KB금융그룹 | 국민의 평생 금융파트너

2025년 상반기 K-디지털 트레이닝

상속

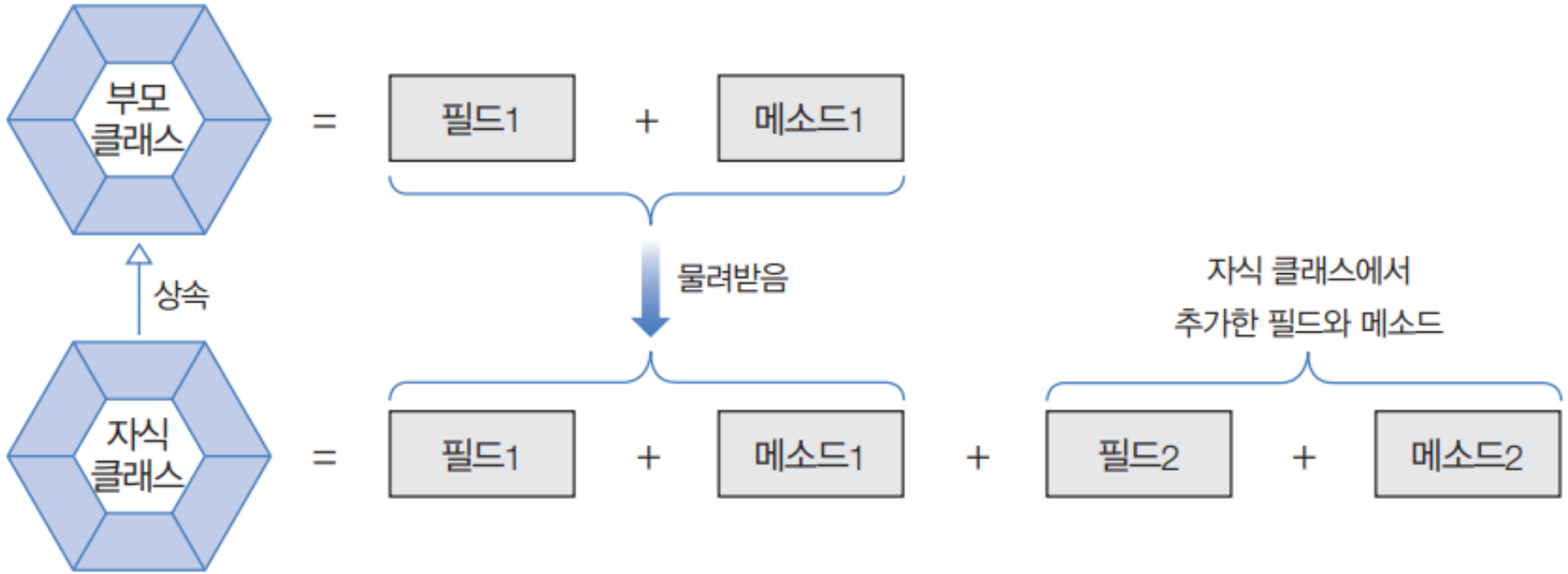
상속을 적절하게 사용하는 능력이 중요!

