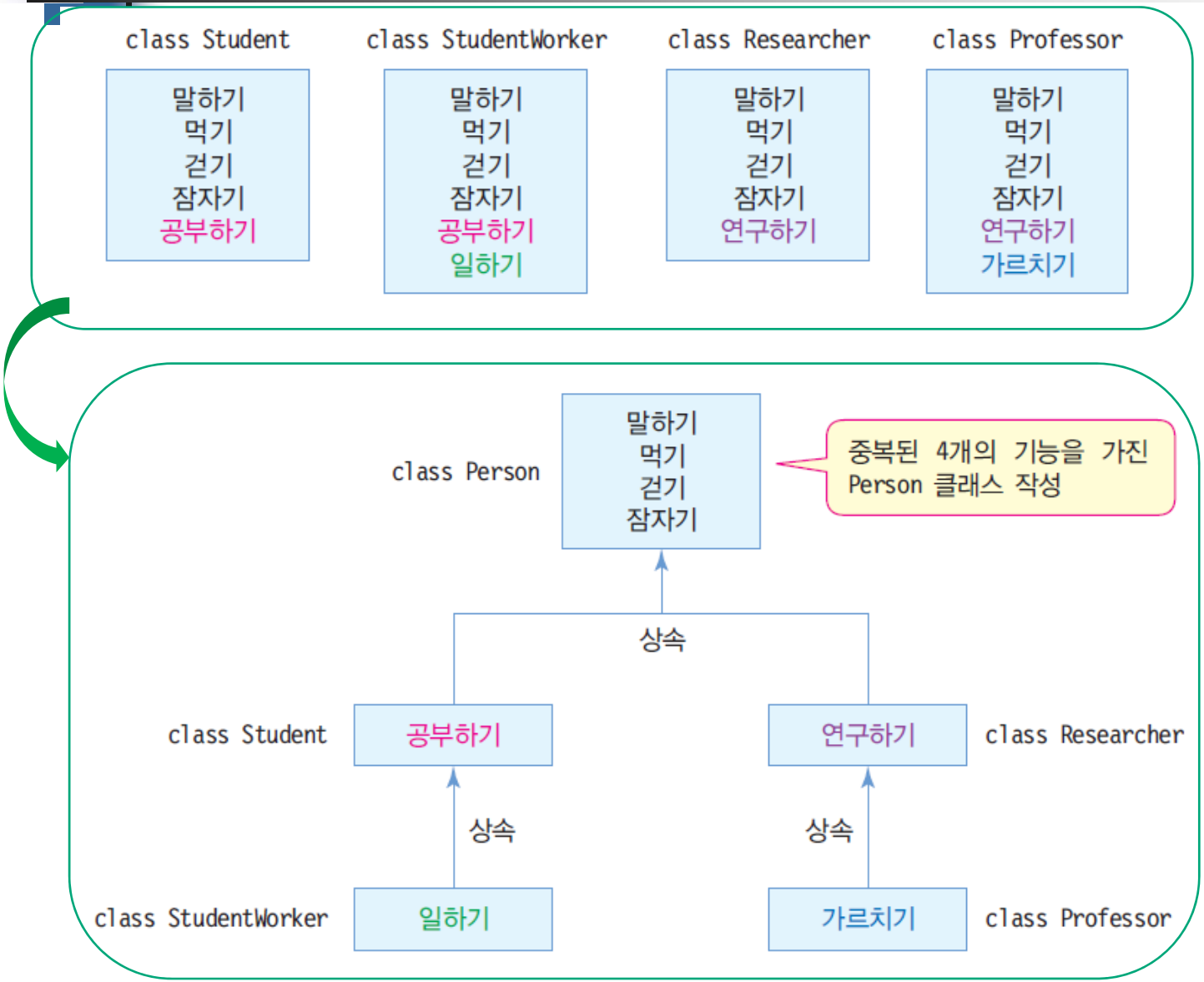
상속 (inheritance)

- 상속
  - 상위 클래스의 특성 (필드, 메소드)을 하위 클래스에 물려주는 것
  - 슈퍼 클래스 (superclass)
    - 특성을 물려주는 상위 클래스
  - 서브 클래스 (subclass)
    - 특성을 물려 받는 하위 클래스
    - 슈퍼 클래스에 자신만의 특성(필드, 메소드) 추가
    - 슈퍼 클래스의 특성(메소드)을 수정 : 구체적으로 오버라이딩이라고 부름
- 슈퍼 클래스에서 하위 클래스로 갈수록 구체적
  - 예) 폰 -> 모바일폰 -> 뮤직폰
- 상속을 통해 간결한 서브 클래스 작성
  - 동일한 특성을 재정의할 필요가 없어 서브 클래스가 간결해짐

상속 관계 예


class Phone
전화 걸기
전화 받기
상속
class MobilePhone
무선 기지국 연결
배터리 충전하기
상속
class MusicPhone
음악 다운받기
음악 재생하기
구체화

상속의 필요성



상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스

상속을 이용한
경우 중복이 제거되고
간결해진 클래스 구조

클래스 상속과 객체

- 상속 선언

```
public class Person {
        …
}
public class Student extends Person { // Person을 상속받는 클래스 Student 선언
        …
}
public class StudentWorker extends Student { // Student를 상속받는 StudentWorker 선언
        …
}
```

- 자바 상속의 특징
  - 다중 상속 지원하지 않음
    - 다수 개의 클래스를 상속받지 못함
  - 상속의 횟수는 무제한
  - 상속의 최상위 조상 클래스는 java.lang.Object 클래스
    - 모든 클래스는 자동으로 java.lang.Object를 상속받음

예제 5-1 : 클래스 상속 만들어 보기

(x,y)의 한 점을 표현하는 Point 클래스와 이를 상속받아 컬러 점을 표현하는 ColorPoint 클래스를 만들어보자.

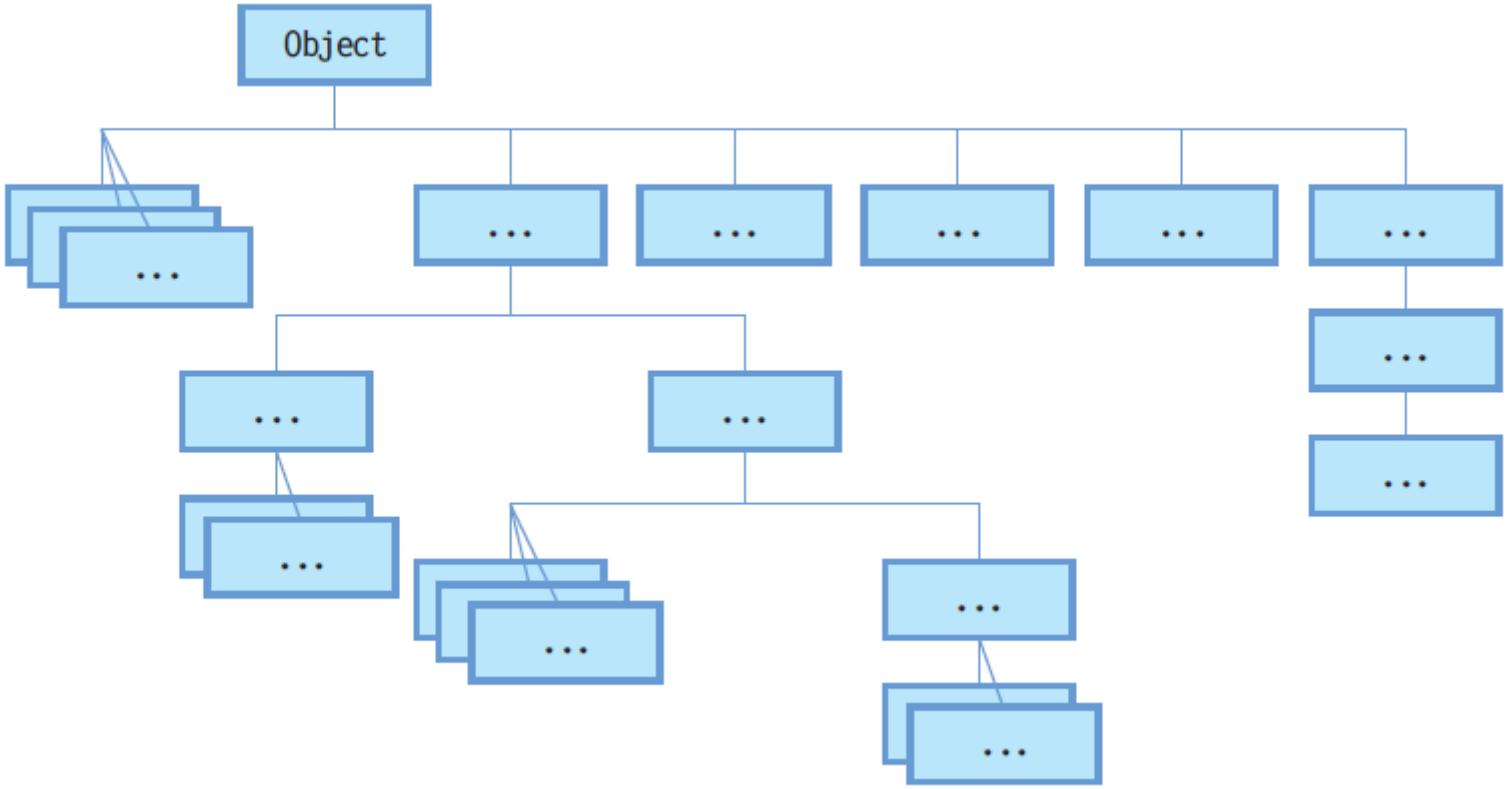
```
class Point {
   int x, y; // 한 점을 구성하는 x, y 좌표
   void set(int x, int y) {
      this.x = x; this.y = y;
   }
   void showPoint() { // 점의 좌표 출력
     System.out.println("(" + x + "," + y + ")");
   }
}
```

```
public class ColorPoint extends Point {
   // Point를 상속받은 ColorPoint 선언
   String color; // 점의 색
   void setColor(String color) {
      this.color = color;
   }
   void showColorPoint() { // 컬러 점의 좌표 출력
      System.out.print(color);
      showPoint(); // Point 클래스의 showPoint() 호출
   }
   public static void main(String [] args) {
      ColorPoint cp = new ColorPoint();
      cp.set(3,4); // Point 클래스의 set() 메소드 호출
      cp.setColor("red"); // 색 지정
      cp.showColorPoint(); // 컬러 점의 좌표 출력
   }
}
```

```
red(3,4)
```

