

COMP217

JAVA Programming

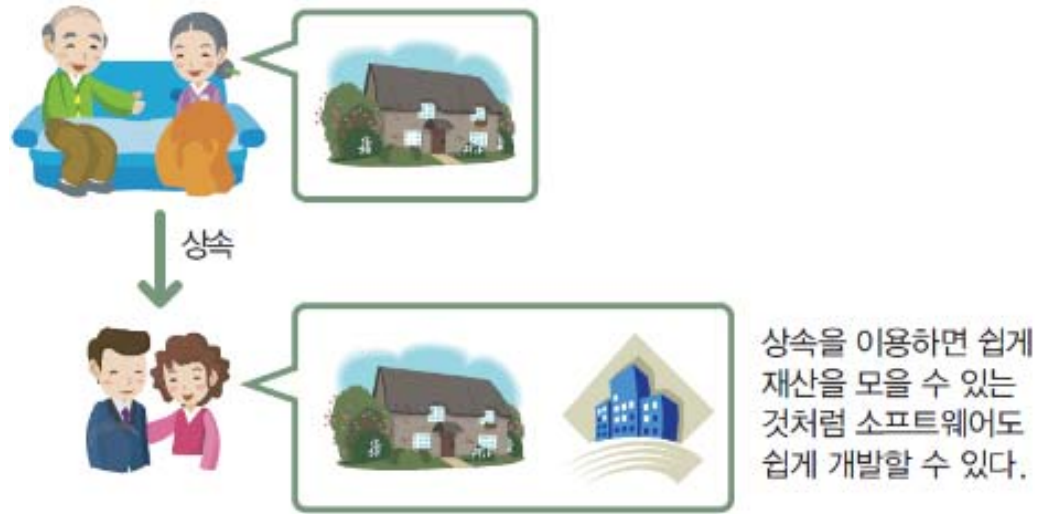
Summer 2018

Day 11

Inheritance, object class

Goals

- 오늘 배우게 될 내용 :
 - 상속이란?
 - 상속의 사용
 - 메소드 재정의
 - 접근 지정자
 - 상속과 생성자
 - Object 클래스
 - 종단 클래스
- 파워 자바 11장(상속)



Concept of Inheritance

Receiving useful features from parents

상속의 장점

- 상속의 장점
 - 상속을 통하여 기존 클래스의 필드와 메소드를 재사용
 - 기존 클래스의 일부 변경도 가능
 - 상속을 이용하게 되면 복잡한 GUI 프로그램을 순식간에 작성
 - 상속은 이미 작성된 검증된 소프트웨어를 재사용
 - 신뢰성 있는 소프트웨어를 손쉽게 개발, 유지 보수
 - 코드의 중복을 줄일 수 있다.

상속

```
class SubClass extends SuperClass
{
    ...// 추가된 메소드와 필드
}
```

상속을 의미한다. 슈퍼 클래스를 확장하여
서브 클래스를 작성한다는 의미이다.

슈퍼 클래스 == 부모 클래스

```
class Car
{
    ...
}
class SportsCar extends Car
{
    ...
}
```



슈퍼 클래스(superclass)

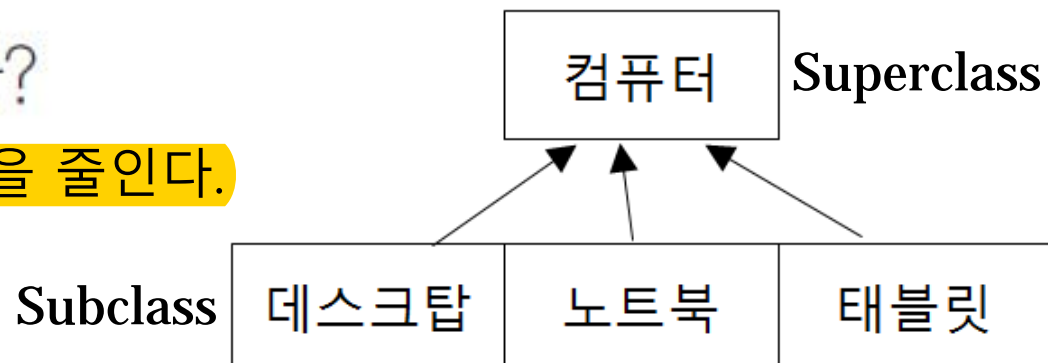
서브 클래스(subclass)

상속

수퍼 클래스	서브 클래스
Animal(동물)	Lion(사자), Dog(개), Cat(고양이)
Bike(자전거)	MountainBike(산악자전거)
Vehicle(탈것)	Car(자동차), Bus(버스), Truck(트럭), Boat(보트), Motorcycle(오토바이), Bicycle(자전거)
Student(학생)	GraduateStudent(대학원생), UnderGraduate(학부생)
Employee(직원)	Manager(관리자)
Shape(도형)	Rectangle(사각형), Triangle(삼각형), Circle(원)

1. 컴퓨터, 데스크탑, 노트북, 태블릿 사이의 상속 관계를 결정하여 보자,
2. 상속의 장점은 무엇인가?

코드를 재사용하여 중복을 줄인다.

