

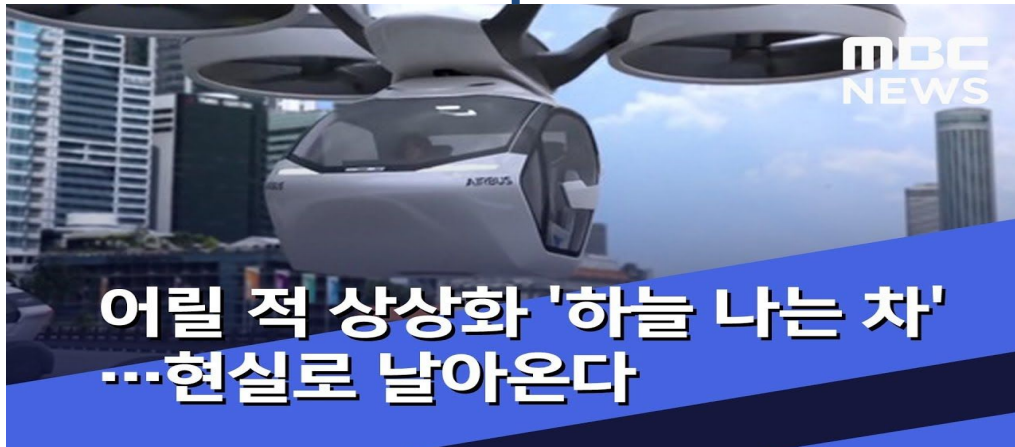
2025년 상반기 K-디지털 트레이닝

상속

[KB] IT's Your Life



↑ 상속



달리다(){

//바퀴가 지면위에서 앞으로 가게 하다.

}

기능의 재정의(오버라이드)

달리다(){

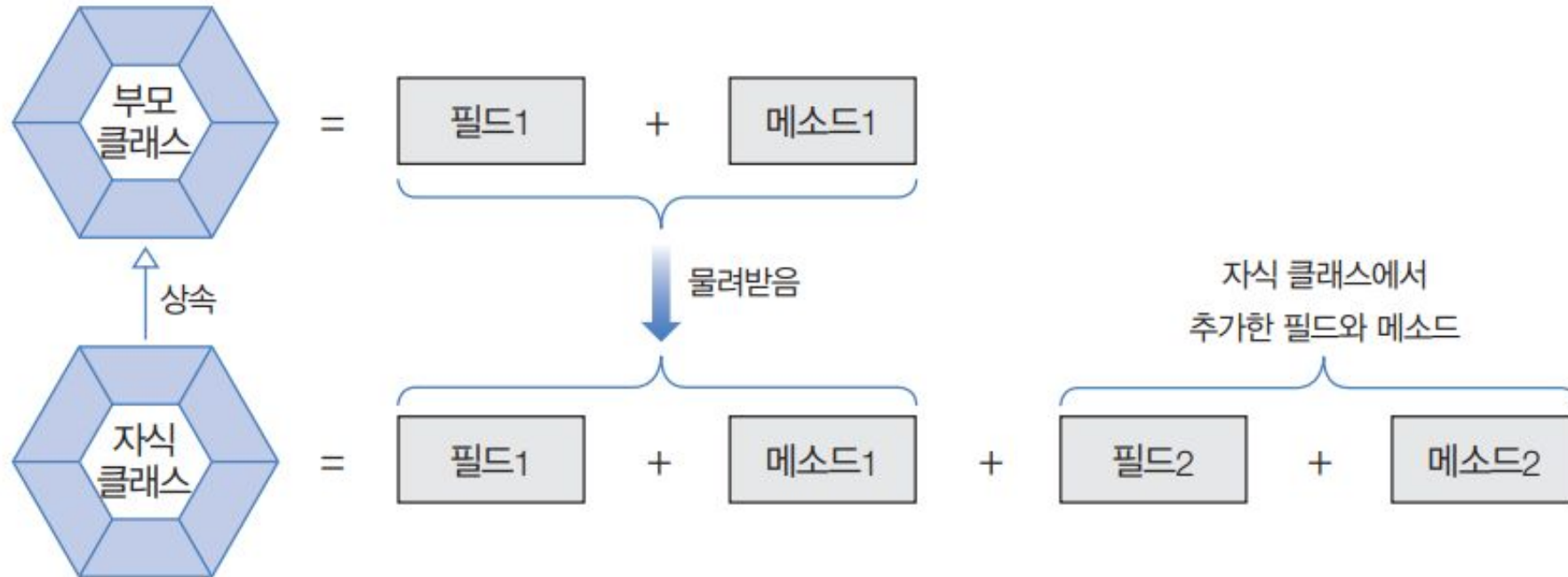
//공중에서 앞으로 나아가다.

}

1 상속 개념

- 상속

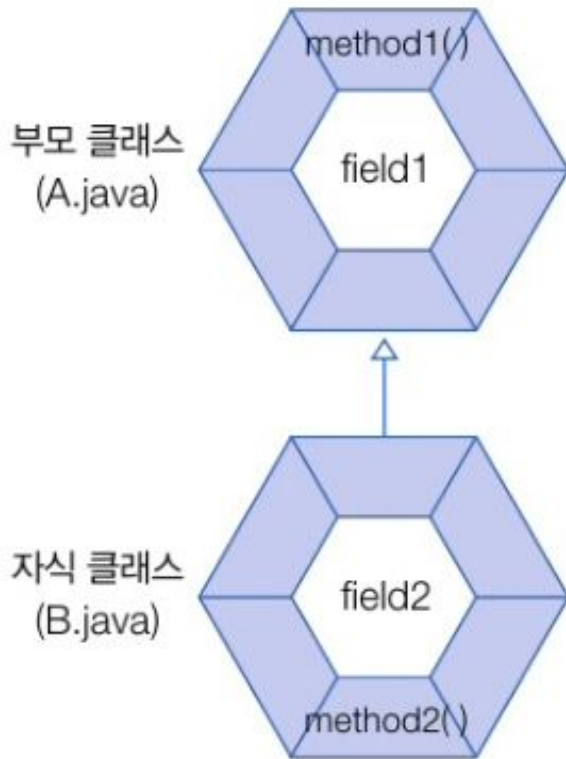
- 부모 클래스의 필드와 메소드를 자식 클래스에게 물려줄 수 있음



1 상속 개념

• 상속의 이점

- 이미 개발된 클래스를 재사용하므로 중복 코드를 줄임
- 클래스 수정을 최소화



```
public class A {  
    int field1;  
    void method1() { ... }  
}
```

A를 상속

```
public class B extends A {  
    String field2;  
    void method2() { ... }  
}
```

```
B b = new B();  
b.field1 = 10;  
b.method1();
```

} A로부터 물려받은 필드와 메소드

```
b.field2 = "홍길동";  
method2();
```

} B가 추가한 필드와 메소드

- 클래스 상속

- 자식 클래스를 선언할 때 어떤 부모로부터 상속받을 것인지를 결정하고, 부모 클래스를 다음과 같이 `extends` 뒤에 기술

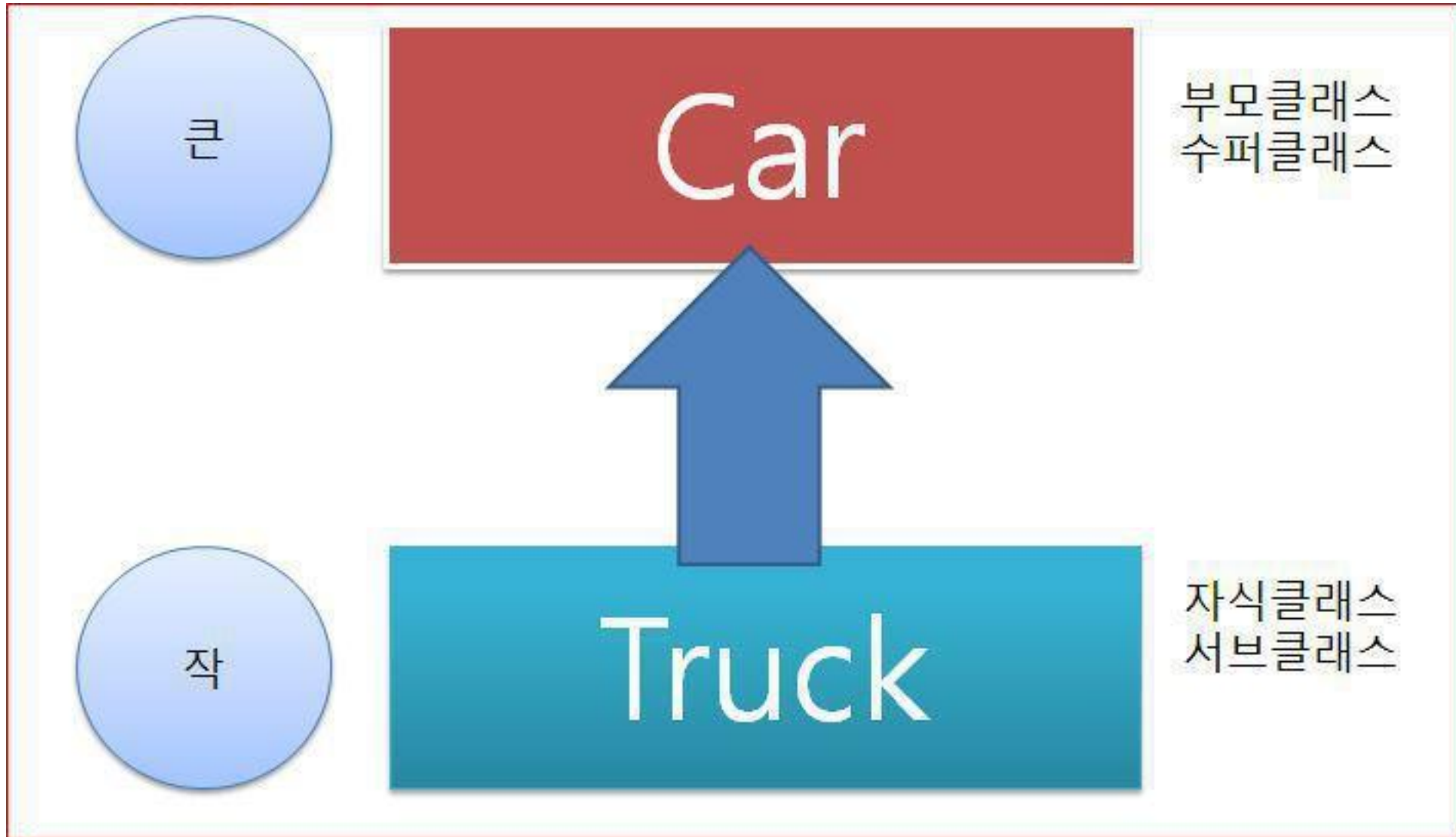
```
public class 자식클래스 extends 부모클래스 {  
}
```

- 다중 상속 허용하지 않음. `extends` 뒤에 하나의 부모 클래스만 상속

```
public class 자식클래스 extends 부모클래스1, 부모클래스2 {  
}
```