[KB] IT's Your Life

KB국민은행

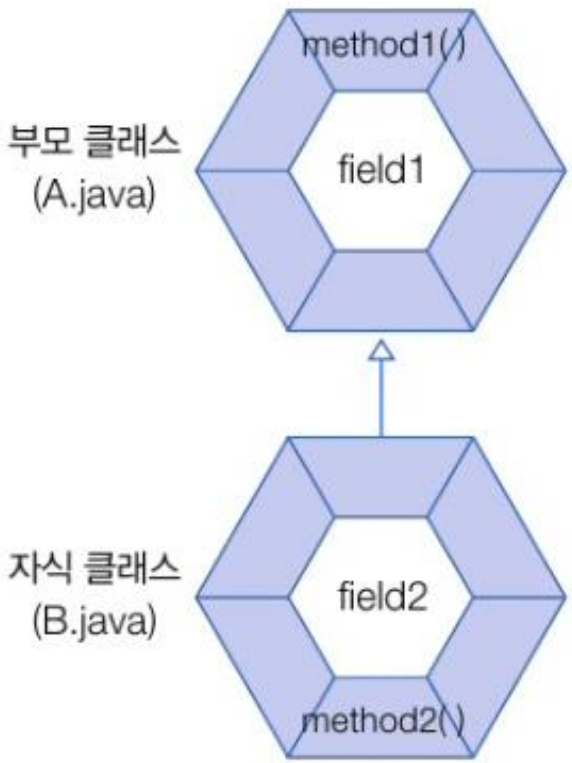
상속

- 부모 클래스의 필드와 메소드를 자식 클래스에게 물려줄 수 있음



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

C++ puython과 다르게

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

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.");
  }
}
```

생성자 없어

기본 생성자 자동으로 대입됨

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.");
  }
}
```

상속 받은 부모의 필드 초기화 중

좋지 않은 코드
자신의 필드는 자신이 초기화하는 게 좋아.
부모의 필드는 부모가 초기화

생성자 정의함.0
자식 생성자는 무조건 호출되고 첫 행동이 부모의 생성자를 호출함.
없으면 부모의 생성자 호출 코드를 자동으로 삽입함.

자동으로 생성된 부모의 생성자 호출 코드가 삽입될땐
기본적인 호출코드인 매개변수 안넘기는 생성자 호출 코드를 삽입함.

=>

부모에는 매개변수를 안받는 생성자가 꼭 있어야 한다.
자동으로 생성삽입된 디폴트 기본 생성자든
아니면 직접 정의한 매개변수를 안받는 생성자든

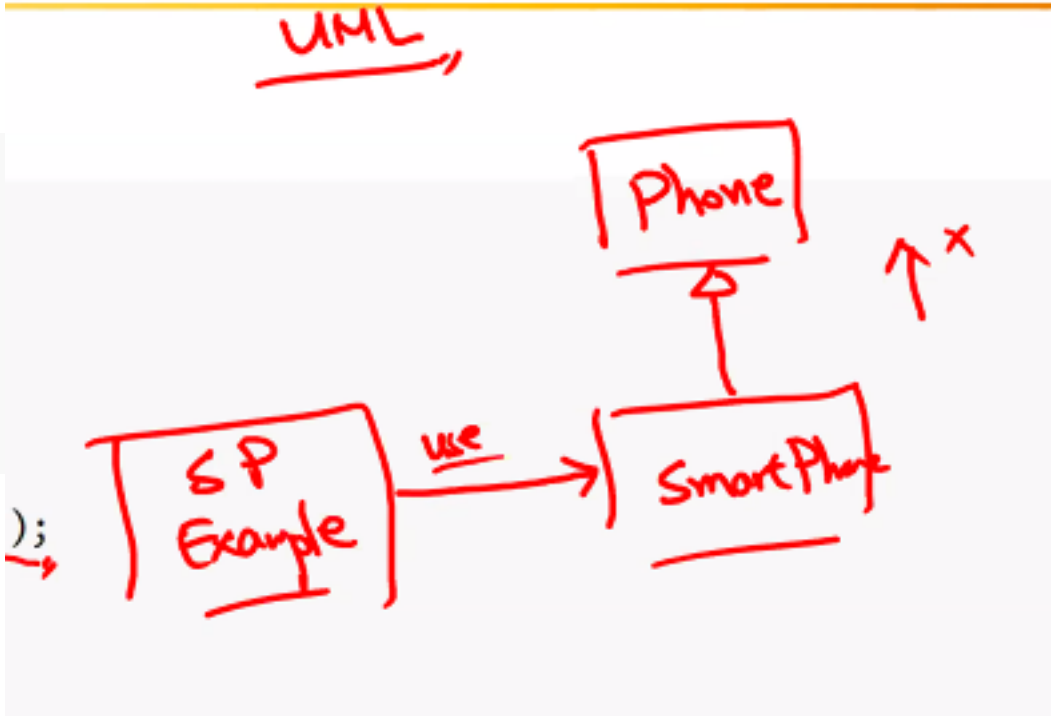
SmartPhoneExample.java

```
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();
    }
}
```