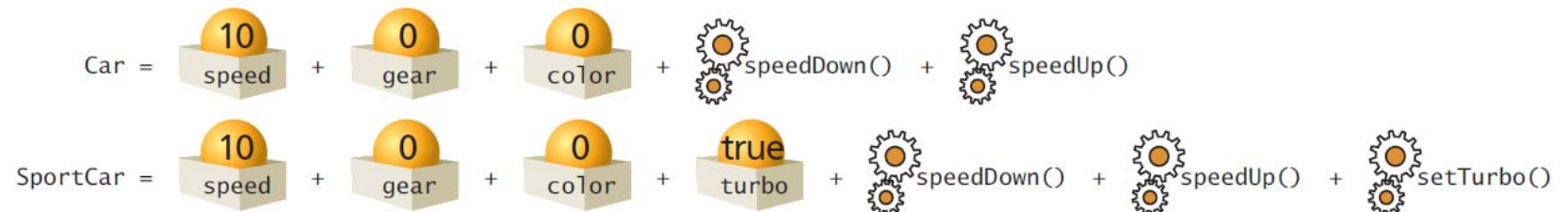
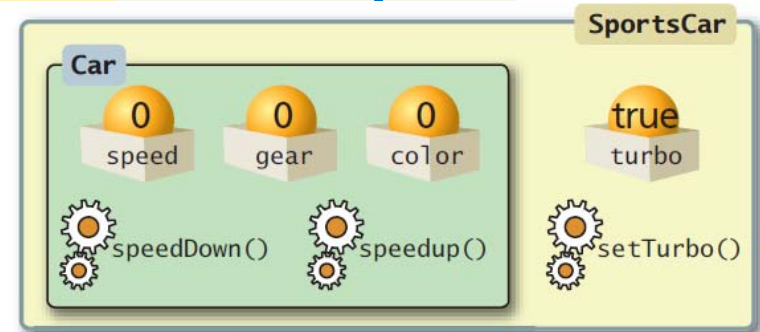


상속 사용

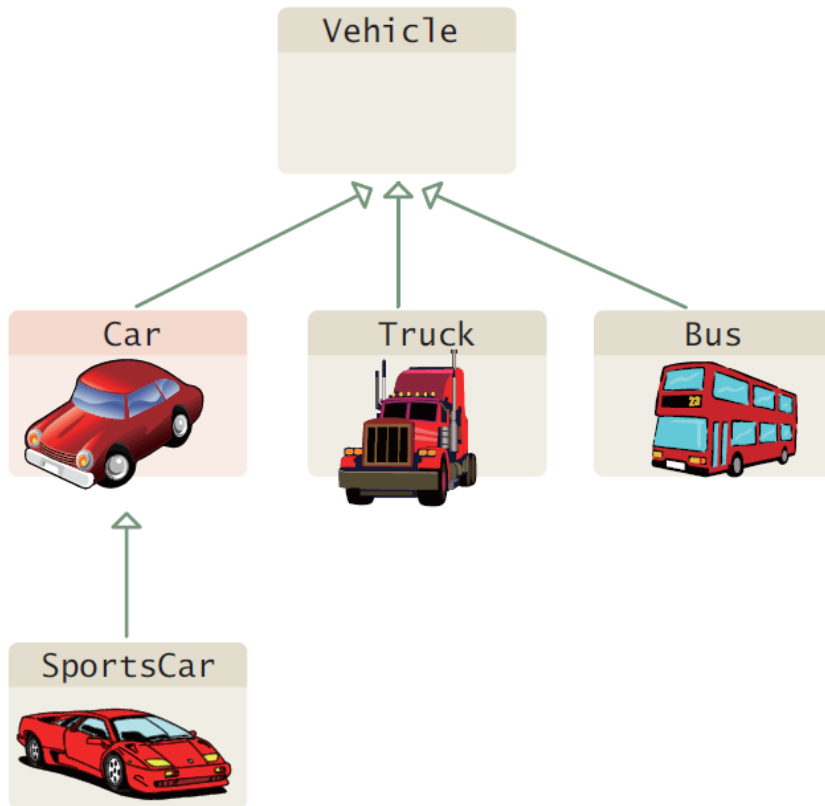
```
public class Car {  
    // 3개의 필드 선언  
    int speed;           // 속도  
    int gear;           // 기어  
    public String color; // 색상, 테스트를 위하여 공용 필드로 만들자.  
  
    public void speedUp(int increment) { // 속도 증가  
        speed += increment;  
    }  
    public void speedDown(int decrement) { // 속도 감소  
        speed -= decrement;  
    }  
}
```

```
public class Test {  
    // 서브 클래스 객체 생성  
    public static void main(String[] args) {  
        SportsCar c = new SportsCar();  
        c.color = "Red";  
        c.speedUp(100);  
        c.speedDown(30);  
        c.setTurbo(true);  
    }  
}
```

```
class SportsCar extends Car { // Car를 상속받는다.  
    boolean turbo;  
  
    public void setTurbo(boolean newValue) { // 터보 모드 설정 메소드  
        turbo = newValue;  
    }  
}
```



상속의 계층 구조



```
class Vehicle { ... }  
class Car extends Vehicle { ... }  
class Truck extends Vehicle { ... }  
class Bus extends Vehicle { ... }  
class SportsCar extends Car { ... }
```