자바의 클래스 계층 구조

자바에서는 모든 클래스는 반드시 java.lang.Object 클래스를 자동으로 상속받는다.



서브 클래스의 객체와 멤버 사용

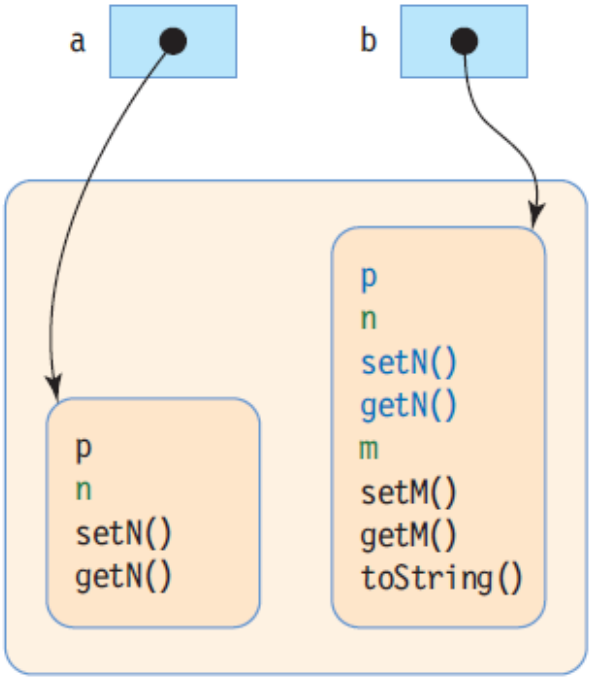
- 서브 클래스의 객체와 멤버 접근
  - 서브 클래스의 객체에는 슈퍼 클래스 멤버 포함
    - 슈퍼 클래스의 private 멤버는 상속되지 않음
      - 서브 클래스에서 직접 접근 불가
    - 슈퍼 클래스의 private 멤버는 슈퍼 클래스의 메소드를 통해 접근
  - 서브 클래스 객체에 슈퍼 클래스 멤버가 포함되므로 슈퍼 클래스 멤버의 접근은 서브 클래스 멤버 접근과 동일

슈퍼 클래스와 서브 클래스의 객체 관계

```
public class A {
    public int p;
    private int n;
    public void setN(int n) {
        this.n = n;
    }
    public int getN() {
        return n;
    }
}
```

```
public class B extends A {
    private int m;
    public void setM(int m) {
        this.m = m;
    }
    public int getM() {
        return m;
    }
    public String toString() {
        String s = getN() + " " + getM();
        return s;
    }
}
```

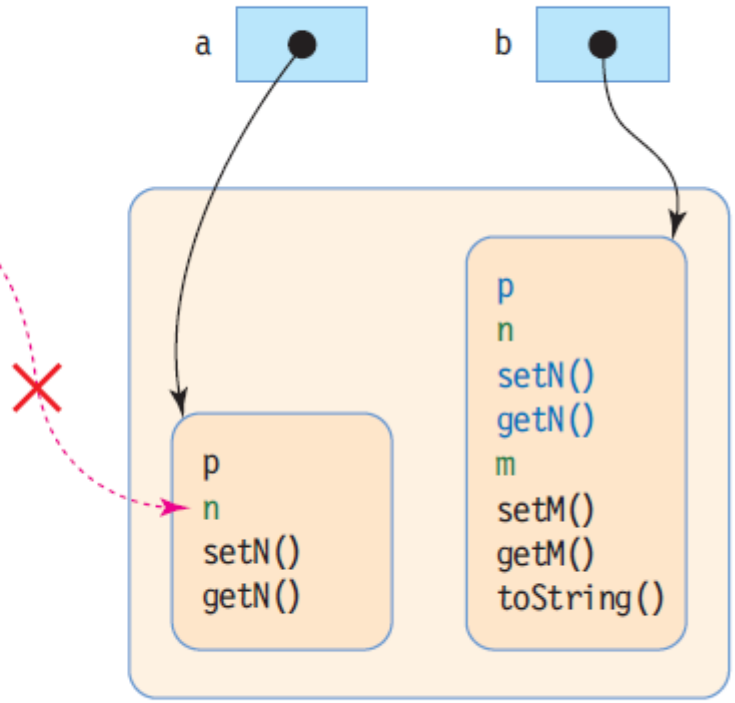
```
public static void main(String [] args) {
    A a = new A();
    B b = new B();
}
```



main() 실행 중 생성된 인스턴스

서브 클래스의 객체 멤버 접근

```
public class MemberAccessExample {
    public static void main(String [] args) {
        A a = new A();
        B b = new B();

        a.p = 5;
        a.n = 5; // n은 private 멤버, 컴파일 오류 발생

        b.p = 5;
        b.n = 5; // n은 private 멤버, 컴파일 오류 발생
        b.setN(10);
        int i = b.getN(); // i는 10

        b.m = 20; // m은 private 멤버, 컴파일 오류 발생
        b.setM(20);
        System.out.println(b.toString());
        // 화면에 10 20이 출력됨
    }
}
```



10 20

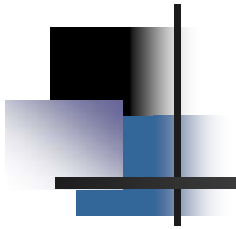
상속 관계에 있는 클래스 간 멤버 접근

클래스 Person을 아래와 같은 멤버 필드를 갖도록 선언하고 클래스 Student는 클래스 Person을 상속받아 각 멤버 필드에 값을 저장하시오. 이 예제에서 Person 클래스의 private 필드인 weight는 Student 클래스에서는 접근이 불가능하여 슈퍼 클래스인 Person의 getter와 setter를 통해서만 조작이 가능하다.

- int age;
- public String name;
- protected int height;
- private int weight;

```
class Person {
   int age;
   public String name;
   protected int height;
   private int weight;
   public void setWeight(int weight) {
      this.weight = weight;
   }
   public int getWeight() {
      return weight;
   }
}
```

```
public class Student extends Person {
   void set() {
      age = 30;
      name = "홍길동";
      height = 175;
      setWeight(99);
   }
   public static void main(String[] args) {
      Student s = new Student();
      s.set();
   }
}
```