상속인 경우 크기 비교

- 개념의 크기 자식이 부모이다.(is~a)



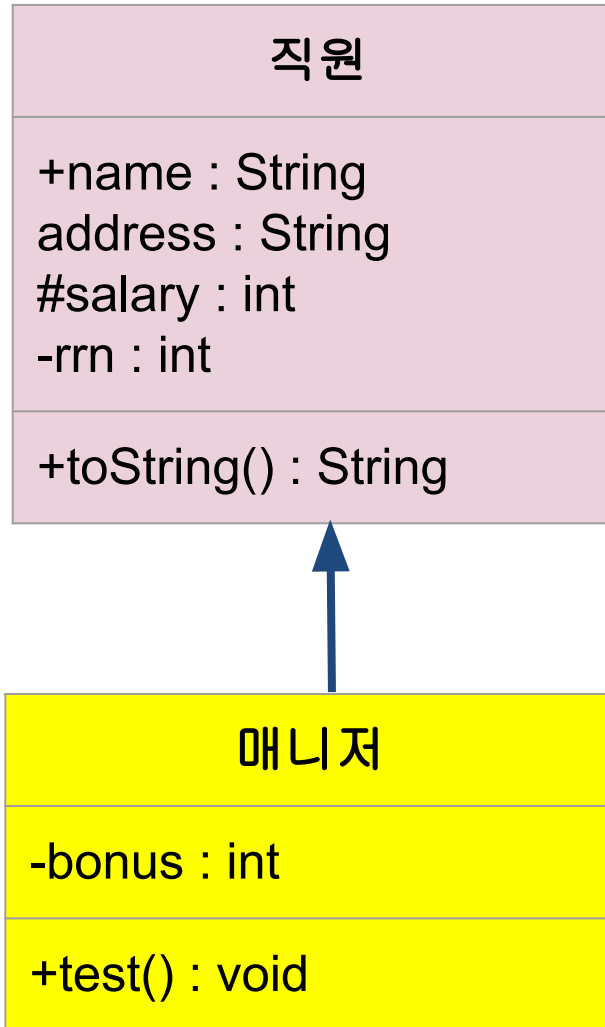
상속과 관련된 용어 정리

부모클래스 / 상위클래스

Parent Class
Super Class

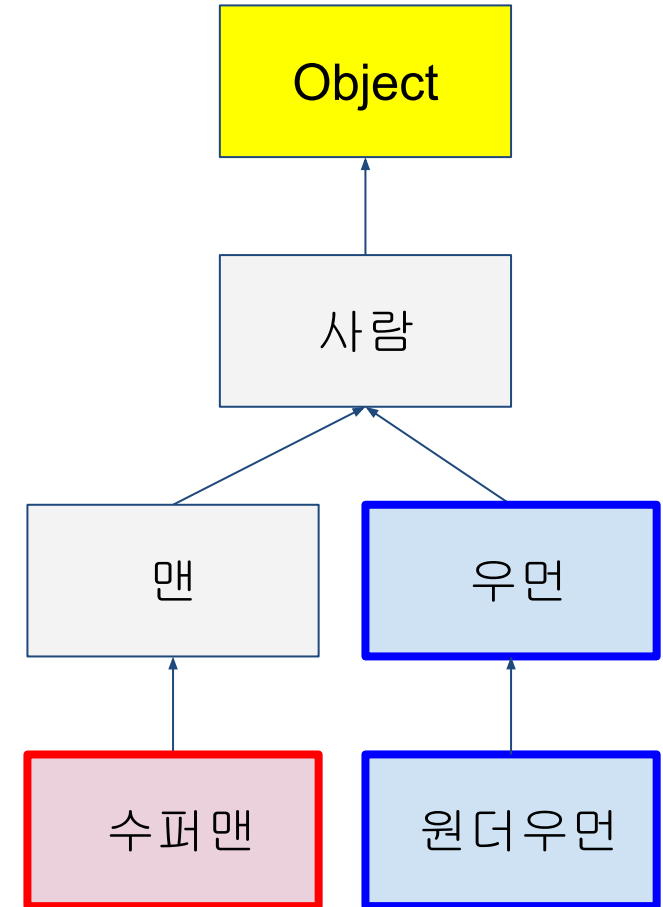
자식클래스 / 하위클래스

Child Class
Sub Class

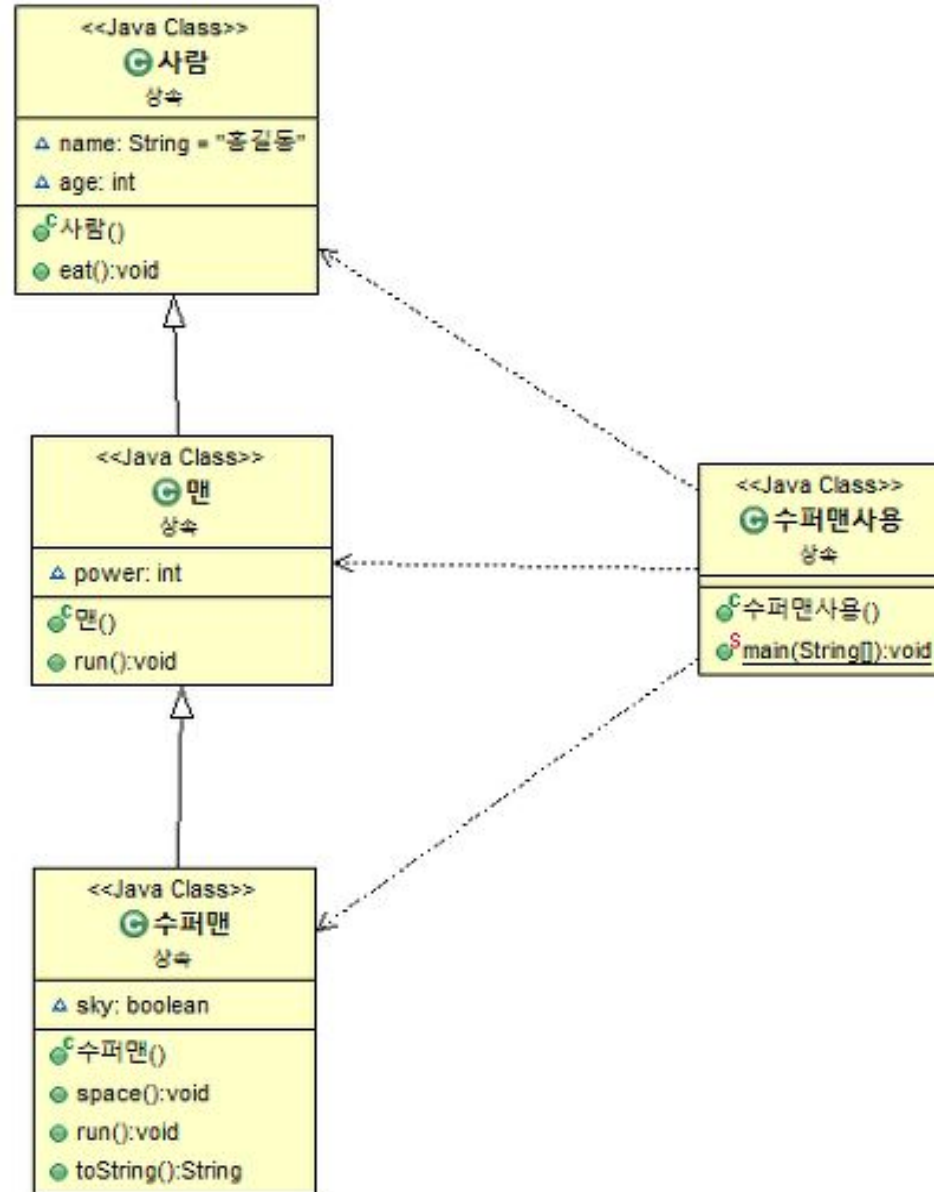


우먼 class와 원더우먼 class을 만들어보자.

사람 - 성별, 이름
 - 잠자다, 먹다
 맨 = 사람 + 맨 특징
 - 힘크기
 - 달리다
 슈퍼맨 = 맨 + 슈퍼맨의 특징
 - 날아다니다.
 스파이더맨 = 맨 + 스파이더맨의 특징
 - 벽을 탄다
 우먼사용!! 변수+메소드 호출 확인



클래스 다이어그램



● Phone.java

```
package ch07.sec02;

public class Phone {
    //필드 선언
    public String model;
    public String color;

    //메소드 선언
    public void bell() {
        System.out.println("벨이 울립니다.");
    }

    public void sendVoice(String message) {
        System.out.println("자기: " + message);
    }

    public void receiveVoice(String message) {
        System.out.println("상대방: " + message);
    }

    public void hangUp() {
        System.out.println("전화를 끊습니다.");
    }
}
```

- **SmartPhone.java**

```
package ch07.sec02;

public class SmartPhone extends Phone {
    //필드 선언
    public boolean wifi;

    //생성자 선언
    public SmartPhone(String model, String color) {
        this.model = model;
        this.color = color;
    }

    //메소드 선언
    public void setWifi(boolean wifi) {
        this.wifi = wifi;
        System.out.println("와이파이 상태를 변경했습니다.");
    }

    public void internet() {
        System.out.println("인터넷에 연결합니다.");
    }
}
```

- **SmartPhoneExample.java**

```
package ch07.sec02;

public class SmartPhoneExample {

    public static void main(String[] args) {
        //SmartPhone 객체 생성
        SmartPhone myPhone = new SmartPhone("갤럭시", "은색");

        //Phone으로부터 상속받은 필드 읽기
        System.out.println("모델: " + myPhone.model);
        System.out.println("색상: " + myPhone.color);

        //SmartPhone의 필드 읽기
        System.out.println("와이파이 상태: " + myPhone.wifi);

        //Phone으로부터 상속받은 메소드 호출
        myPhone.bell();
        myPhone.sendVoice("여보세요.");
        myPhone.receiveVoice("안녕하세요! 저는 홍길동인데요.");
        myPhone.sendVoice("아~ 네, 반갑습니다.");
        myPhone.hangUp();
    }
}
```

- SmartPhoneExample.java

```
//SmartPhone의 메소드 호출  
myPhone.setWifi(true);  
myPhone.internet();  
}  
}
```

모델: 갤럭시
색상: 은색
와이파이 상태: false
벨이 울립니다.
자기: 여보세요.
상대방: 안녕하세요! 저는 홍길동인데요.
자기: 아~ 네, 반갑습니다.
전화를 끊습니다.
와이파이 상태를 변경했습니다.
인터넷에 연결합니다.