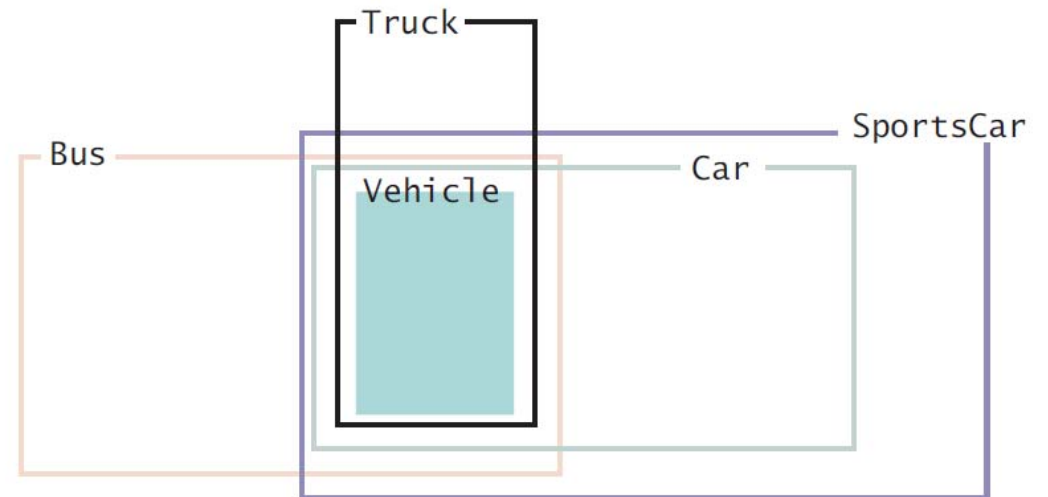


그림11-3. 상속 계층 구조도

상속은 중복을 줄인다.

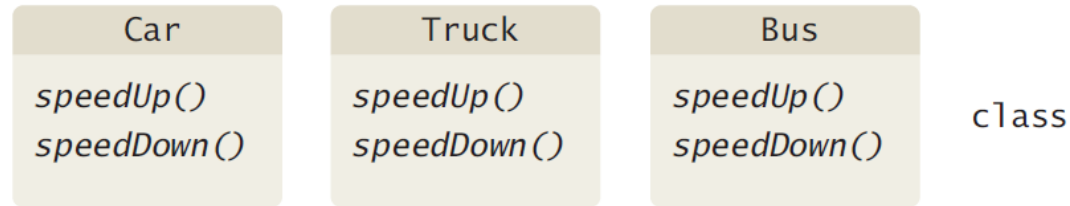


그림11-5. 각 클래스에 코드가 중복된다.

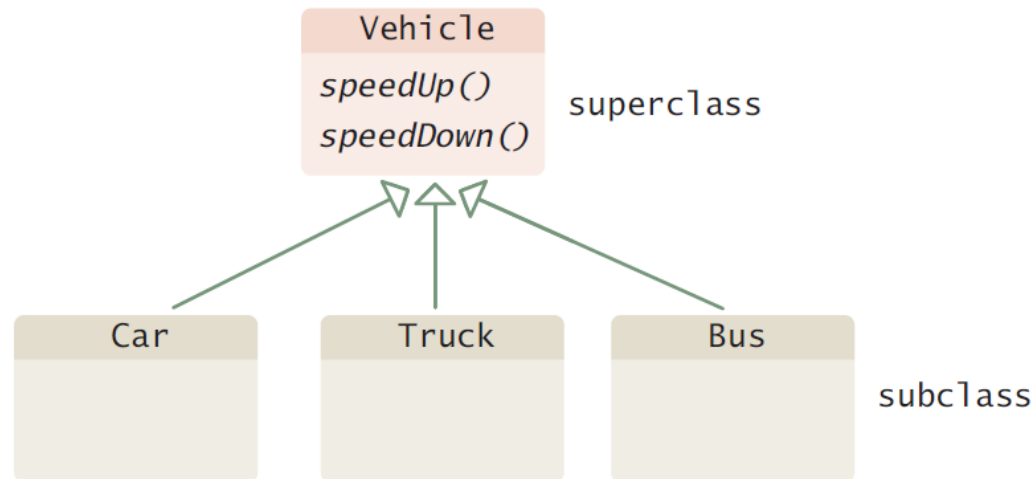


그림11-6. 중복되는 코드는 수퍼 클래스에 모은다.

상속은 is-a관계

- 상속으로 구현
 - 자동차는 탈것이다. (Car **is a** Vehicle).
 - 사자, 개, 고양이는 동물이다.
- 집합관계로 구현
 - 도서관은 책을 가지고 있다(Library **has a** book).
 - 거실은 소파를 가지고 있다.

```
class Point {  
    int x;  
    int y;  
}
```

```
class Line {  
    Point p1; // 객체 포함  
    Point p2; // 객체 포함  
}
```

중간점검

1. Animal 클래스 (sleep(), eat()), Dog 클래스(sleep(), eat(), bark()), Cat 클래스 (sleep(), eat(), play())를 상속을 이용하여서 표현하면 어떻게 코드가 간결해지는가?

sleep()과 eat()가 슈퍼클래스에서만 정의되므로 코드가 간결해진다.