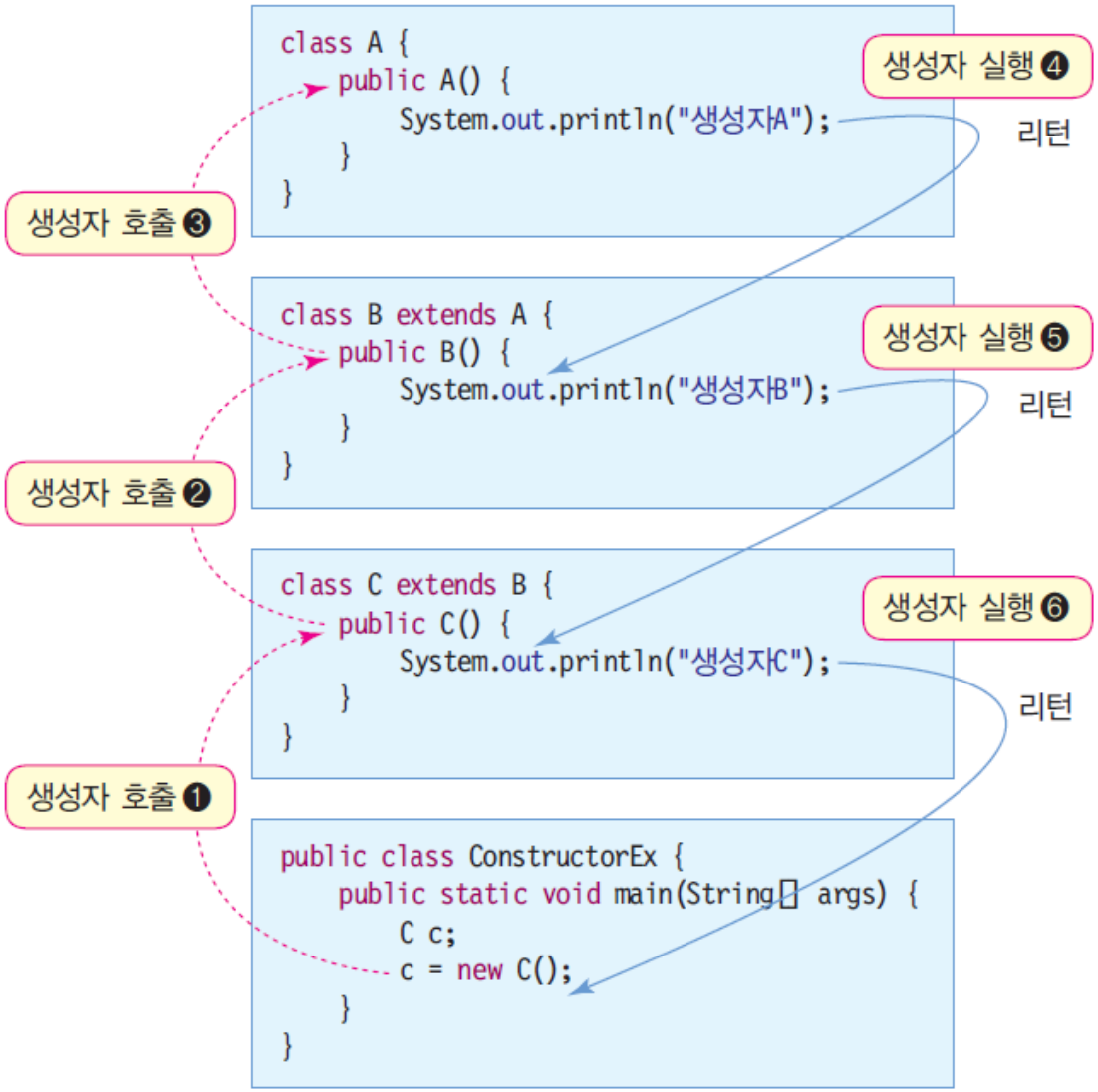


상속 관계에서의 생성자 호출

- 서브 클래스의 인스턴스가 생성될 때 서브 클래스의 생성자와 슈퍼 클래스의 생성자가 모두 실행됨

- 실행 순서 : superclass => subclass

상속 관계에서의 생성자 호출



예상 실행 결과는 ?

```
생성자A
생성자B
생성자C
```

위 코드는 모두 ConstructorEx.java 파일에 저장된다.

서브 클래스와 슈퍼 클래스의 생성자 짝 맞추기

- 슈퍼 클래스와 서브 클래스
  - 각각 여러 개의 생성자 가능
- 슈퍼 클래스와 서브 클래스의 생성자 사이의 짝 맞추기
  - 서브클래스의 객체 생성 시, 실행 가능한 슈퍼 클래스와 서브 클래스의 생성자 조합
    - 컴파일러는 서브 클래스의 생성자를 기준으로 아래 표와 같은 슈퍼 클래스의 생성자를 찾음
      - 경우 1, 3
    - 개발자가 서브 클래스의 생성자에 슈퍼 클래스의 짝을 지정하는 방법
      - 경우 2, 4
      - super() 키워드 이용

경우	1	2	3	4
서브 클래스	기본 생성자	기본 생성자	매개 변수를 가진 생성자	매개 변수를 가진 생성자
슈퍼 클래스	기본 생성자	매개 변수를 가진 생성자	기본 생성자	매개 변수를 가진 생성자

1: 슈퍼클래스(기본생성자),서브클래스(기본생성자)

아래 코드는 모두 ConstructorEx2.java 파일에 저장된다.

```
class A {
    public A() {
        System.out.println("생성자A");
    }
    public A(int x) {
    .....
    }
}
```

서브클래스의 생성자가 기본 생성자인 경우, 컴파일러는 자동으로 슈퍼클래스의 기본 생성자와 짝을 맺음

```
class B extends A {
    public B() {
        System.out.println("생성자B");
    }
}
```

```
public class ConstructorEx2 {
    public static void main(String[] args) {
        B b;
        b = new B(); // 생성자 호출
    }
}
```

생성자A
생성자B

```
class A {
    public A(int x) {
        System.out.println("생성자A");
    }
}
```

컴파일러가 public B()에 대한 짝을 찾을 수 없음

```
class B extends A {
    public B() {
        System.out.println("생성자B");
    }
}
```

```
public class ConstructorEx2 {
    public static void main(String[] args) {
        B b;
        b = new B();
    }
}
```

컴파일러에 의해 "Implicit super constructor A() is undefined. Must explicitly invoke another constructor" 오류 발생

3:서브 클래스에 매개변수 있는 생성자는 슈퍼 클래스의기본생성자와 짝을 이룸

옆의 코드는 모두 ConstructorEx3.java 파일에 저장된다.

```
class A {
    public A() {
        System.out.println("생성자A");
    }
    public A(int x) {
        System.out.println("매개변수생성자A");
    }
}
```

```
class B extends A {
    public B() {
        System.out.println("생성자B");
    }
    public B(int x) {
        System.out.println("매개변수생성자B");
    }
}
```

```
public class ConstructorEx3 {
    public static void main(String[] args) {
        B b;
        b = new B(5);
    }
}
```

생성자A
매개변수생성자B

super()

- super()
  - 서브 클래스에서 명시적으로 슈퍼 클래스의 생성자를 선택 호출할 때 사용
  - 사용 방식
    - super(parameter);
    - 인자를 이용하여 슈퍼 클래스의 적당한 생성자 호출
    - 반드시 서브 클래스 생성자 코드의 제일 첫 라인에 와야 함

super()를 이용한 사례

옆의 코드는 모두 ConstructorEx4.java 파일에 저장된다.

```
class A {
    public A() {
        System.out.println("생성자A");
    }
    public A(int x) {
        System.out.println("매개변수생성자A" + x);
    }
}
```

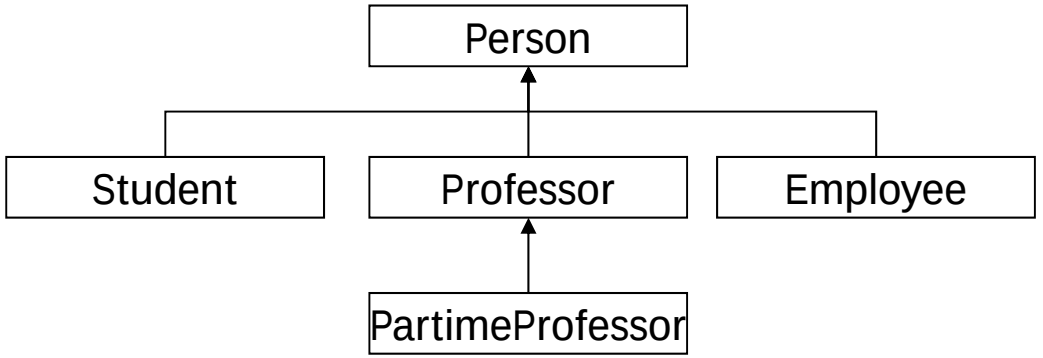
```
class B extends A {
    public B() {
        System.out.println("생성자B");
    }
    public B(int x) {
        super(x);
        System.out.println("매개변수생성자B" + x);
    }
}
```

```
public class ConstructorEx4 {
    public static void main(String[] args) {
        B b;
        b = new B(5);
    }
}
```

매개변수생성자A5
매개변수생성자B5

상속에서의 다형성

- 상속 클래스 변수의 할당



- 상위 클래스 변수에 하위 클래스 객체 할당 가능
  - Student/Professor는 Person중 하나이므로 Person 변수에 할당 가능

```
// 예제 1
Person p;

p = new Person();                  // O
P = new Professor();               // O
p = new PartimeProfessor();        // O
```

```
// 예제 2
Person[]   per = new Person[3];
Professor  prof = new Professor();
Student    stud = new Student();

per[0] = prof;                          // O
per[1] = stud ;                         // O
per[2] = new PartimeProfessor();        // O
```