3 부모 생성자 호출

• 부모 생성자 호출

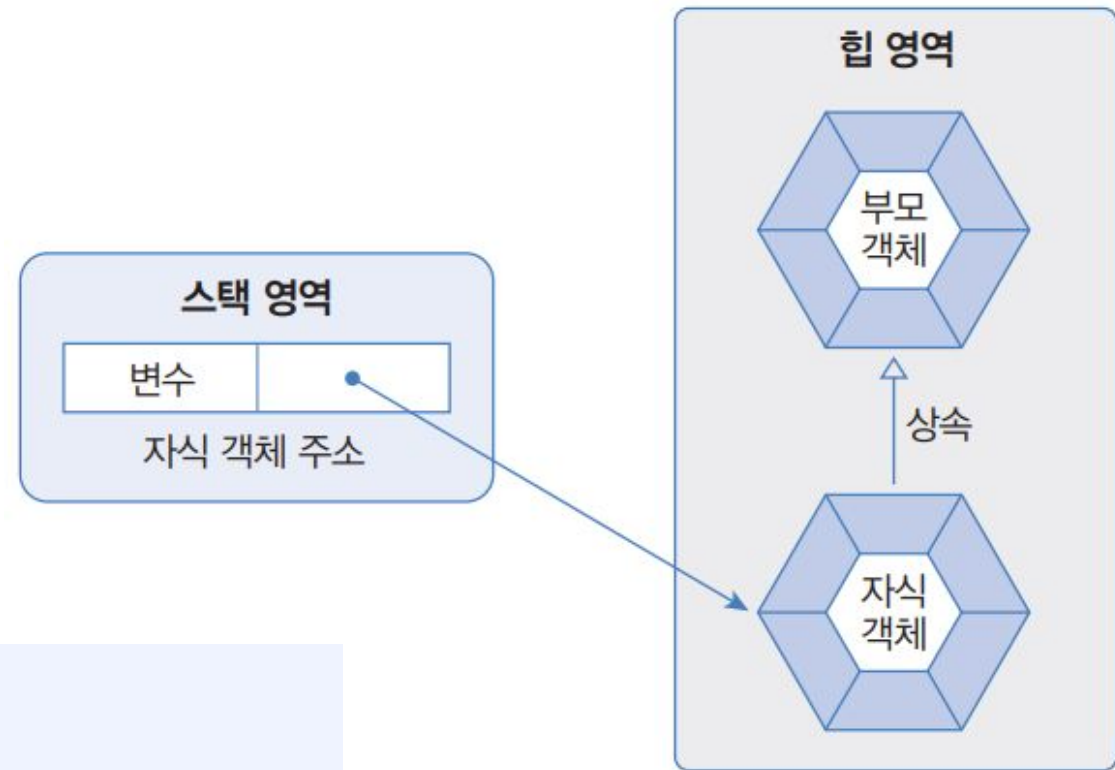
- 자식 객체를 생성하면 부모 객체가 먼저 생성된 다음에 자식 객체가 생성

```
자식클래스 변수 = new 자식클래스( );
```

○

```
//자식 생성자 선언
public 자식클래스(...) {
    super();
    ...
}
```

```
//자식 생성자 선언
public 자식클래스(...) {
    super(매개값, ... );
    ...
}
```



- **Phone.java**

```
package ch07.sec03.exam01;

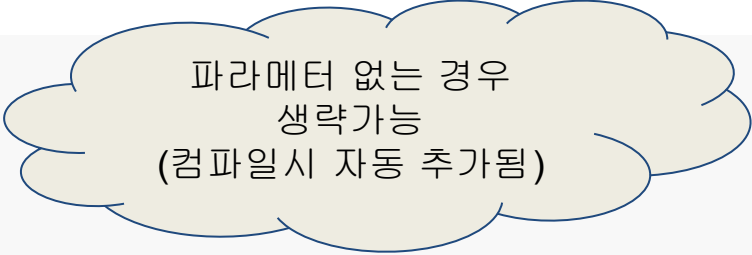
public class Phone {
    //필드 선언
    public String model;
    public String color;

    //기본 생성자 선언
    public Phone() {
        System.out.println("Phone() 생성자 실행");
    }
}
```

- **SmartPhone.java**

```
package ch07.sec03.exam01;

public class SmartPhone extends Phone {
    //자식 생성자 선언
    public SmartPhone(String model, String color) {
        super();
        this.model = model;
        this.color = color;
        System.out.println("SmartPhone(String model, String color) 생성자 실행됨");
    }
}
```



파라미터 없는 경우
생략가능
(컴파일시 자동 추가됨)

- **SmartPhoneExample.java**

```
package ch07.sec03.exam01;

public class SmartPhoneExample {

    public static void main(String[] args) {
        //SmartPhone 객체 생성
        SmartPhone myPhone = new SmartPhone("갤럭시", "은색");

        //Phone으로부터 상속 받은 필드 읽기
        System.out.println("모델: " + myPhone.model);
        System.out.println("색상: " + myPhone.color);
    }
}
```

Phone() 생성자 실행

SmartPhone(String model, String color) 생성자 실행됨

모델: 갤럭시

색상: 은색

- Phone.java

```
package ch07.sec03.exam02;

public class Phone {
    //필드 선언
    public String model;
    public String color;

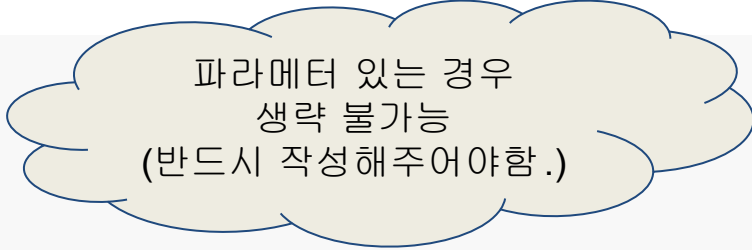
    //매개변수를 갖는 생성자 선언
    public Phone(String model, String color) {
        this.model = model;
        this.color = color;
        System.out.println("Phone(String model, String color) 생성자 실행");
    }
}
```

3 부모 생성자 호출

● SmartPhone.java

```
package ch07.sec03.exam02;

public class SmartPhone extends Phone {
    //자식 생성자 선언
    public SmartPhone(String model, String color) {
        super(model, color);
        System.out.println("SmartPhone(String model, String color) 생성자 실행됨");
    }
}
```



파라미터 있는 경우
생략 불가능
(반드시 작성해주어야함.)

- **SmartPhoneExample.java**

```
package ch07.sec03.exam02;

public class SmartPhoneExample {

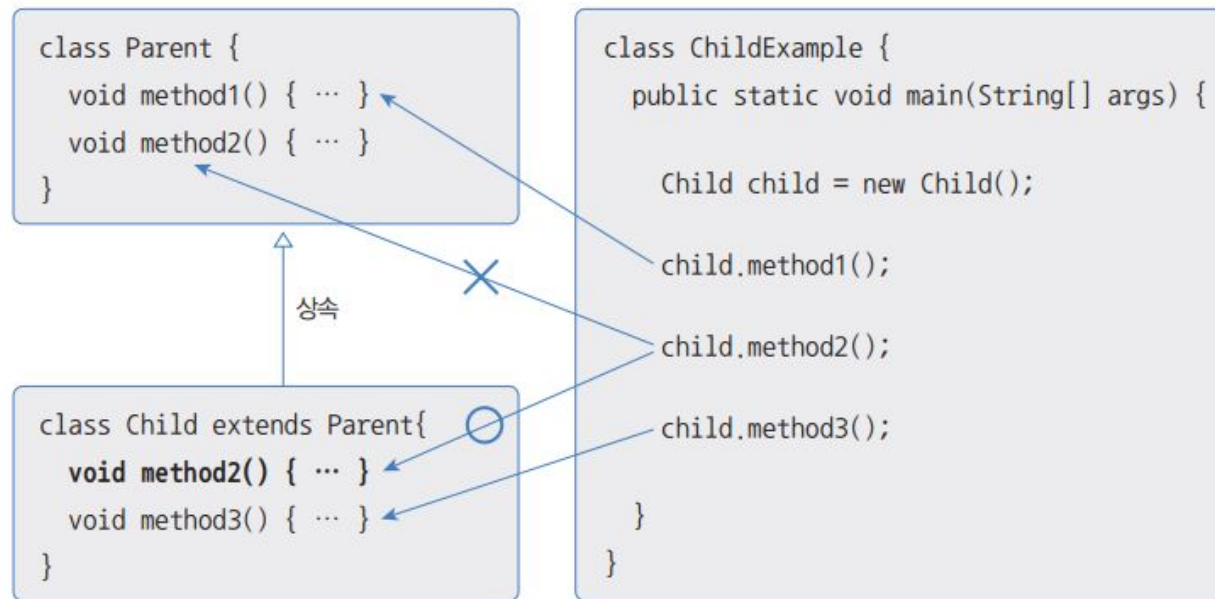
    public static void main(String[] args) {
        //SmartPhone 객체 생성
        SmartPhone myPhone = new SmartPhone("갤럭시", "은색");

        //Phone으로부터 상속 받은 필드 읽기
        System.out.println("모델: " + myPhone.model);
        System.out.println("색상: " + myPhone.color);
    }
}
```

Phone(String model, String color) 생성자 실행
SmartPhone(String model, String color) 생성자 실행됨
모델: 갤럭시
색상: 은색

- 메소드 오버라이딩

- 상속된 메소드를 자식 클래스에서 재정의하는 것.
- 해당 부모 메소드는 숨겨지고, 자식 메소드가 우선적으로 사용



- 부모 메소드의 선언부(리턴 타입, 메소드 이름, 매개변수)와 동일해야 함

- **Calculator.java**

```
package ch07.sec04.exam01;

public class Calculator {
    //메소드 선언
    public double areaCircle(double r) {
        System.out.println("Calculator 객체의 areaCircle() 실행");
        return 3.1459 * r * r;
    }
}
```

- **Computer.java**

```
package ch07.sec04.exam01;

public class Computer extends Calculator {
    //메소드 오버라이딩
    @Override    // 컴파일 시 정확히 오버라이딩이 되었는지 체크 해줌
    public double areaCircle(double r) {
        System.out.println("Computer 객체의 areaCircle() 실행");
        return Math.PI * r * r;
    }
}
```