2. 일반적인 상자(box)를 클래스 Box로 표현하고, Box를 상속받는 서브 클래스인 ColorBox(컬러 박스) 클래스를 정의하여 보자. 적절한 필드(길이, 폭, 높이)와 메소드(부피 계산)를 정의한다.

```
class Box {  
    int width, length, height;  
    public int calVolume(){  
        return width*height*height;  
    }  
}  
class ColorBox extends Box {  
    String color;  
}
```

접근 지정자

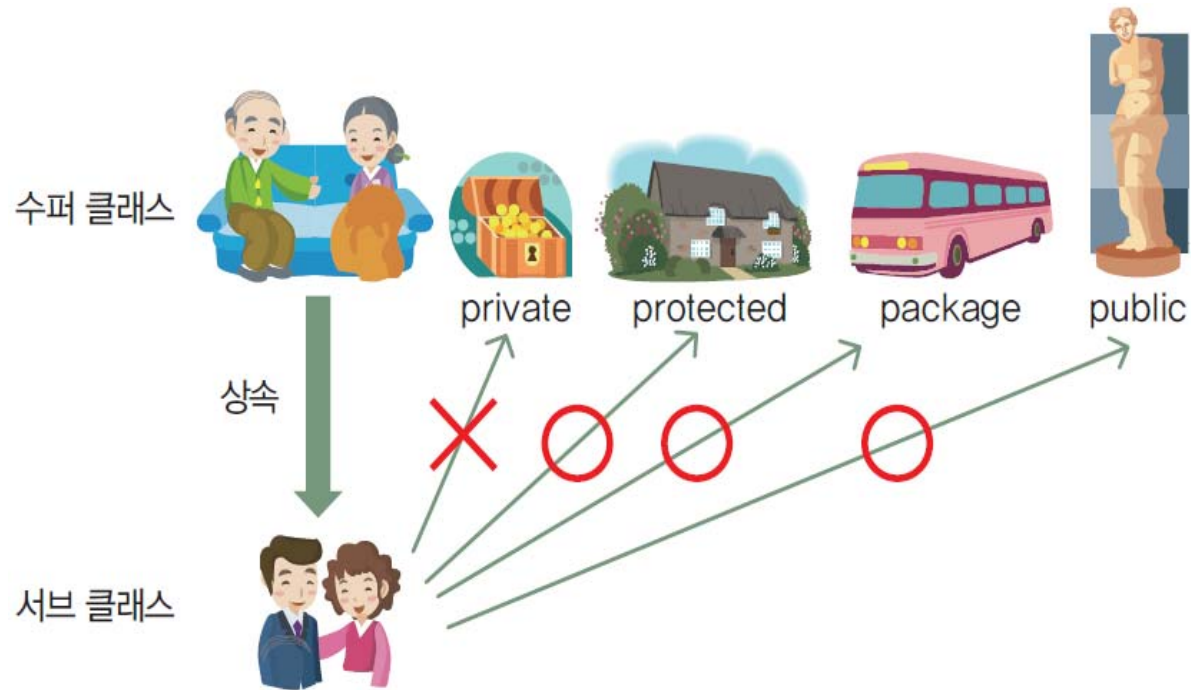


그림11-7. 상속에서 접근 지정자

예제

```

01 class Employee {
02     public String name;        // 이름: 공용 멤버
03     String address;           // 주소: 패키지 멤버
04     protected int salary;      // 월급: 보호 멤버
05     private int RRN;           // 주민등록번호: 전용 멤버
06
07     public String toString() {
08         return name + ", " + address + ", " + RRN + ", " + salary;
09     }
10 }
11
12 class Manager extends Employee {
13     private int bonus;
14
15     public void printSalary() {
16         System.out.println(name + "(" + address + "):" + (salary + bonus));
17     }
18
19     public void printRRN() {
20         System.out.println(RRN);
21     }
22 }
23 public class ManagerTest {
24     public static void main(String[] args) {
25         Manager m = new Manager();
26         m.printRRN();
27     }
28 }

```

수퍼클래스에서 private로 정의된 멤버는 서브 클래스에서 접근할 수 없다.

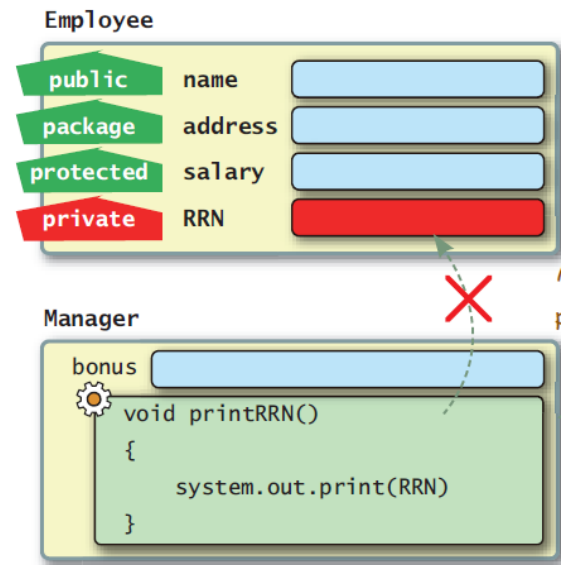
수퍼클래스의 private 멤버를 제외한 모든 멤버 접근 가능

오류! private는 서브 클래스에서 접근 못함!