상속에서의 다형성

- 상위 클래스 변수에 할당된 객체는 상위 클래스의 객체로 간주

```
per[0] = prof;

// O : per[0]는 Person 객체로 간주되므로 아래 호출은 성립
per[0].getName();

// X : per[0]는 Professor 객체로 간주되므로 아래 호출은 에러
per[0].getEmployeeNumber();
```

- 하위 클래스 변수에 상위 클래스 객체를 지정 할 수 없음
  - 모든 파트교수는 교수이지만, 교수가 파트교수는 아님

```
Professor prof;

prof = new Person();              // X
prof = per[0];                    // X
prof = new PartimeProfessor();    // O
```

상속 클래스간의 캐스팅

- 캐스팅
  - 하나의 타입을 다른 타입으로 강제로 변환하는 과정

```
double   x = 3.405;
int      nx = (int)x;
```

- 클래스간 캐스팅
  - 하나의 클래스 객체를 다른 클래스로 강제로 변환
    - 하위 클래스 객체를 상위 클래스로 캐스팅은 허용되나,
    - 상위 클래스 객체를 하위 클래스로 캐스팅은 허용되지 않음

```
Person[]   per = new Person[3];

per[0] = new Professor();
per[1] = new Student();
per[2] = new PartimeProfessor();

Professor prof1 = (Professor)per[0];              // O
Professor prof2 = (Professor)per[1];              // X
Professor prof3 = (Professor)per[2];              // O
Professor prof4 = (Professor)(new Person());      // X
```

instanceof

- instanceof 연산자
  - 어떤 클래스의 인스턴스인지 판단하기 위한 연산자
  - true 또는 false 값을 반환

- instanceof의 사용 예

```
Person per = new Professor();

if  ( per instanceof Person )
     System.out.println( per.getName() ) ;                    // true

if ( per instanceof Student )
     System.out.println( ((Student)per).getStudentNumber()); // false

if ( per instanceof Professor )
     System.out.println( ((Professor)per).getEmployeeNumber()); // true
```

instanceof

- instanceof의 사용 예

코드	결 과
`per instanceof Person`	true
`prof instanceof Person`	true
`per instanceof Professor`	false
`! (prof instanceof Person)`	false
`! prof instanceof Person`	syntax error
`dalmatian instanceof "Dog"`	syntax error
`null instanceof String`	false

instanceof

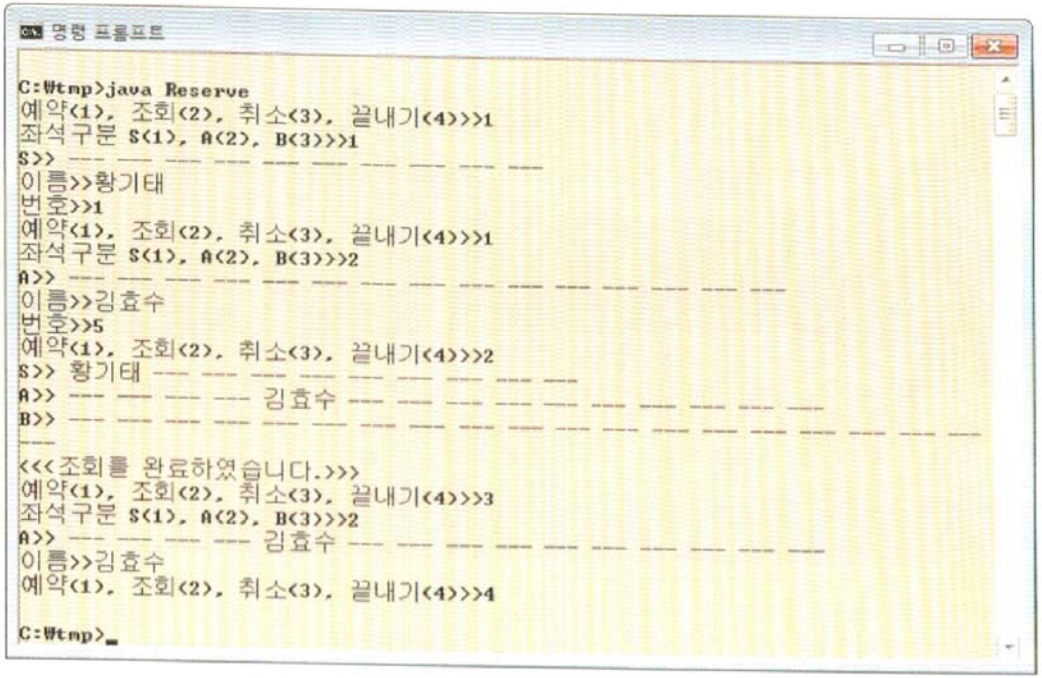
- instanceof의 사용 예

```
Person[]   per = new Person[3];

per[0] = new Professor();
per[1] = new Student();
per[2] = new PartimeProfessor();


for ( int i=0; i < per.length; i++ )
{
    if ( per[i] instanceof Person )
        System.out.println( "per["+i + "]는 Person 클래스의 객체입니다");
    if ( per[i] instanceof Student )
        System.out.println( "per["+i + "]는 Student 클래스의 객체입니다");
    if ( per[i] instanceof Professor )
        System.out.println( "per["+i + "]는 Professor 클래스의 객체입니다");
    if ( per[i] instanceof PartimeProfessor )
        System.out.println( "per["+i + "]는 PartimeProfessor 클래스의 객체입니다");
}
```

실행결과는?

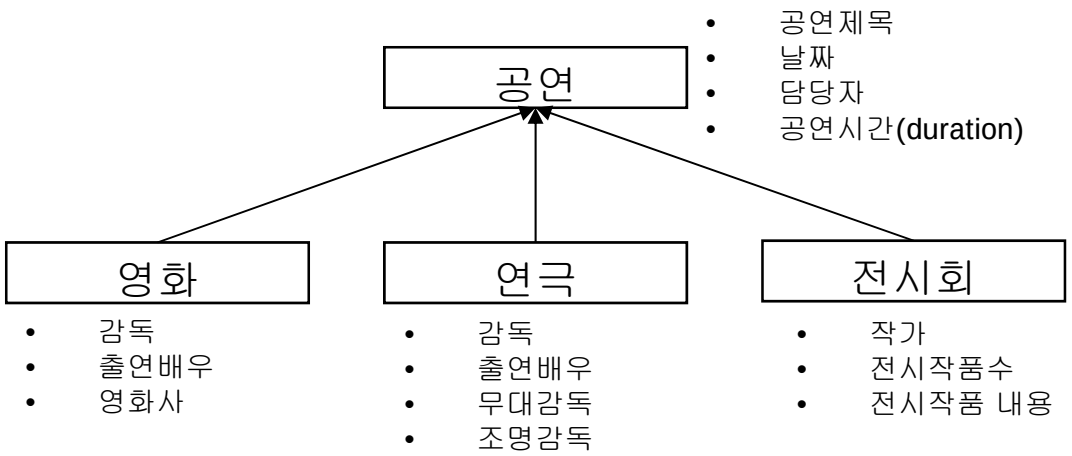
리포트 : 공연 예약 시스템 상속 확장

- 공연은 하루에 한 번 있다.
- 좌석은 S석, A석, B석 타입이 있으며 모두 10석의 좌석이 있다.
- 공연 예약 시스템의 메뉴는 "예약", "조회", "취소", "끝내기"가 있다.
- 예약은 한 자리만 예약할 수 있고 좌석 타입, 예약자 이름, 좌석 번호를 순서대로 입력받아 예약한다.
- 조회는 모든 종류의 좌석을 표시한다.
- 취소는 예약자의 이름을 입력하여 취소한다.
- 없는 이름, 없는 번호, 없는 메뉴, 잘못된 취소 등에 대해서 오류 메시지를 출력하고 사용자가 다시 시도하도록 한다.