- **ComputerExample.java**

```
package ch07.sec04.exam01;

public class ComputerExample {
    public static void main(String[] args) {
        int r = 10;

        Calculator calculator = new Calculator();
        System.out.println("원 면적: " + calculator.areaCircle(r));
        System.out.println();

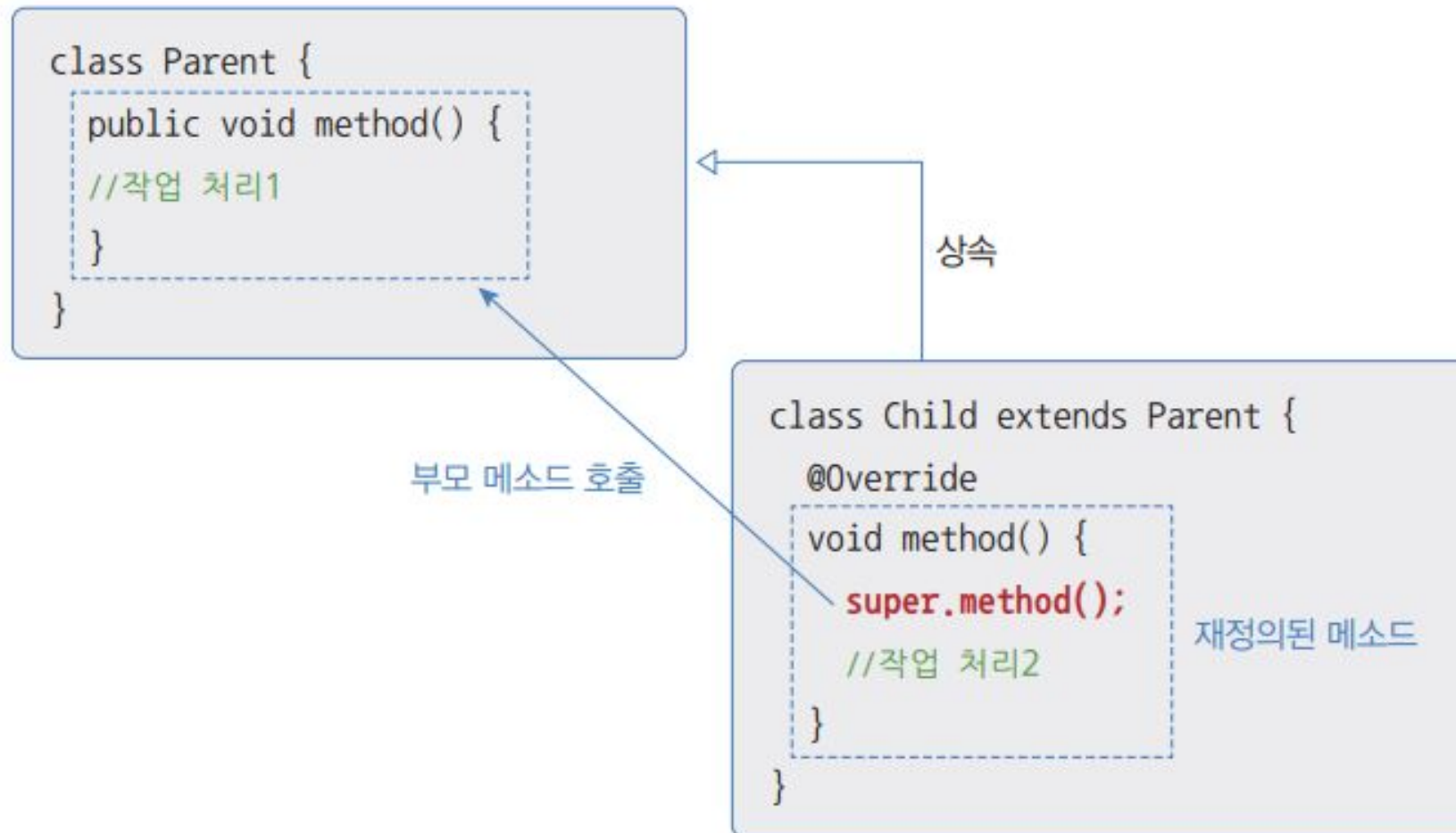
        Computer computer = new Computer();
        System.out.println("원 면적: " + computer.areaCircle(r));
    }
}
```

Calculator 객체의 areaCircle() 실행
원 면적: 314.159

Computer 객체의 areaCircle() 실행
원 면적: 314.1592653589793

- 부모 메소드 호출

- 자식 메소드 내에서 **super** 키워드와 도트(.) 연산자를 사용하면 숨겨진 부모 메소드를 호출
- 부모 메소드를 재사용함으로써 자식 메소드의 중복 작업 내용을 없애는 효과



- **Airplane.java**

```
package ch07.sec04.exam02;

public class Airplane {
    //메소드 선언
    public void land() {
        System.out.println("착륙합니다.");
    }

    public void fly() {
        System.out.println("일반 비행합니다.");
    }

    public void takeOff() {
        System.out.println("이륙합니다.");
    }
}
```

- **SupersonicAirplane.java**

```
package ch07.sec04.exam02;

public class SupersonicAirplane extends Airplane {
    //상수 선언
    public static final int NORMAL = 1;
    public static final int SUPERSONIC = 2;
    //상태 필드 선언
    public int flyMode = NORMAL;

    //메소드 재정의
    @Override
    public void fly() {
        if(flyMode == SUPERSONIC) {
            System.out.println("초음속 비행합니다.");
        } else {
            //Airplane 객체의 fly() 메소드 호출
            super.fly();
        }
    }
}
```

- **ComputerExample.java**

```
package ch07.sec04.exam02;

public class SupersonicAirplaneExample {
    public static void main(String[] args) {
        SupersonicAirplane sa = new SupersonicAirplane();
        sa.takeOff();
        sa.fly();
        sa.flyMode = SupersonicAirplane.SUPERSONIC;
        sa.fly();
        sa.flyMode = SupersonicAirplane.NORMAL;
        sa.fly();
        sa.land();
    }
}
```

이륙합니다.
일반 비행합니다.
초음속 비행합니다.
일반 비행합니다.
착륙합니다.

- **final 클래스**

- final 클래스는 부모 클래스가 될 수 없어 자식 클래스를 만들 수 없음

```
public final class 클래스 { ... }
```

- **final 메서드**

- 메소드를 선언할 때 **final** 키워드를 붙이면 오버라이딩할 수 없음
- 부모 클래스를 상속해서 자식 클래스를 선언할 때,
부모 클래스에 선언된 **final** 메소드는 자식 클래스에서 재정의할 수 없음

```
public final 리턴타입 메소드( 매개변수, ... ) { ... }
```

- Member.java

```
package ch07.sec05.exam01;  
  
public final class Member {  
}
```

- VeryImportantPerson.java

```
package ch07.sec05.exam01;  
  
public class VeryImportantPerson extends Member {  
}
```

// 컴파일 에러 상속 불가

- **Car.java**

```
package ch07.sec05.exam02;

public class Car {
    //필드 선언
    public int speed;

    //메소드 선언
    public void speedUp() {
        speed += 1;
    }

    //final 메소드
    public final void stop() {
        System.out.println("차를 멈춤");
        speed = 0;
    }
}
```

- Member.java

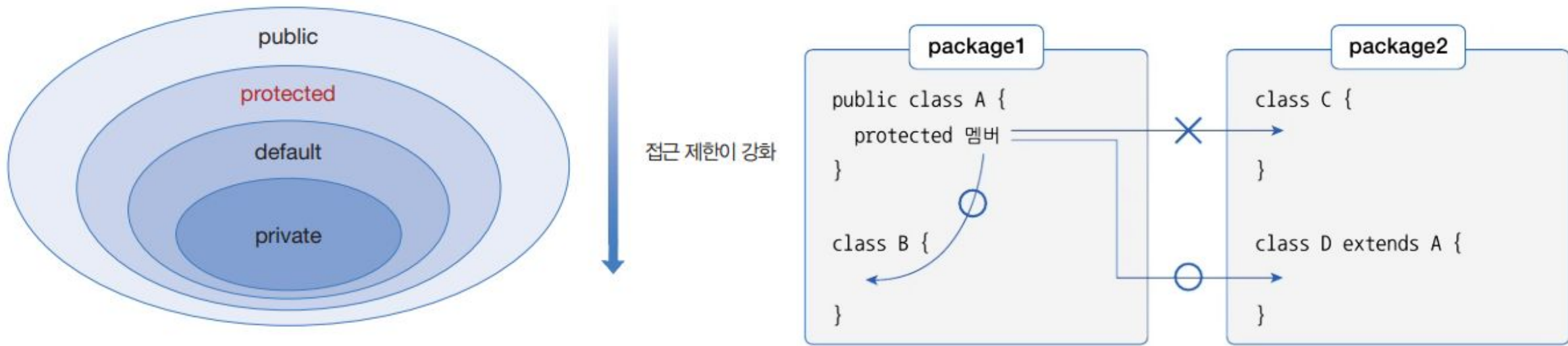
```
package ch07.sec05.exam02;

public class SportsCar extends Car {
    @Override
    public void speedUp() {
        speed += 10;
    }

    // 컴파일 에러, final 메서드는 오버라이딩을 할 수 없음
    @Override
    public void stop() {
        System.out.println("스포츠카를 멈춤");
        speed = 0;
    }
}
```


• protected 접근 제한자

- **protected**는 상속과 관련이 있고, **public**과 **default**의 중간쯤에 해당하는 접근 제한
- **protected**는 같은 패키지에서는 **default**처럼 접근이 가능하나, 다른 패키지에서는 자식 클래스만 접근을 허용



NOTE ▶ default는 접근 제한자가 아니라 접근 제한자가 붙지 않은 상태를 말한다.

접근 제한자	제한 대상	제한 범위
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능

protected 접근 제한자

- package1.A.java

```
package ch07.sec06.package1;
```

```
public class A {
    //필드 선언
    protected String field;

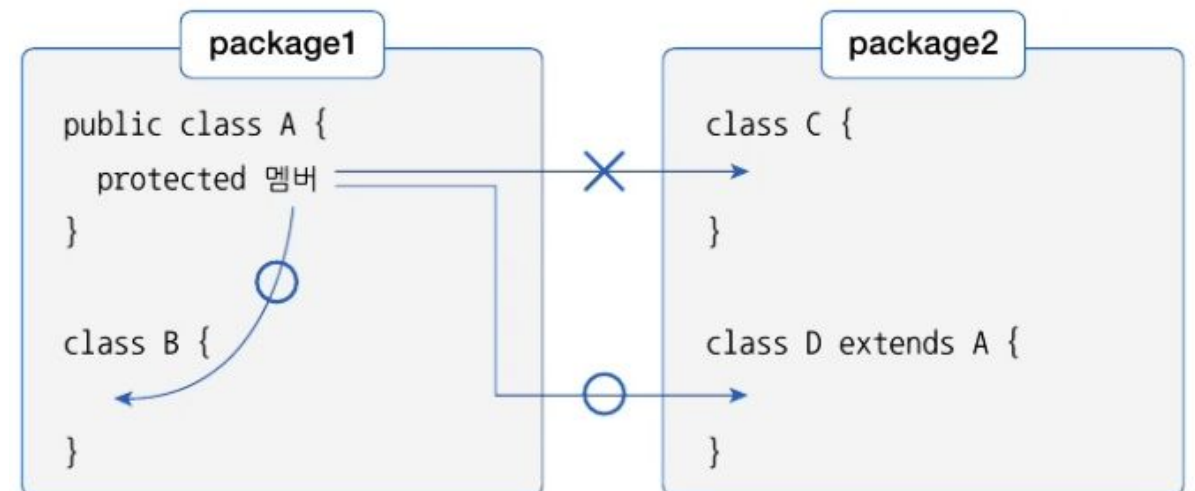
    //생성자 선언
    protected A() {
    }

    //메소드 선언
    protected void method() {
    }
}
```

- package1.B.java

```
package ch07.sec06.package1;
```

```
public class B {
    //메소드 선언
    public void method() {
        A a = new A();           //o
        a.field = "value";        //o
        a.method();               //o
    }
}
```