실행결과

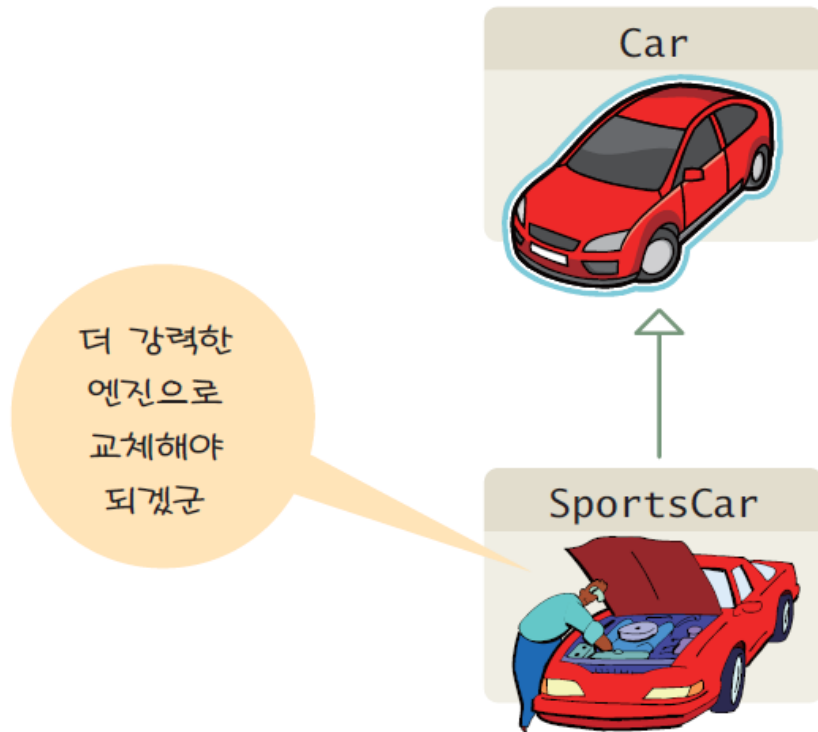
The field Employee.RRN is not visible

ManagerTest.java:20: error: RRN has private access in Employee
System.out.println(RRN);



메소드 재정의

- 메소드 재정의(method overriding): 서브 클래스가 필요에 따라 상속된 메소드를 다시 정의하는 것



메소드 재정의의 예

```
class Animal {  
    public void sound()  
    {  
        // 아직 특정한 동물이 지정되지 않았으므로 몸체는 비어 있다.  
    }  
}
```

```
class Dog extends Animal {  
    public void sound()  
    {  
        System.out.println("멍멍!");  
    }  
}  
  
public class DogTest {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        d.sound();  
    }  
}
```

메소드 재정의

재정의된 메소드가 호출된다.

실행결과

멍멍!

@Override

```
class Dog extends Animal {
```

```
    void saund() {  
        System.out.println("멍멍!");  
    }
```

```
}
```

재정의할 의도하였으나 이름을 잘못 입력하였기
때문에 컴파일러는 새로운 메소드 정의로 간주한다.

```
class Dog extends Animal {
```

```
    @Override  
    void saund() {                // 오류 발생!  
        System.out.println("멍멍!");  
    }
```

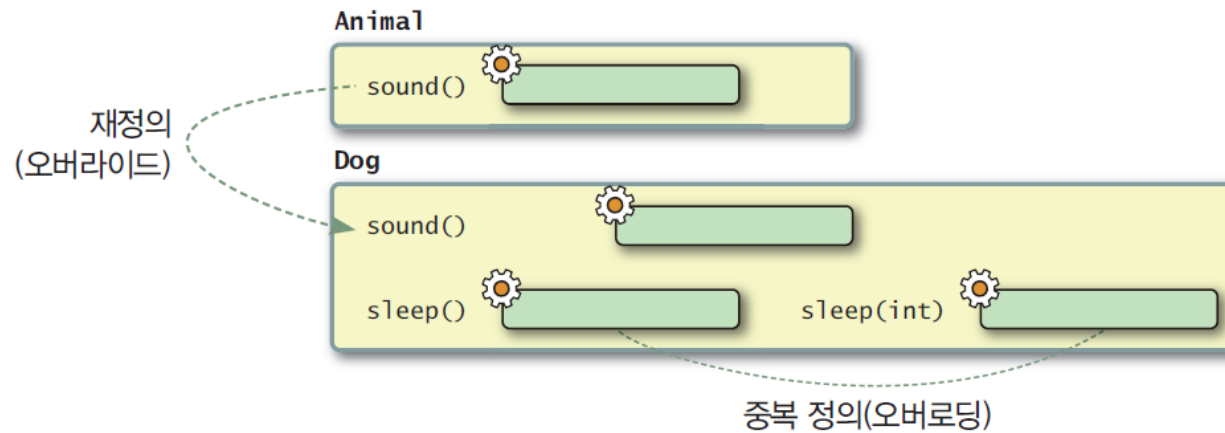
```
}
```

재정의를 의도하였다는 것을 확실하게
컴파일러에게 전달하여 오류를 막는다.