리포트 : 공연 예약 시스템 상속 확장

- 공연의 종류 클래스 계층

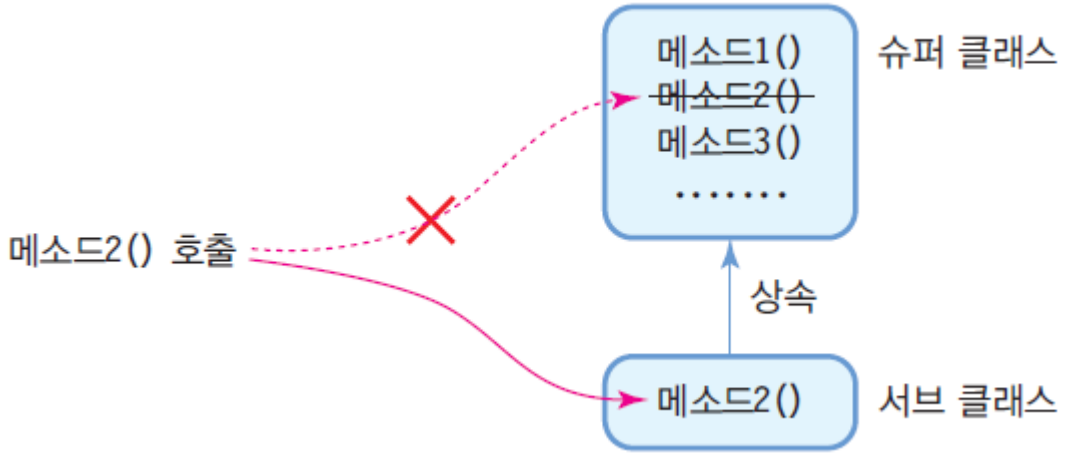


메뉴의 추가
(1) 공연리스트 보기
(2) 공연 추가
(3) 공연 삭제
(4) 공연별 예약 : 공연 리스트 보여주고, 예약하기
(5) 공연별 좌석 조회 : 공연 리스트 보여주고, 선택해서 좌석조회
(6) 공연별 예약 취소

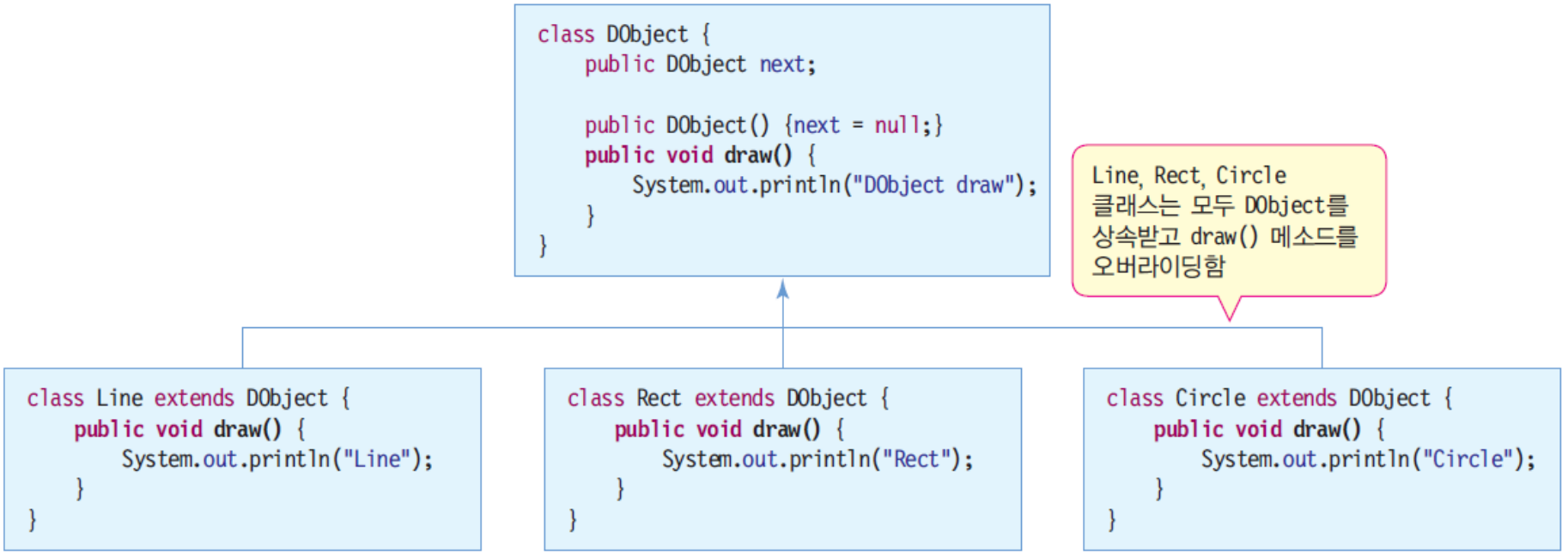
공연은 하나의 배열로 관리되어함

메소드 오버라이딩

- 메소드 오버라이딩(Method Overriding)
  - 슈퍼 클래스의 메소드를 서브 클래스에서 재정의하는 것
    - 슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수, 리턴 타입 등 모든 것 동일하게 정의
      - 이 중 하나라도 다르면 메소드 오버라이딩 실패
  - "메소드 무시하기"로 번역되기도 함
  - 동적 바인딩 발생
    - 오버라이딩된 메소드가 무조건 실행되도록 동적 바인딩 됨

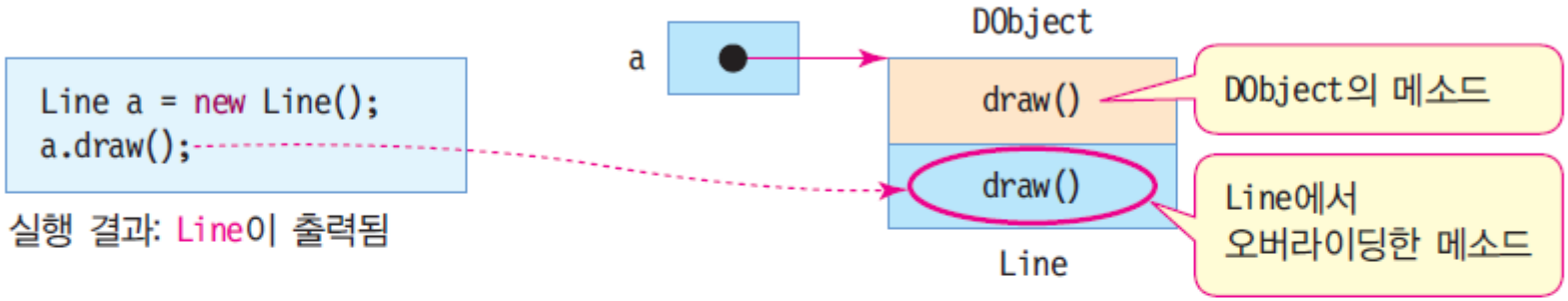


메소드 오버라이딩 사례

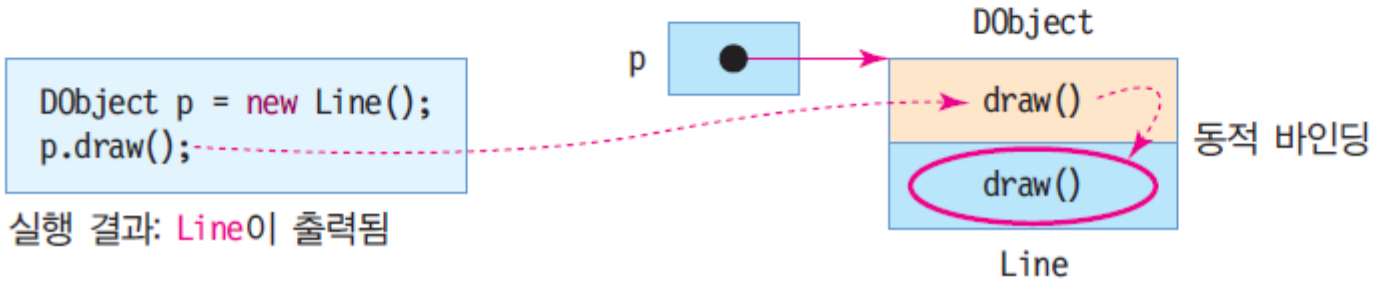

class DObject {
public DObject next;
public DObject() {next = null;}
public void draw() {
System.out.println("DObject draw");
}
}
Line, Rect, Circle 클래스는 모두 DObject를 상속받고 draw() 메소드를 오버라이딩함
class Line extends DObject {
public void draw() {
System.out.println("Line");
}
}
class Rect extends DObject {
public void draw() {
System.out.println("Rect");
}
}
class Circle extends DObject {
public void draw() {
System.out.println("Circle");
}
}

서브 클래스 객체와 오버라이딩된 메소드 호출

(1) 서브 클래스 레퍼런스로 오버라이딩된 메소드 호출



(2) 업캐스팅에 의해 슈퍼 클래스 레퍼런스로 오버라이딩된 메소드 호출(동적 바인딩)