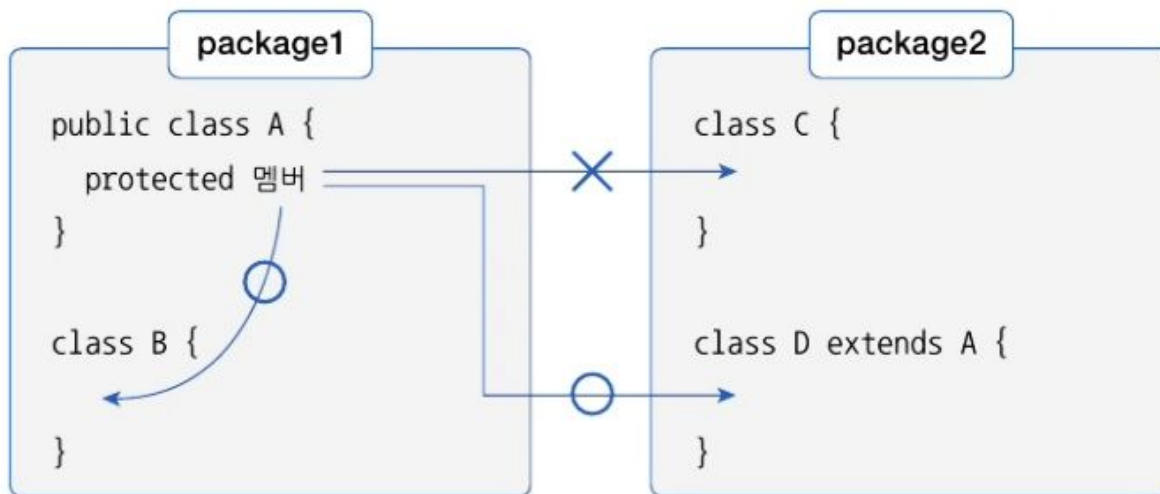
protected 접근 제한자

- package2.C.java

```
package ch07.sec06.package2;
```

```
import ch07.sec06.package1.A;
```

```
public class C {
    //메소드 선언
    public void method() {
        //A a = new A();           //x
        //a.field = "value";       //x
        //a.method();              //x
    }
}
```



- package2.D.java

```
package ch07.sec06.package2;
```

```
import ch07.sec06.package1.A;
```

```
public class D extends A {
    public D() {
        //A() 생성자 호출
        super();           //o
    }
```

```
    public void method1() {
        //A 필드값 변경
        this.field = "value"; //o
        //A 메소드 호출
        this.method();       //o
    }
```

상속을 통해서만
가능

```
    //메소드 선언
    public void method2() {
        //A a = new A(); //x
        //a.field = "value"; //x
        //a.method();    //x
    }
}
```

직접 객체
생성해서
사용하는 것은
안됨

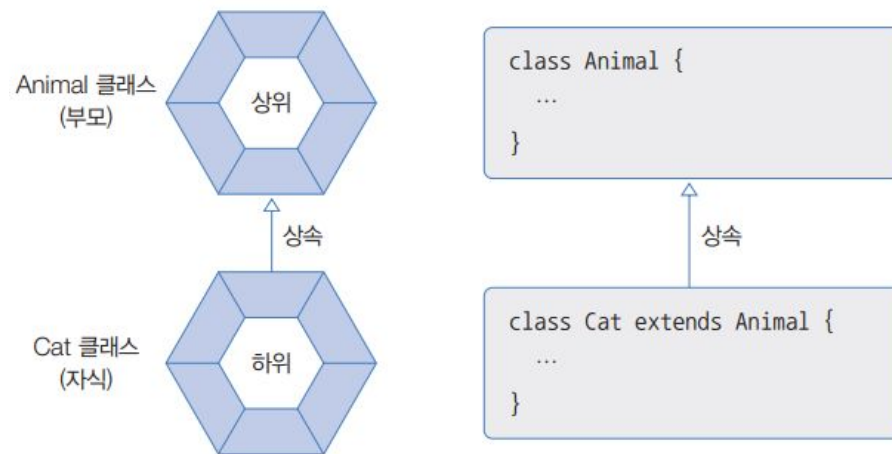
7 타입 변환

• 자동 타입 변환

- 자동적으로 타입 변환이 일어나는 것



- 자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급

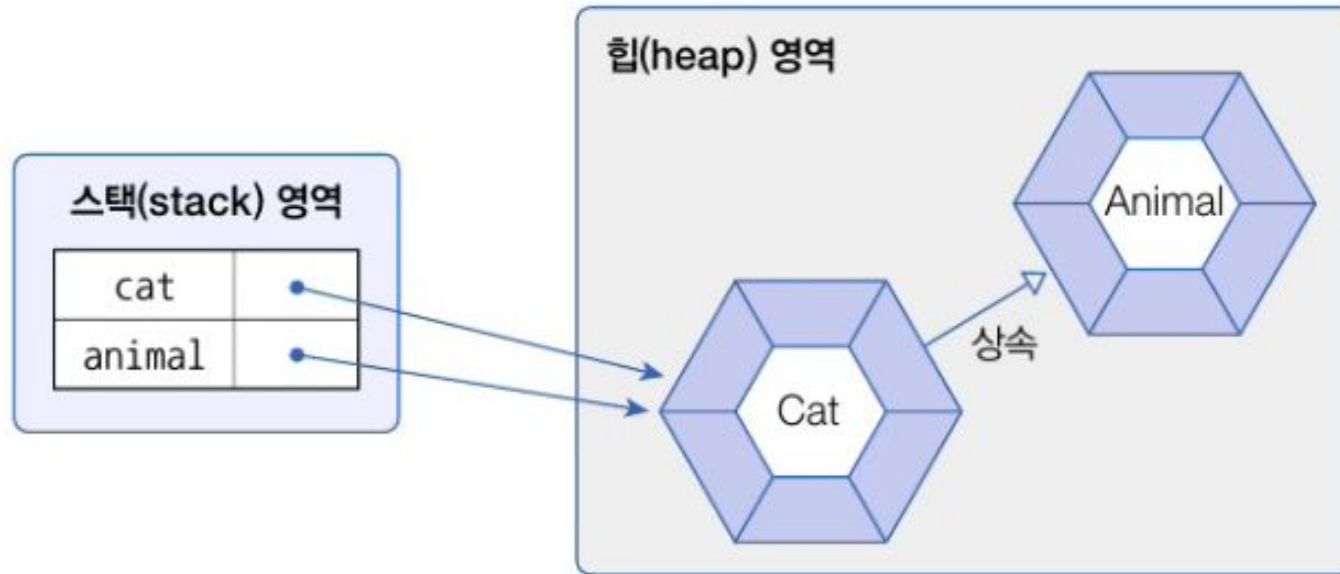


7 타입 변환

• 자동 타입 변환

```
Cat cat = new Cat();  
Animal animal = cat;
```

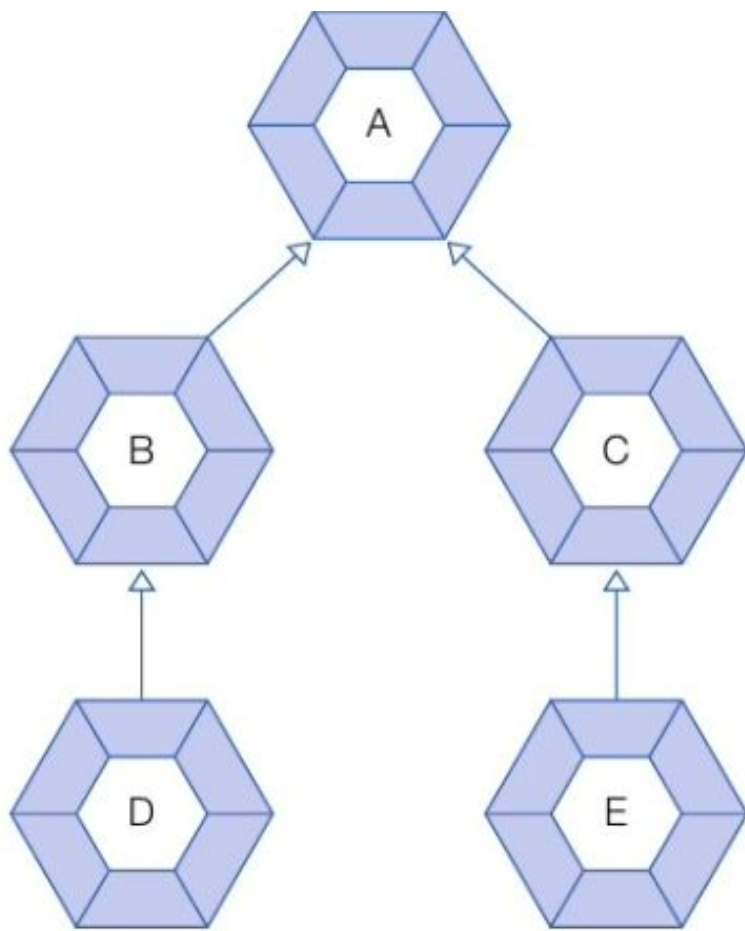
Animal animal = new Cat();도 가능



```
cat == animal //true
```

7 타입 변환

• 자동 타입 변환



```
B b = new B();  
C c = new C();  
D d = new D();  
E e = new E();
```

```
A a1 = b; (가능)  
A a2 = c; (가능)  
A a3 = d; (가능)  
A a4 = e; (가능)
```

```
B b1 = d; (가능)  
C c1 = e; (가능)
```

```
B b3 = e; (불가능)  
C c2 = d; (불가능)
```

● PromotionExample.java

```
package ch07.sec07.exam01;

class A {
}

class B extends A {
}

class C extends A {
}

class D extends B {
}

class E extends C {
}

public class PromotionExample {
    public static void main(String[] args) {
        B b = new B();
        C c = new C();
        D d = new D();
        E e = new E();

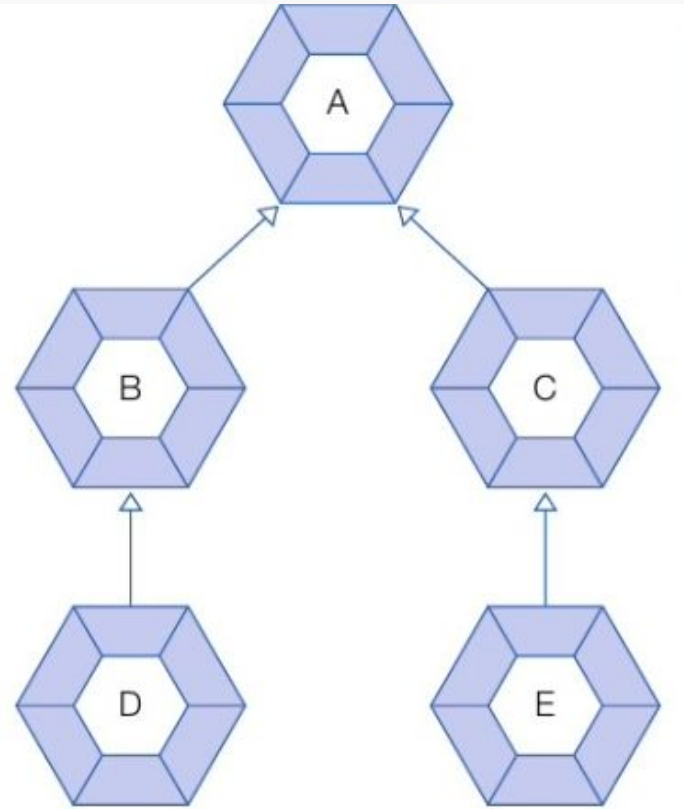
        A a1 = b;
        A a2 = c;
        A a3 = d;
        A a4 = e;

        B b1 = d;
        C c1 = e;

        // B b3 = e;
        // C c2 = d;
    }
}
```