실행결과

The method saund() of type Dog must override or implement a supertype method

중복 정의와 재정의의 차이



```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

Super


```
class ParentClass {  
    int data=100;  
    public void print() {  
        System.out.println("수퍼 클래스의 print() 메소드");  
    }  
}
```

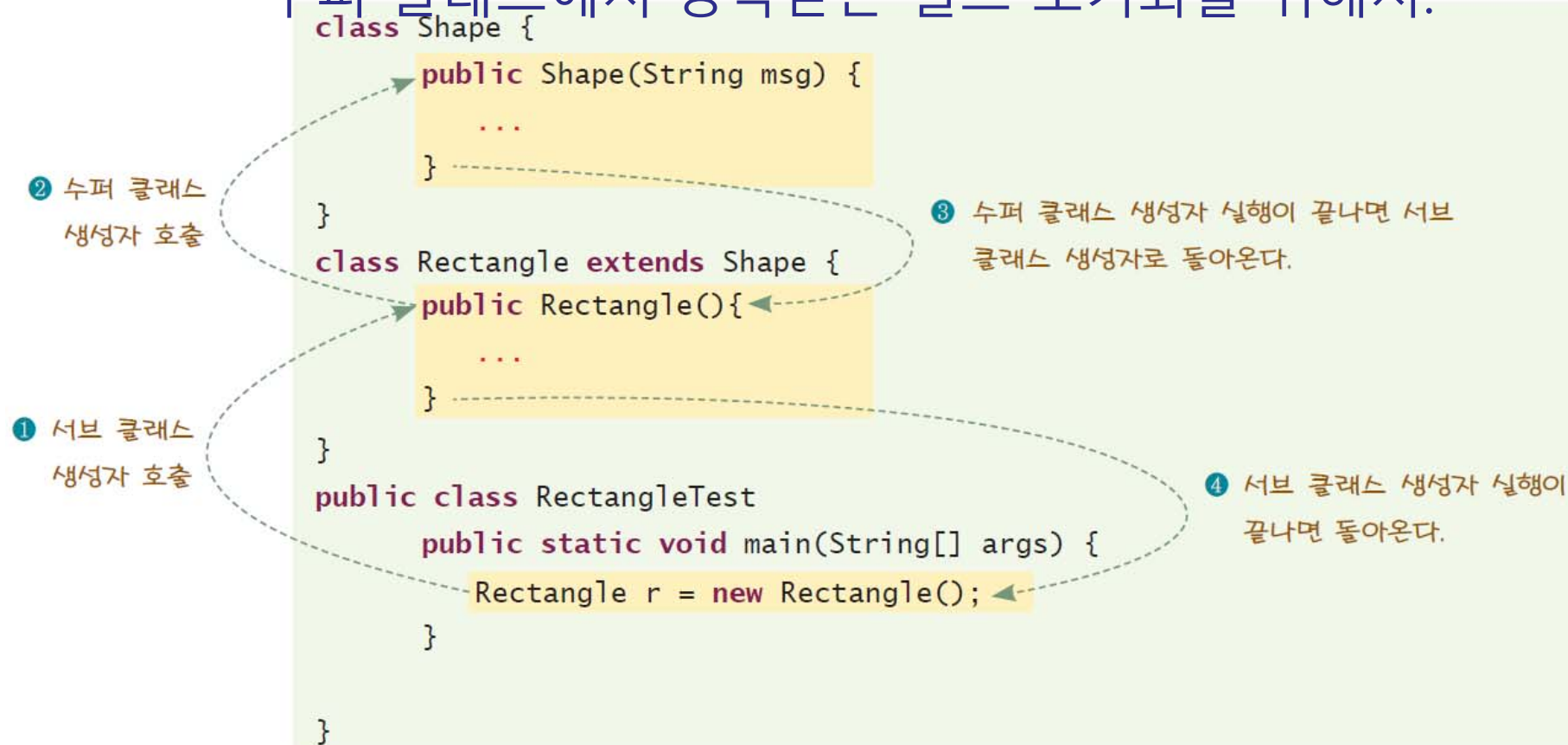
```
public class ChildClass extends ParentClass {  
    int data=200;   
    public void print() { //메소드 재정의  
        super.print();  수퍼 클래스의 메소드 호출  
        System.out.println("서브 클래스의 print() 메소드 ");  
        System.out.println(this.data);  
        System.out.println(super.data);  수퍼 클래스의 필드 접근  
    }  
    public static void main(String[] args) {  
        ChildClass obj = new ChildClass();  
        obj.print();  
    }  
}
```

실행결과

```
수퍼 클래스의 print() 메소드  
서브 클래스의 print() 메소드  
200  
100
```

상속과 생성자

- 서브 클래스의 객체가 생성될 때
 - 서브 클래스의 생성자만 호출될까? 
 - 수퍼 클래스의 생성자도 호출되는가?
 - 수퍼 클래스에서 상속받은 필드 초기화를 위해서.



명시적인 호출

- super를 이용하여서 명시적으로 수퍼 클래스의 생성자 호출