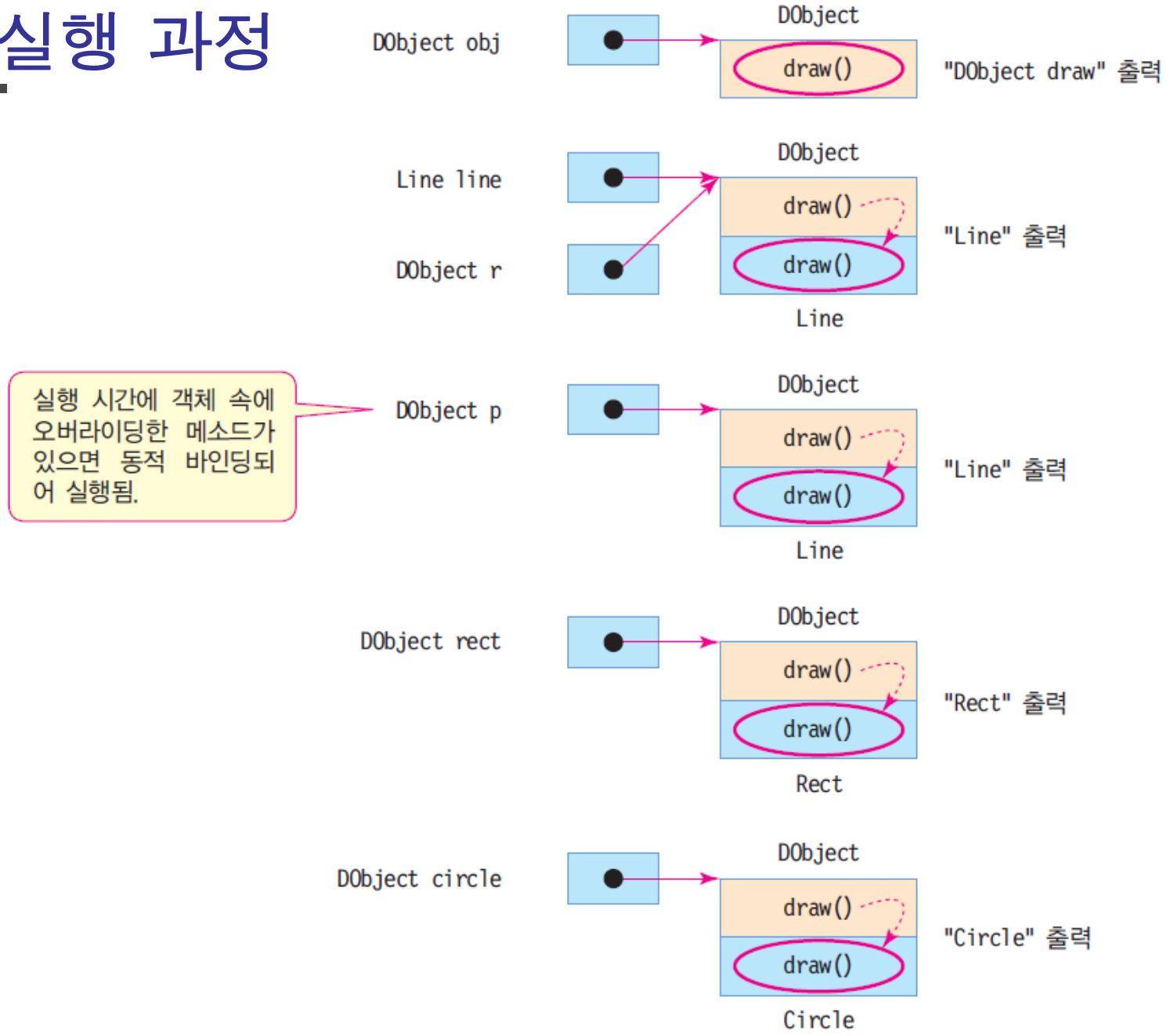
예제 5-4 : 메소드 오버라이딩 만들기

```
class DObject {
   public DObject next;

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

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

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

class Circle extends DObject {
   public void draw() { // 메소드 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" 출
력
   }
}
```

```
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" 출력
Line
실행 시간에 객체 속에 오버라이딩한 메소드가 있으면 동적 바인딩되어 실행됨.
DObject p
DObject
draw()
draw()
"Line" 출력
Line
DObject rect
DObject
draw()
draw()
"Rect" 출력
Rect
DObject circle
DObject
draw()
draw()
"Circle" 출력
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.

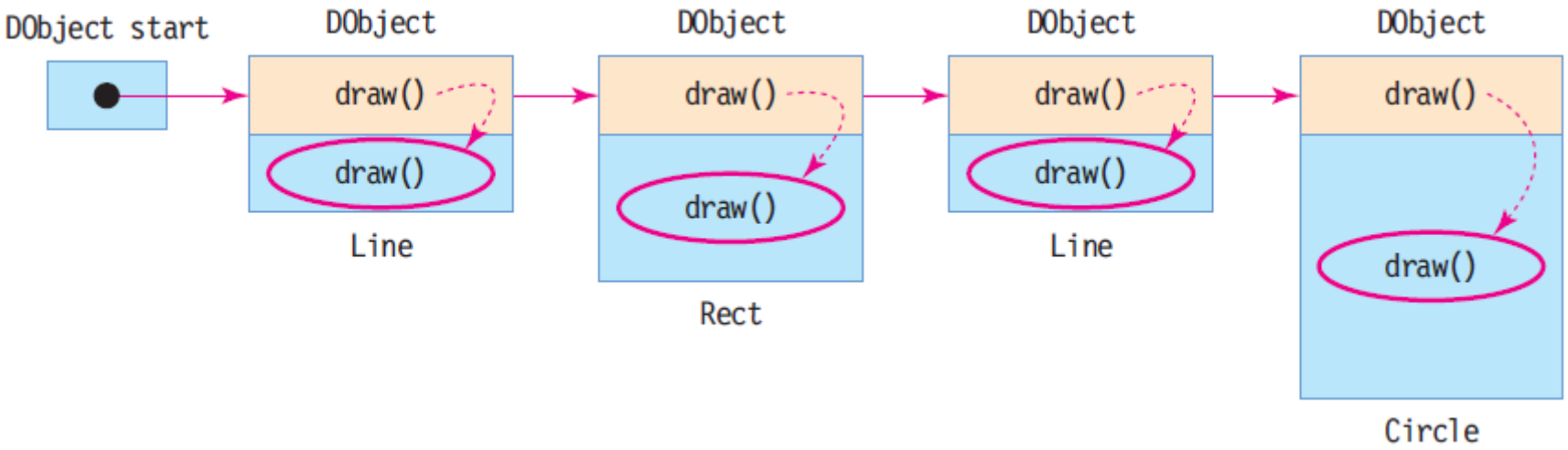
   public > protected > private 순으로 지정 범위가 좁아진다.
3. 반환 타입만 다르면 오류
4. static, private, 또는 final 메소드는 오버라이딩 될 수 없다.

```
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
        }
}
```

오버

```
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
```



동적 바인딩

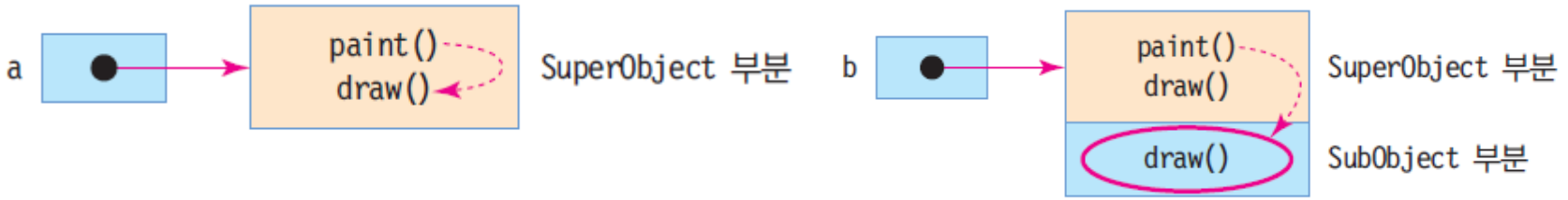
```
public class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
    public static void main(String [] args) {
        SuperObject a = new SuperObject();
        a.paint();
    }
}
```

Super Object

```
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();
    }
}
```

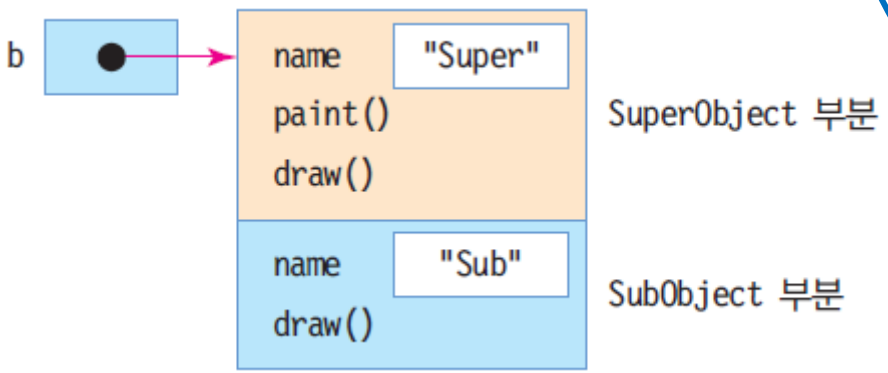
동적바인딩

Sub Object



super 키워드

• super는 서브클래스에서 슈퍼 클래스의 멤버를 접근할 때 사용되는 슈퍼클래스 타입의 레퍼런스.
• 상속관계에 있는 서브 클래스에서만 사용됨
•오버라이딩된 슈퍼 클래스의 메소드 호출 시 사용




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

Super
Sub

예제 5-5 : 메소드 오버라이딩

Person을 상속받는 Professor라는 새로운 클래스를 만들고 Professor 클래스에서 getPhone() 메소드를 재정의하라. 그리고 이 메소드에서 슈퍼 클래스의 메소드를 호출하도록 작성하라.