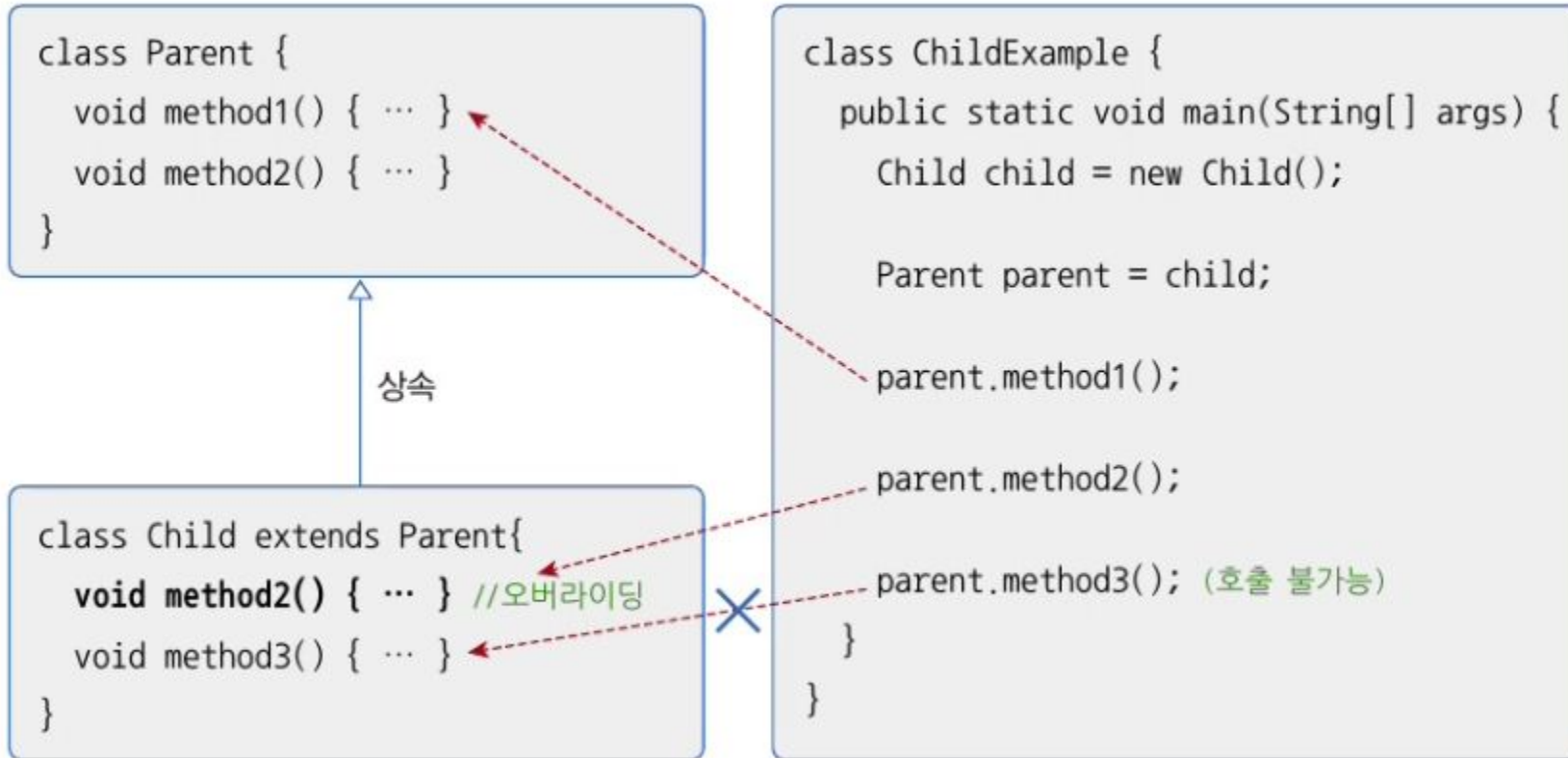
자동 타입 변환(상속 관계에 있음)

컴파일 에러(상속 관계에 있지 않음)



7 타입 변환

• 자동 타입 변환



- **Parent.java**

```
package ch07.sec07.exam02;

public class Parent {
    public void method1() {
        System.out.println("Parent-method1()");
    }

    public void method2() {
        System.out.println("Parent-method2()");
    }
}
```

- **Child.java**

```
package ch07.sec07.exam02;

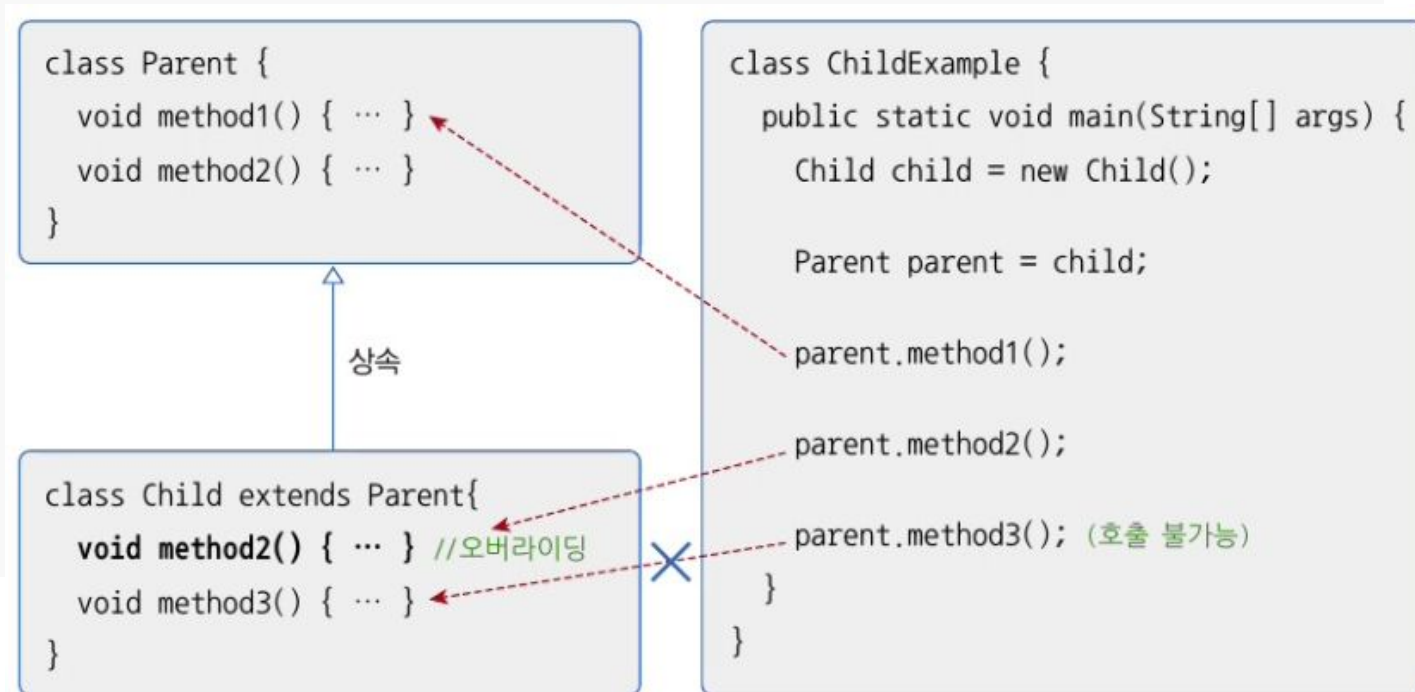
public class Child extends Parent {
    //메소드 오버라이딩
    @Override
    public void method2() {
        System.out.println("Child-method2()");
    }

    //메소드 선언
    public void method3() {
        System.out.println("Child-method3()");
    }
}
```

• Parent.java

```
package ch07.sec07.exam02;
```

```
public class ChildExample {  
    public static void main(String[] args) {  
        //자식 객체 생성  
        Child child = new Child();  
  
        //자동 타입 변환  
        Parent parent = child;  
  
        //메소드 호출  
        parent.method1();  
        parent.method2();  
        //parent.method3(); (호출 불가능)  
    }  
}
```



7 타입 변환

- 강제 타입 변환

- 부모 타입은 자식 타입으로 자동 변환되지 않음. 대신 캐스팅 연산자로 강제 타입 변환 가능

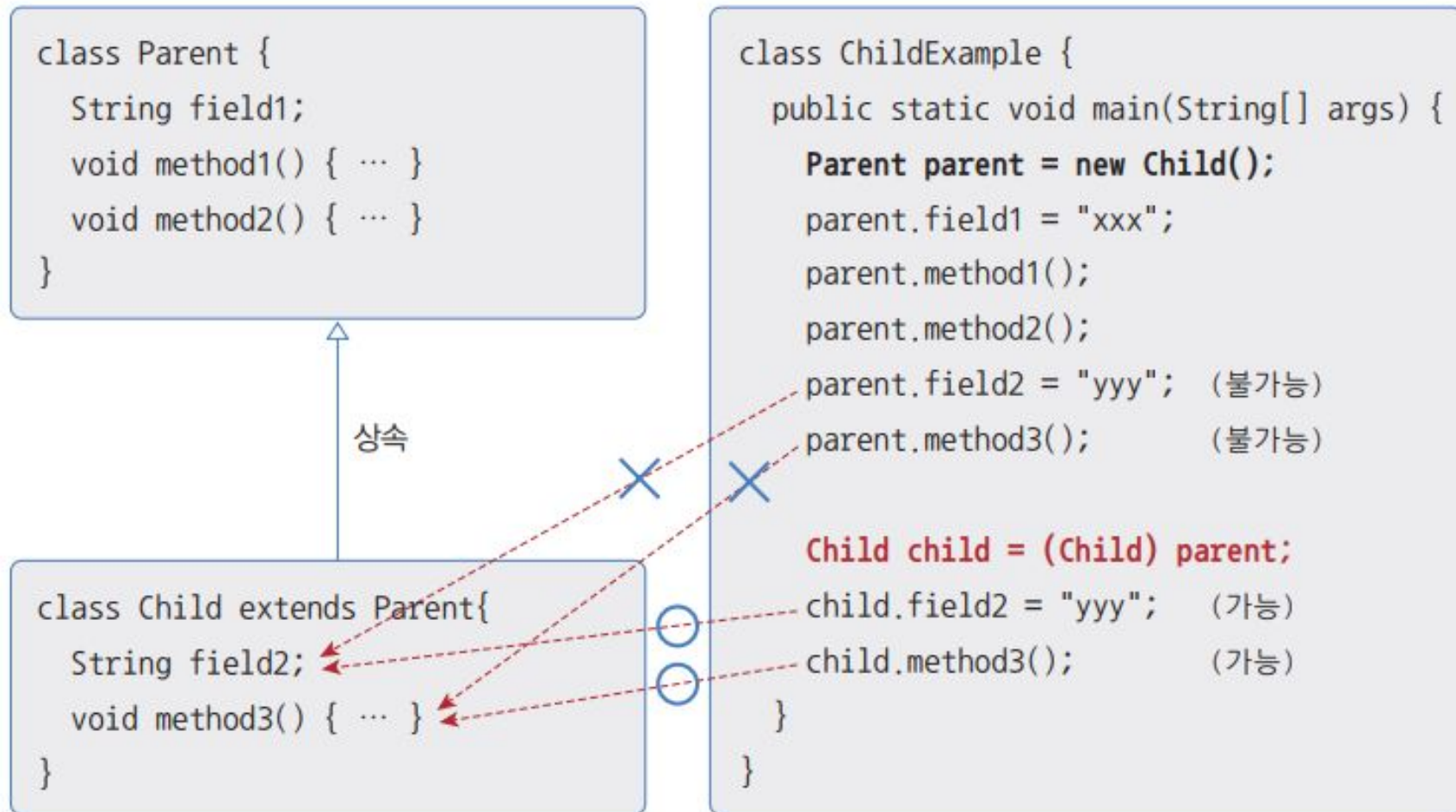


```
Parent parent = new Child();    //자동 타입 변환  
Child child = (Child) parent;  //강제 타입 변환
```