```
class Shape {  
    public Shape(String msg) {  
        System.out.println("Shape 생성자() " + msg);  
    }  
}
```

new Rectangle();

```
public class Rectangle extends Shape {  
    public Rectangle() {  
        super("from Rectangle");  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

실행결과

// 명시적인 호출

실행결과

```
Shape 생성자 from Rectangle  
Rectangle 생성자
```

묵시적인 호출

```
class Shape {  
    public Shape() {  
        System.out.println("Shape 생성자()");  
    }  
}  
class Rectangle extends Shape {  
    public Rectangle() {  
        Shape ( ) ;  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

new Rectangle();

컴파일러가 Shape();을 자동적으로 넣어준다고 생각하라.

실행결과

Shape 생성자
Rectangle 생성자

묵시적인 호출

```
class Shape {  
    Shape() {} // OK! 자동으로 디폴트 생성자 추가!  
}
```

new Rectangle();

```
public class Rectangle extends Shape {  
    public Rectangle() {  
        Shape();  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

Shape();을 자동적으로
넣어준다고 생각하라.

실행결과

Rectangle 생성자()

```
class Shape {  
    public Shape(String msg) { // 디폴트 생성자는 없음! new Rectangle();  
        System.out.println("Shape 생성자()" + msg);  
    }  
}
```

디폴트 생성자 Shape()을 호출할 수 없기
때문에 컴파일 오류가 발생한다.

```
public class Rectangle extends Shape {  
    public Rectangle() { Shape(); // 오류: Shape()를 호출할 수 없음!  
        System.out.println("Rectangle 생성자()");  
    }  
}
```

실행결과


Implicit super constructor Shape() is undefined. Must explicitly invoke another constructor

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as *constructor chaining*.

```
public class Faculty extends Employee {
    public static void main(String[] args) { new Faculty(); }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

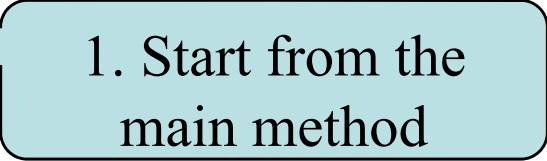
class Employee extends Person {
    public Employee() { 
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) { System.out.println(s); }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



1. Start from the main method

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Invoke Faculty constructor

3. Invoke Employee's no-arg constructor

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

6. Invoke Employee(String) constructor

4. Invoke Person() constructor

5. Execute println

Trace Execution

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }
}
```

- THE FOLLOWING IS THE EXECUTION OF THE PROGRAM:
- (1) Person's no-arg constructor is invoked
 - (2) Invoke Employee's overloaded constructor
 - (3) Employee's no-arg constructor is invoked
 - (4) Faculty's no-arg constructor is invoked

```
    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}
```

9. Execute println

```
class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }
}
```

8. Execute println

```
    public Employee(String s) {
        System.out.println(s);
    }
}
```

7. Execute println

```
class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

Object class

Origin of all classes

Object 클래스

- Object 클래스는 java.lang 패키지에 들어 있으며 자바 클래스 계층 구조에서 맨 위에 위치하는 클래스

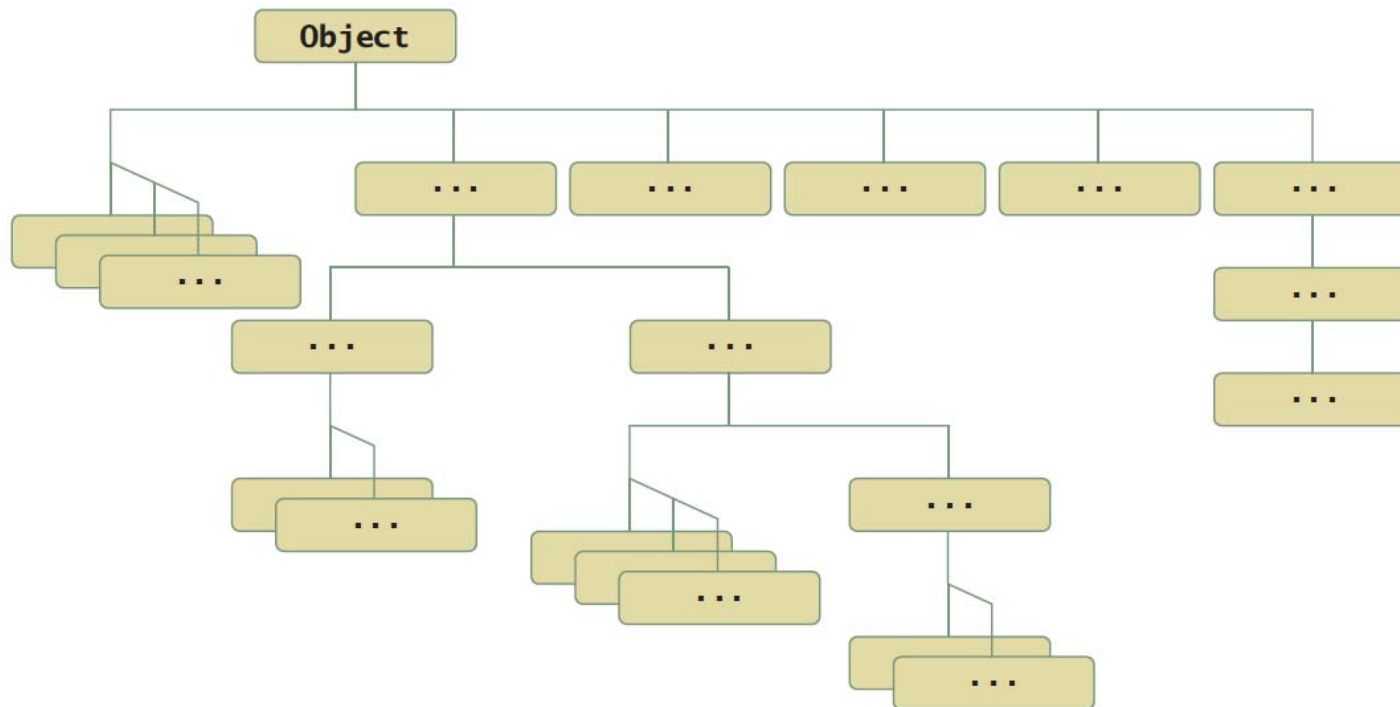


그림11-9.Object 클래스는 상속 계층 구조의 맨 위에 있다(출처: java.sun.com)

Object의 메소드

- **protected Object clone()**
 - 객체 자신의 복사본을 생성하여 반환한다.
- **public boolean equals(Object obj)**
 - obj가 이 객체와 같은지를 나타낸다.
- **protected void finalize()**
 - 가비지 콜렉터에 의하여 호출된다.
- **public final Class getClass()**
 - 객체를 생성한 클래스 정보를 반환한다.
- **public int hashCode()**
 - 객체에 대한 해쉬 코드를 반환한다.
- **public String toString()**
 - 객체의 문자열 표현을 반환한다.

equals(), finalize(), toString() 은 보통 **child class** 에서 **override** 되어야 한다.
추가되는 **member**에 대한 처리를 해주어야 하기 때문.

getClass()