```
class Person {
   String phone;
   public void setPhone(String phone) {
      this.phone = phone;
   }
   public String getPhone() {
      return phone;
   }
}

class Professor extends Person {
   public String getPhone() {
      return "Professor : " + super.getPhone();
   }
}
```

super.getPhone()은 아래 p.getPhone()과 달리 동적 바인딩이 일어나지 않는다.

```
public class Overriding {
   public static void main(String[] args) {
      Professor a = new Professor();
      a.setPhone("011-123-1234");
      System.out.println(a.getPhone());
      Person p = a;
      System.out.println(p.getPhone());
   }
}
```

동적 바인딩에 의해 Professor의 getPhone() 호출.

```
Professor : 011-123-1234
Professor : 011-123-1234
```

오버라이딩 vs. 오버로딩

비교요소	메소드 오버로딩	메소드 오버라이딩
정의	같은 클래스나 상속 관계에서 동일한 이름의 메소드 중복 작성	서브 클래스에서 슈퍼 클래스에 있는 메소드와 동일한 이름의 메소드 재작성
관계	동일한 클래스 내 혹은 상속 관계	상속 관계
목적	이름이 같은 여러 개의 메소드를 중복 정의하여 사용의 편리성을 향상	슈퍼 클래스에 구현된 메소드를 무시하고 서브 클래스에서 새로운 기능의 메소드를 재정의 하고자 함
조건	메소드 이름은 반드시 동일함. 메소드의 인자의 개수나 인자의 타입이 달라야 성립	메소드의 이름, 인자의 타입, 인자의 개수, 인자의 리턴 타입 등이 모두 동일하여야 성립
바인딩	정적 바인딩. 컴파일 시에 중복된 메소드 중 호출되는 메소드 결정	동적 바인딩. 실행 시간에 오버라이딩된 메소드 찾아 호출

Object 최상위 클래스의 사용

- Object 클래스
  - 모든 Java 클래스의 근원적인 조상 클래스로서, 자바의 모든 클래스는 Object 클래스를 확장한 것임
  - Object는 상속을 명확하게 표현하지 않아도 자동 상속됨

class Person extends Object ⟶ 생략

  - 모든 클래스는 Object클래스의 메소드 사용 가능
    - **int hashCode()**
    - **Class getClass()**
    - **boolean equals(Object otherObject)**
    - **String toString()**
    - **Object clone()**

  - 모든 객체는 Object 타입의 변수를 사용할 수 있음

```
Object  obj = new Professor( 101011,  “Kwang Woo”, 3500)  // OK
Professor  prof = (Professor)obj;                          // OK
Object  obj2 = new Professor[10];                          // OK
Object  obj3 = 300.125;                                    // Error!
```

Object : hashCode() 메소드

- 해쉬코드
  - 객체의 값을 대표하기 위해 해쉬함수에 의해 생성된 값
  - 서로 다른 값을 가진 객체는 hashCode의 값이 다르다는 높은 가능성이 있어야 함(동일 값은 동일 hashCode)

- int hashCode()
  - Object의 값을 기반으로 해쉬 코드 값을 생성하는 메소드

- 문자열의 hashCode 함수로부터 나온 해쉬코드의 예

문자열	해쉬코드
“Hello”.hashCode()	140207504
“Harry”.hashCode()	140013338
“Hacker”.hashCode()	884756206

Object : hashCode() 메소드

- 해쉬코드의 사용
  - 모든 객체가 갖는 기본 해쉬코드는 객체의 메모리 주소를 사용
  - 클래스 구현시 hashCode()를 재구현 할 수 있음
    - String 클래스는 문자열 값을 해쉬 코드에 이용
    - 동일 객체는 동일 hashCode 값을 가져야 함

- 해쉬 코드의 예

```
String  s = "OK";
StringBuffer sb = new StringBuffer(s);
System.out.println( s.hashCode() + "  "+sb.hashCode() );

String t = new String("OK");
StringBuffer tb = new StringBuffer(t);
System.out.println( t.hashCode() + "  "+tb.hashCode() );
```

```
// 출력결과
3030   20526976
3030   20527144
```

```
// String 클래스의 해쉬코드
int hash = 0;
for (int i=0; i<length(); i++)
    hash = 31*hash + charAt(i);
```

Object : hashCode() 메소드

- 해쉬코드의 재구현
  - 동일 객체에 대하여 동일 해쉬코드를 반환토록 구현
    - 기본 hashCode 구현은 객체의 메모리 주소를 사용하여 hashCode를 생성
    - 동일 메모리 주소가 아닌 동일 값을 갖는 객체에 대하여 동일 hashCode를 생성토록 재구현
      - 이름, 사번, 월급이 동일한 Professor 객체에 대하여 동일 hashCode 생성

- 해쉬코드의 재구현 예

```
public Class Professor
{
    public int hashCode()
    {
        return 7*name.hashCode() + 11*new Double(salary).hashCode()
                + 13*hireDay.hashCode();
     }
}
```

Object : equals() 메소드

- equals() : 동등검사
  - 동일 객체 인지를 검사 => Object는 hashCode() 를 사용
  - 동일 객체의 의미는 각 클래스별로 상이할 수 있으므로 상황에 따라 재구현 필요

- equals()의 사용예

```
public class Professor
{
    public boolean equals(Object  otherObject)
    {
        if ( this == otherObject)   return true;
        if (otherObject == null )   return false;
        if (getClass() != otherObject.getClass() )    return false;

        Professor  other = (Professor)otherObject;

        return  name.equals(other.name) && salary == other.salary
                && hireDay.equals( other.hireDay );
     }
}
```

Object : equals() 메소드

- equals()의 사용 예 : 상속 클래스

```
public class PartimeProfessor extends Professor
{
    public boolean equals(Object  otherObject)
    {
        if ( !super.equals( otherObject ) ) return false;

        PartimeProfessor part = (PartimeProfessor) otherObejct;
        return  lectureTime == other.lectureTime;
     }
}
```

Object : toString() 메소드

- toString()
  - 객체의 값을 표현하는 문자열을 반환
  - Object 클래스의 기본 구현은 클래스의 이름과 객체 참조값을 반환하는 것임

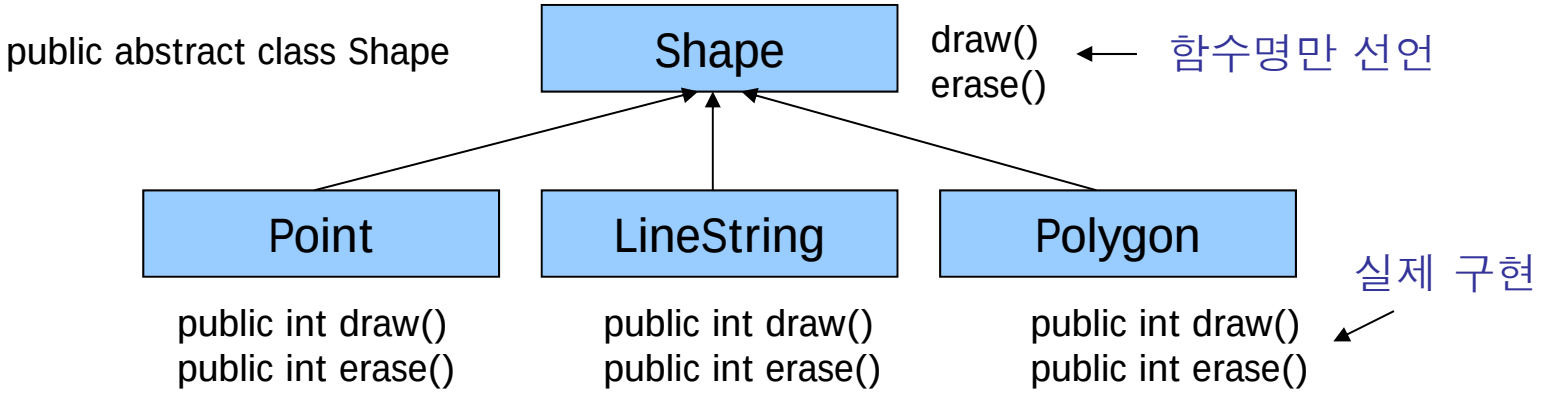
```
System.out.println( System.out );