7 타입 변환

• 강제 타입 변환

- 자식 객체가 부모 타입으로 자동 변환하면 부모 타입에 선언된 필드와 메소드만 사용 가능



- **Parent.java**

```
package ch07.sec07.exam03;

public class Parent {
    //필드 선언
    public String field1;

    //메소드 선언
    public void method1() {
        System.out.println("Parent-method1()");
    }

    //메소드 선언
    public void method2() {
        System.out.println("Parent-method2()");
    }
}
```

- **Child.java**

```
package ch07.sec07.exam03;

public class Child extends Parent {
    //필드 선언
    public String field2;

    //메소드 선언
    public void method3() {
        System.out.println("Child-method3()");
    }
}
```

• Parent.java

```
package ch07.sec07.exam03;
```

```
public class ChildExample {
    public static void main(String[] args) {
        //객체 생성 및 자동 타입 변환
        Parent parent = new Child();

        //Parent 타입으로 필드와 메소드 사용
        parent.field1 = "data1";
        parent.method1();
        parent.method2();
        /*
        parent.field2 = "data2"; //(불가능)
        parent.method3();      //(불가능)
        */

        //강제 타입 변환
        Child child = (Child) parent;

        //Child 타입으로 필드와 메소드 사용
        child.field2 = "data2";    //(가능)
        child.method3();          //(가능)
    }
}
```

```
class Parent {
    String field1;
    void method1() { ... }
    void method2() { ... }
}
```

상속

```
class Child extends Parent{
    String field2;
    void method3() { ... }
}
```

```
class ChildExample {
    public static void main(String[] args) {
        Parent parent = new Child();
        parent.field1 = "xxx";
        parent.method1();
        parent.method2();
        parent.field2 = "yyy"; (불가능)
        parent.method3();      (불가능)

        Child child = (Child) parent;
        child.field2 = "yyy";    (가능)
        child.method3();        (가능)
    }
}
```

• 다형성

- 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질
- 다형성을 구현하기 위해서는 자동 타입 변환과 메소드 재정의가 필요



• 필드 다형성

- 필드 타입은 동일하지만, 대입되는 객체가 달라져서 실행 결과가 다양하게 나올 수 있는 것

```
public class Car {
    //필드 선언
    public Tire tire;
```

```
    //메소드 선언
    public void run() {
        tire.roll();
    }
}
```

tire 필드에 대입된 객체의 roll() 메소드 호출

```
//Car 객체 생성
```

```
Car myCar = new Car();
```

```
//HankookTire 장착
```

```
myCar.tire = new HankookTire();
```

```
//KumhoTire 장착
```

```
myCar.tire = new KumhoTire();
```



```
myCar.run();
```

대입된(장착된) 타이어의 roll() 메소드 호출

- **Tire.java**

```
package ch07.sec08.exam01;

public class Tire {
    //메소드 선언
    public void roll() {
        System.out.println("회전합니다.");
    }
}
```

- **HankookTire.java**

```
package ch07.sec08.exam01;

public class HankookTire extends Tire {
    //메소드 재정의(오버라이딩)
    @Override
    public void roll() {
        System.out.println("한국 타이어가 회전합니다.");
    }
}
```

- **KumhoTire.java**

```
package ch07.sec08.exam01;

public class KumhoTire extends Tire {
    //메소드 재정의(오버라이딩)
    @Override
    public void roll() {
        System.out.println("금호 타이어가 회전합니다.");
    }
}
```

- **Car.java**

```
package ch07.sec08.exam01;

public class Car {
    //필드 선언
    public Tire tire;

    //메소드 선언
    public void run() {
        //tire 필드에 대입된 객체의 roll() 메소드 호출
        tire.roll();
    }
}
```

- **CarExample.java**

```
package ch07.sec08.exam01;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //Tire 객체 장착
        myCar.tire = new Tire();
        myCar.run();

        //HankookTire 객체 장착
        myCar.tire = new HankookTire();
        myCar.run();

        //KumhoTire 객체 장착
        myCar.tire = new KumhoTire();
        myCar.run();
    }
}
```

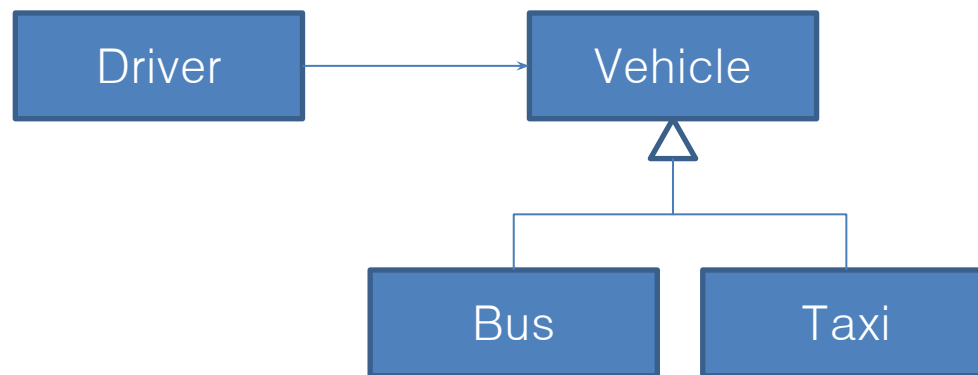
회전합니다.
한국 타이어가 회전합니다.
금호 타이어가 회전합니다.

• 매개변수 다형성

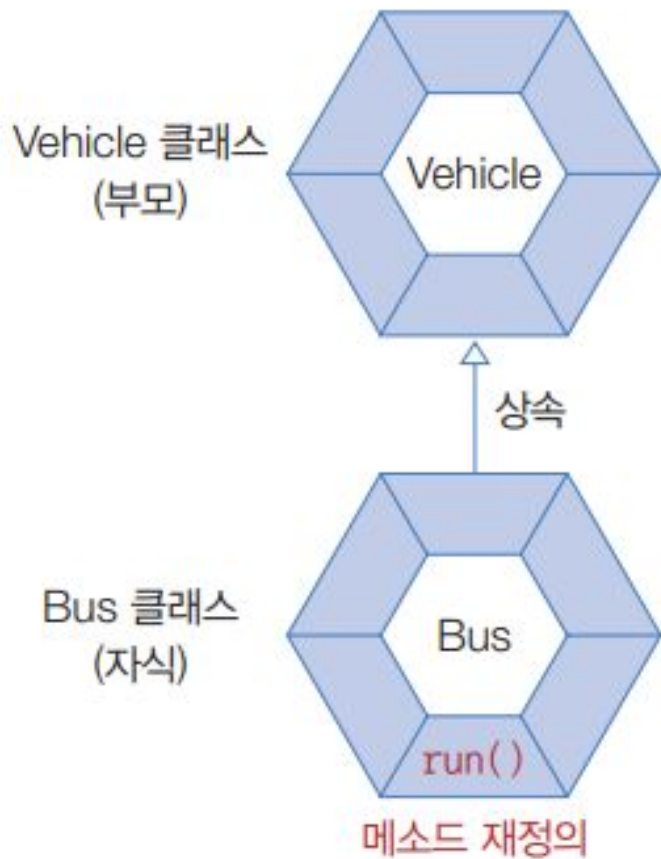
- 메소드가 클래스 타입의 매개변수를 가지고 있을 경우,
 - 호출할 때 동일한 타입의 자식 객체를 제공할 수 있음
- 어떤 자식 객체가 제공되느냐에 따라서 메소드의 실행 결과가 달라짐(전략 **strategy** 패턴)

```
public class Driver {  
    public void drive(Vehicle vehicle) {  
        vehicle.run();  
    }  
}
```

```
Driver driver = new Driver();  
Vehicle vehicle = new Vehicle();  
driver.drive(vehicle);
```



• 매개변수 다형성



```

Driver driver = new Driver();
Bus bus = new Bus();
driver.drive( bus ); •----- Bus 객체의 run() 호출
  
```

자동 타입 변환 발생

```
Vehicle vehicle = bus;
```

자식 객체

```

void drive(Vehicle vehicle) {
    vehicle.run(); •----- 자식 객체가 재정의한 run() 메소드 호출
}
  
```

- **Vehicle.java**

```
package ch07.sec08.exam02;

public class Vehicle {
    //메소드 선언
    public void run() {
        System.out.println("차량이 달립니다.");
    }
}
```

- **Bus.java**

```
package ch07.sec08.exam02;

public class Bus extends Vehicle {
    //메소드 재정의(오버라이딩)
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }
}
```

- **Taxi.java**

```
package ch07.sec08.exam02;

public class Taxi extends Vehicle {
    //메소드 재정의(오버라이딩)
    @Override
    public void run() {
        System.out.println("택시가 달립니다.");
    }
}
```


- **Driver.java**

```
package ch07.sec08.exam02;

public class Driver {
    //메소드 선언(클래스 타입의 매개변수를 가지고 있음)
    public void drive(Vehicle vehicle) {
        vehicle.run();
    }
}
```

- DriverExample

```
package ch07.sec08.exam02;

public class DriverExample {
    public static void main(String[] args) {
        //Driver 객체 생성
        Driver driver = new Driver();

        //매개값으로 Bus 객체를 제공하고 driver() 메소드 호출
        Bus bus = new Bus();
        driver.drive(bus);                // driver.drive(new Bus()); 와 동일


        //매개값으로 Taxi 객체를 제공하고 driver() 메소드 호출
        Taxi taxi = new Taxi();
        driver.drive(taxi);             // driver.drive(new Taxi()); 와 동일
    }
}
```

버스가 달립니다.
택시가 달립니다.