```
class Car {  
    ...  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car obj = new Car();  
        System.out.println("obj is of type " + obj.getClass().getName());  
    }  
}
```

객체를 생성한 클래스 이름을 반환한다.

실행결과

obj is of type Car

equals() 메소드

```
class Car {  
    private String model;  
    public Car(String model) {        this.model= model;    }  
    public boolean equals(Object obj) {  
        if (obj instanceof Car)  
            return model.equals(((Car) obj).model);  
        else  
            return false;  
    }  
}
```

← equals()를 재정의한다. String의 equals()를 호출하여서 문자열이 동일한지를 검사한다.

```
public class CarTest {  
    public static void main(String[] args) {  
        Car firstCar = new Car("BMW520");  
        Car secondCar = new Car("BMW520");  
        if (firstCar.equals(secondCar)) {  
            System.out.println("동일한 종류의 자동차입니다.");  
        }  
        else {  
            System.out.println("동일한 종류의 자동차가 아닙니다.");  
        }  
    }  
}
```

이 equals() 메소드를 사용하여 검사하는 다음과 같은 코드를 가정할 있다.

실행결과

동일한 종류의 자동차입니다.

method equals()

```
1 class Car {
2     private String model;
3     public Car(String model){
4         this.model= model;
5     }
6     public boolean equals(Object obj) {
7         if (obj instanceof Car)
8             return model.equals(((Car) obj).model);
9         else
10            return false;
11    }
12 }
13 class Bike{
14     private int numOfWeek;
15     Bike(int i){
16         numOfWeek = i;
17     }
18 }
19 public class CarTest {
20     public static void main(String[] args) {
21         Car firstCar = new Car("AVANTE");
22         Car secondCar = new Car("AVANTE");
23         Bike bike = new Bike(4);
24
25         if (firstCar.equals(bike)) { //if (firstCar.equals(secondCar)) {
26             System.out.println("동일한 종류의 자동차입니다.");
27         } else {
28             System.out.println("동일한 종류의 자동차가 아닙니다.");
29         }
30     }
31 }
```

toString() 메소드

- Object 클래스의 toString() 메소드는 객체의 문자열 표현을 반환

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

실행결과

모델 HMW 520...

```
class Car {  
    private String model;  
    public Car(String model) {        this.model= model  
    public boolean equals(Object obj) {  
        if (obj instanceof Car)  
            return model.equals(((Car) obj).model);  
        else  
            return false;  
    }  
}
```

```
public String toString(){  
    return "모델"+this.model+"...";  
}
```

종단 클래스와 종단 메소드

- Class에 키워드 `final`을 붙이면 더 이상 상속할 수 없다.
- 메서드에 키워드 `final`을 붙이면 메서드를 재정의할 수 없다.

```
final class String {  
    ...  
}
```

```
class Baduk {  
    enum BadukPlayer { WHITE, BLACK }  
    ...  
    final BadukPlayer getFirstPlayer() {  
        return BadukPlayer.BLACK;  
    }  
}
```

서브 클래스에서 재정의할 수
없도록 `final`로 지정한다.