// 출력결과
java.io.PrintStream@2f684
```

- toString()의 재구현 예

```
public class Professor
{
    public String toString()
    {
        // getClass() 메소드는 객체의 클래스를 반환
        return   getClass().getName()
                    + "[name=" + name + ", salary =" + salary;
    }
}
```

추상 클래스

- abstract 클래스
  - 상위 클래스가 너무 일반적이어서 객체로 만들어 사용하기 보다는 다른 클래스를 위한 기본 클래스로 사용될 경우 사용



  - abstract 메소드가 하나라도 있는 경우 abstract 클래스
  - abstract 클래스는 객체를 생성할 수 없음

```
public abstract class Shape
{
    int shapeNumber = 0;

    public abstract draw();
    public abstract erase();
    public getShapeNumber() { return shapeNumber; }
}
```

abstract 클래스도 변수와 구현을 포함할 수 있음

추상 클래스

- abstract 클래스의 사용 예

```
public abstract class Person
{
    private String name;

    public Person( String n )
    {
        name = n;
    }

    public abstract String getDescription();

    public String getName()
    {
        return name;
    }
}
```

```
public class Student extends Person
{
    private String   department;
    private String   studentNumber;

    public Student( String n, String d, String num)
    {
        super( n );
        department = d;
        studentNumber = num;
    }

    public String getDescription()
    {
        return "이 학생의 전공 : " + major;
    }
}
```

```
Person  p = new Person("Kwang Woo");                          // Error
Person  p = new Student("Kwang Woo", "컴퓨터", "89022011");  // OK
```

참고 : 설계상 필요에 따라 추상 메소드가 없는 추상 클래스도 생성할 수 있음

추상 클래스

- University DB

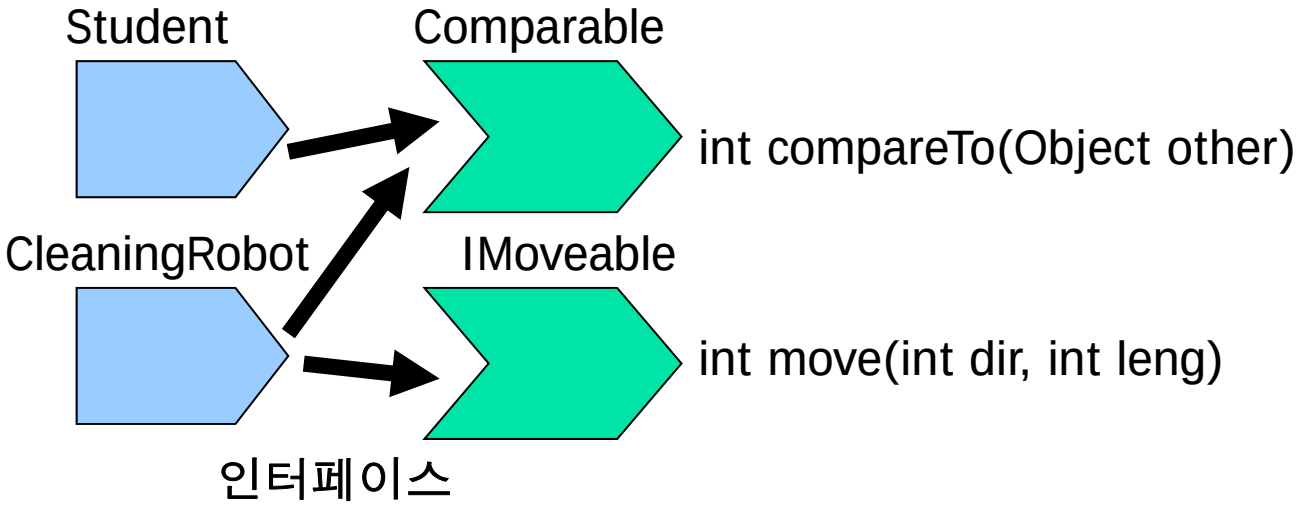
```
public class University
{
    public static void main( String[] args )
    {
        Person[] people = new Person[2];

        people[0] = new Professor(“Kwang”, 50000, 2005, 4, 15);
        people[1] = new Stduent(“Kim”, “전산”, “2003055011”);

        for( Person p : people )
            System.out.println( p.getName() + “,” + p.getDescription();
    }
}
```

상속 받은 메소드

각 클래스의
abstract 메소드 구현을 실행

interface 클래스

- interface
  - 골격만 있고 몸체가 없는 메소드만으로 이루어진 클래스
  - 클래스가 어떻게(how) 수행하기보다는 무엇(what)을 수행해야 할지 설명
  - 한 클래스는 하나 또는 그 이상의 인터페이스를 구현(implements) 할 수 있음

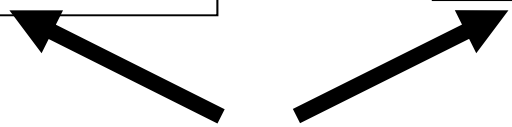


interface 클래스

- interface의 사용 예

```
public interface Comparable<T>
{
    int compareTo(Object other);
}
```

```
public interface IMoveable
{
    int move(int dir, int leng);
}
```



```
public class CleaningRobot extends Robot
                implements IComparable<CleaningRobot>, IMoveable
{
    ....
    public int compareTo(CleaningRobot robot)
    {
          return  this.getSpeed() - robot.getSpeed();
    }

    public int move(int dir, int leng)
    {
         if ( dir >=1 && dir <=4 )
          ...
    }
}
```

interface 클래스

- interface 클래스 응용
  - Arrays.sort()로 정렬이 가능한 클래스 만들기
    - Arrays.sort()는 Comparable 인터페이스를 구현해야 함

```
public class Student implements Comparable<Student>
{
    private String number;

    .....
    public String getNumber()
    {
       return number;
     }

    public int compareTo(Student other)
    {
       return getNumber().compareTo( other.getNumber() );
     }
     ....
}
```

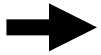
```
public class StudentSort
{
    public static void main(String[] args)
    {
        Student[] stds = new Student[3];

        stds[0] = new Student("Kwang", "8911");
        stds[1] = new Student("Park", "0322");
        stds[2] = new Student("Kim", "0422");

        Arrays.sort( stds );

        for( Student s : stds )
            System.out.println(e.getNumber());
     }
}
```

interface 클래스

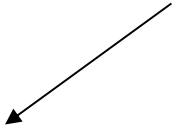
- interface클래스의 멤버변수
  - 메소드 구현이 없으므로 일반적 변수는 가질 수 없음
  - 단, 정적 상수는 가질 수 있음

- interface 클래스의 정적변수 예

```
public interface IMoveable
{
    int move(int dir, int leng);
}
```



```
public interface IMoveable
{
    public static final int EAST = 1;
    public static final int WEST = 2;
    public static final int SOUTH = 3;
    public static final int NORTH =4;

    int move(int dir, int leng);
}
```



```
// interface 정적변수의 사용

IMoveable mv = new CleaningRobot();

mv.move( IMoveable.EAST, 20);
```

interface 클래스

- interface의 상속
  - interface는 interface간 상속이 가능함

- interface 상속의 예

```
public interface IMoveable
{
    int move(int dir, int leng);
}
```

```
public interface IExtendedMoveable extends IMoveable
{
    public static final int SPEED_LIMIT = 120;
    int move(int dir, int leng, int speed);
}
```

interface 클래스

- TimerTest 예제

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

class TimerPrinter implements ActionListner
{
    public void actionPerformed(ActionEvent event)
    {
        Date now = new Date();
        System.out.println("At the tone, the time is " + now );
        Toolkit.getDefaultToolkit().beep();
     }
}
```

```
public class TimerTest
{
    public static void main(String[] args)
    {
        ActionListner listner=new TimerPrinter();

        //리스너를 호출하는 타이머 생성
        // 10초마다 한번
        Timer t = new Timer(10000, listner);
         t.start();

         JOptionPane.showMessageDialog(null, "Quit?");
         System.exit(0);
    }
}
```

내부 클래스(Inner Class)

- 내부 클래스의 정의
  - 다른 클래스의 내부에 정의되어 있는 클래스
  - 같은 패키지 내의 다른 클래스로 부터 은닉하기 위해 사용

- Inner 클래스의 사용 예

```
public class Student
{

    .....
     class Address
     {
          String province;
          String city;
          String number;
     }
     ...

    Address addr;
}
```

Student.java

javac Student.java

Student.class
Student$Address.class

상속방지 : final 클래스와 메소드

- final
  - 클래스를 상속하여 새로운 클래스를 만들수 없도록 함
  - 특정 클래스 또는 메소드에 적용가능
    - String 클래스는 final임

```
final class TimerPrinter implements ActionListner
{
    public final void actionPerformed(ActionEvent event)
    {
        Date now = new Date();
        System.out.println("At the tone, the time is " + now );
        Toolkit.getDefaultToolkit().beep();
     }
}
```