- instanceof 연산자

- 매개변수가 아니더라도 변수가 참조하는 객체의 타입을 확인할 때 instanceof 연산자를 사용
- instanceof 연산자에서 좌항의 객체가 우항의 타입이면 true를 산출하고 그렇지 않으면 false를 산출

```
boolean result = 객체 instanceof 타입;
```



```
public void method(Parent parent) {  
    if(parent instanceof Child) {  
        Child child = (Child) parent;  
        //child 변수 사용  
    }  
}
```

parent 매개변수가 참조하는 객체가 Child인지 조사

- instanceof 연산자

- Java 12부터는 instanceof 연산의 결과가 true일 경우 우측 타입 변수를 사용할 수 있기 때문에 강제 타입 변환이 필요 없음

```
if(parent instanceof Child child) {  
    //child 변수 사용  
}
```

- **Person.java**

```
package ch07.sec09;

public class Person {
    //필드 선언
    public String name;

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

    //메소드 선언
    public void walk() {
        System.out.println("걸습니다.");
    }
}
```

- **Student.java**

```
package ch07.sec09;

public class Student extends Person {
    //필드 선언
    public int studentNo;

    //생성자 선언
    public Student(String name, int studentNo) {
        super(name);
        this.studentNo = studentNo;
    }

    //메소드 선언
    public void study() {
        System.out.println("공부를 합니다.");
    }
}
```

● InstanceofExample.java

```
package ch07.sec09;

public class InstanceofExample {
    //main() 메소드에서 바로 호출하기 위해 정적 메소드 선언
    public static void personInfo(Person person) {
        System.out.println("name: " + person.name);
        person.walk();

        //person이 참조하는 객체가 Student 타입인지 확인
        /*if (person instanceof Student) {
            //Student 객체일 경우 강제 타입 변환
            Student student = (Student) person;
            //Student 객체만 가지고 있는 필드 및 메소드 사용
            System.out.println("studentNo: " + student.studentNo);
            student.study();
        */

        //person이 참조하는 객체가 Student 타입일 경우
        //student 변수에 대입(타입 변환 발생)
        if(person instanceof Student student) {
            System.out.println("studentNo: " + student.studentNo);
            student.study();
        }
    }
}
```

- InstanceofExample.java

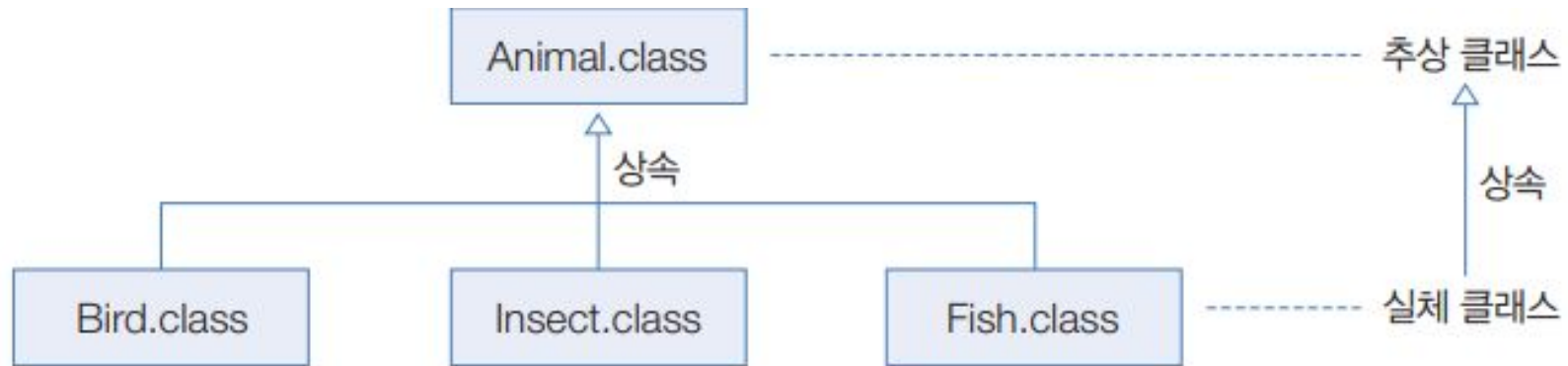
```
public static void main(String[] args) {  
    //Person 객체를 매개값으로 제공하고 personInfo() 메소드 호출  
    Person p1 = new Person("홍길동");  
    personInfo(p1);  
  
    System.out.println();  
  
    //Student 객체를 매개값으로 제공하고 personInfo() 메소드 호출  
    Person p2 = new Student("김길동", 10);  
    personInfo(p2);  
}
```

name: 홍길동
건습니다.

name: 김길동
건습니다.
studentNo: 10
공부를 합니다.

- 추상 클래스

- 객체를 생성할 수 있는 실제 클래스들의 공통적인 필드나 메소드를 추출해서 선언한 클래스
- 추상 클래스는 실제 클래스의 부모 역할. 공통적인 필드나 메소드를 물려받을 수 있음



```
Animal animal = new Animal(); //x
```

```
class Fish extends Animal {
    ...
}
```

- 추상 클래스 선언

- 클래스 선언에 **abstract** 키워드를 붙임
- **new** 연산자를 이용해서 객체를 직접 만들지 못하고 상속을 통해 자식 클래스만 만들 수 있다.

```
public abstract class 클래스명 {  
    //필드  
    //생성자  
    //메소드  
}
```

- Phone.java

```
package ch07.sec10.exam01;

public abstract class Phone {
    //필드 선언
    String owner;

    //생성자 선언
    Phone(String owner) {
        this.owner = owner;
    }

    //메소드 선언
    void turnOn() {
        System.out.println("폰 전원을 켭니다.");
    }

    void turnOff() {
        System.out.println("폰 전원을 끕니다.");
    }
}
```

- **SmartPhone.java**

```
package ch07.sec10.exam01;

public class SmartPhone extends Phone {
    //생성자 선언
    SmartPhone(String owner) {
        //Phone 생성자 호출
        super(owner);
    }

    //메소드 선언
    void internetSearch() {
        System.out.println("인터넷 검색을 합니다.");
    }
}
```

- **PhoneExample.java**

```
package ch07.sec10.exam01;

public class PhoneExample {
    public static void main(String[] args) {
        //Phone phone = new Phone();

        SmartPhone smartPhone = new SmartPhone("홍길동");

        smartPhone.turnOn();
        smartPhone.internetSearch();
        smartPhone.turnOff();
    }
}
```

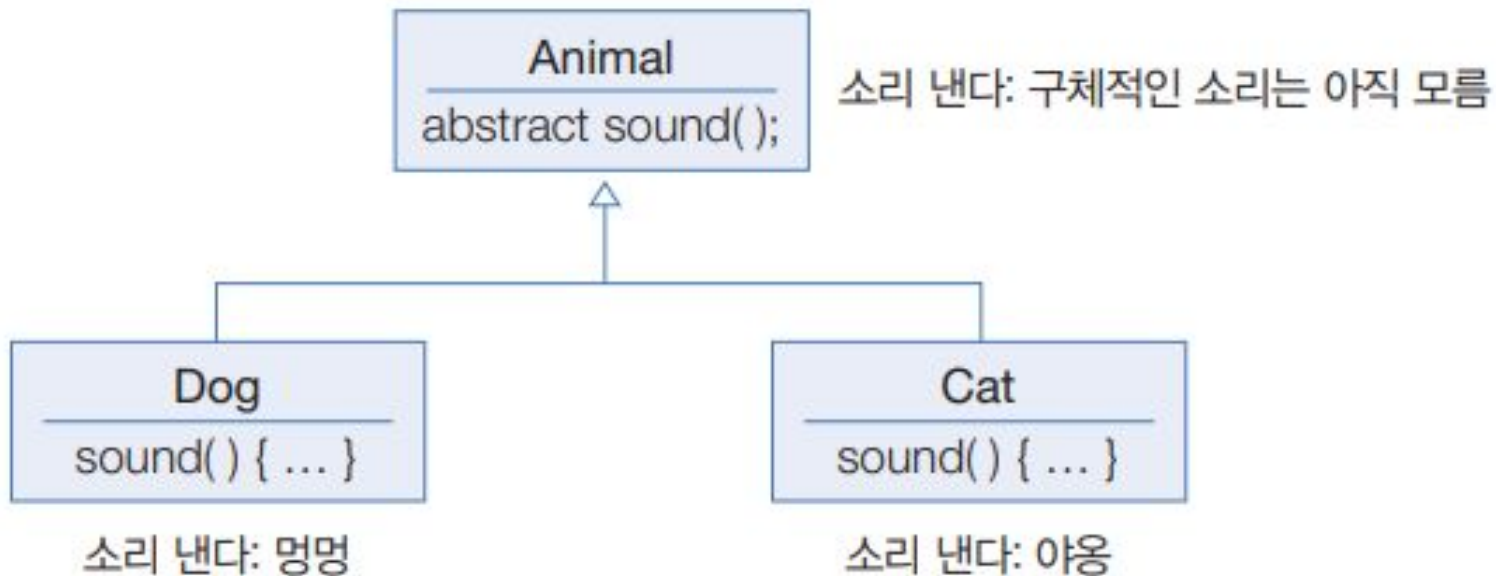
폰 전원을 켭니다.
인터넷 검색을 합니다.
폰 전원을 끕니다.

• 추상 메소드와 재정의

- 자식 클래스들이 가지고 있는 공통 메소드를 뽑아내어 추상 클래스로 작성할 때, 메소드 선언부만 동일하고 실행 내용은 자식 클래스마다 달라야 하는 경우 추상 메소드를 선언할 수 있음
- 일반 메소드 선언과의 차이점은 **abstract** 키워드가 붙고, 메소드 실행 내용인 중괄호 **{ }**가 없다.

abstract 리턴타입 메소드명(매개변수, ...);

```
public abstract class Animal {
    abstract void sound();
}
```



- **Animal.java**

```
package ch07.sec10.exam02;

public abstract class Animal {
    //메소드 선언
    public void breathe() {
        System.out.println("숨을 쉽니다.");
    }

    //추상 메소드 선언
    public abstract void sound();
}
```

- **Dog.java**

```
package ch07.sec10.exam02;

public class Dog extends Animal {
    //추상 메소드 재정의
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

- **Cat.java**

```
package ch07.sec10.exam02;

public class Cat extends Animal {
    //추상 메소드 재정의
    @Override
    public void sound() {
        System.out.println("야옹");
    }
}
```


- AbstractMethodExample.java

```
package ch07.sec10.exam02;
```

```
public class AbstractMethodExample {  
    public static void main(String[] args) {  
        Dog dog = new Dog();  
        dog.sound();
```

```
        Cat cat = new Cat();  
        cat.sound();
```

```
        //매개변수의 다형성
```

```
        animalSound(new Dog());  
        animalSound(new Cat());
```

```
    }
```

자동 타입
변환

```
    public static void animalSound( Animal animal ) {  
        animal.sound(); // 재정의된 메소드 호출
```

```
    }
```

```
}
```

멍멍
야옹
멍멍
야옹