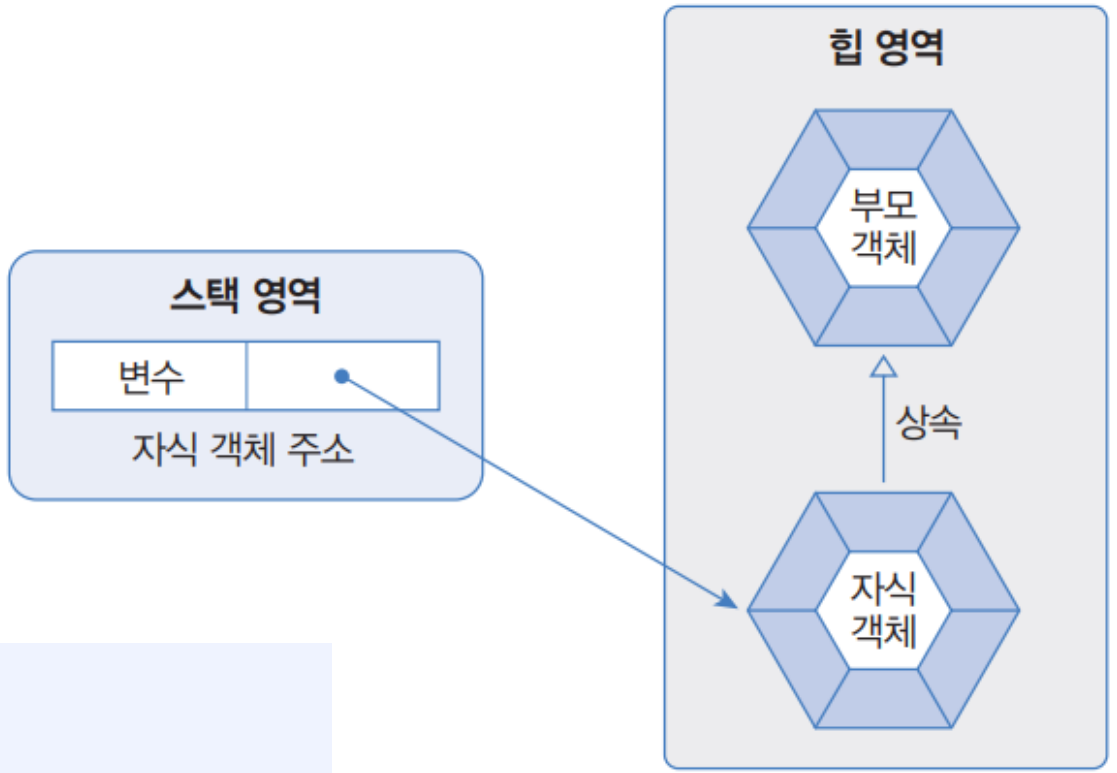
```
모델: 갤럭시
색상: 은색
와이파이 상태: false
벨이 울립니다.
자기: 여보세요.
상대방: 안녕하세요! 저는 홍길동인데요.
자기: 아~ 네, 반갑습니다.
전화를 끊습니다.
와이파이 상태를 변경했습니다.
인터넷에 연결합니다.
```



부모 생성자 호출

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

```
자식클래스 변수 = new 자식클래스( );
```



- 부모 생성자는 자식 생성자의 맨 첫 줄에 숨겨져 있는 super()에 의해 호출

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

앞서 말한 부모 생성자 호출을 명시한 코드

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

부모의 다른 생성자(매개변수 받는 버전의 생성자)를 호출하고 싶을 때는 해당 코드를 명시

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");
  }
}
```

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");
  }
}
```

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");
  }
}
```

만약 명시한 부모 생성자 호출을 안하면?

자식 생성자에서 문법 에러 발생함

맨 윗줄아니어도 에러

생성자( ){

super ( ) ;

this ( ) ;

}

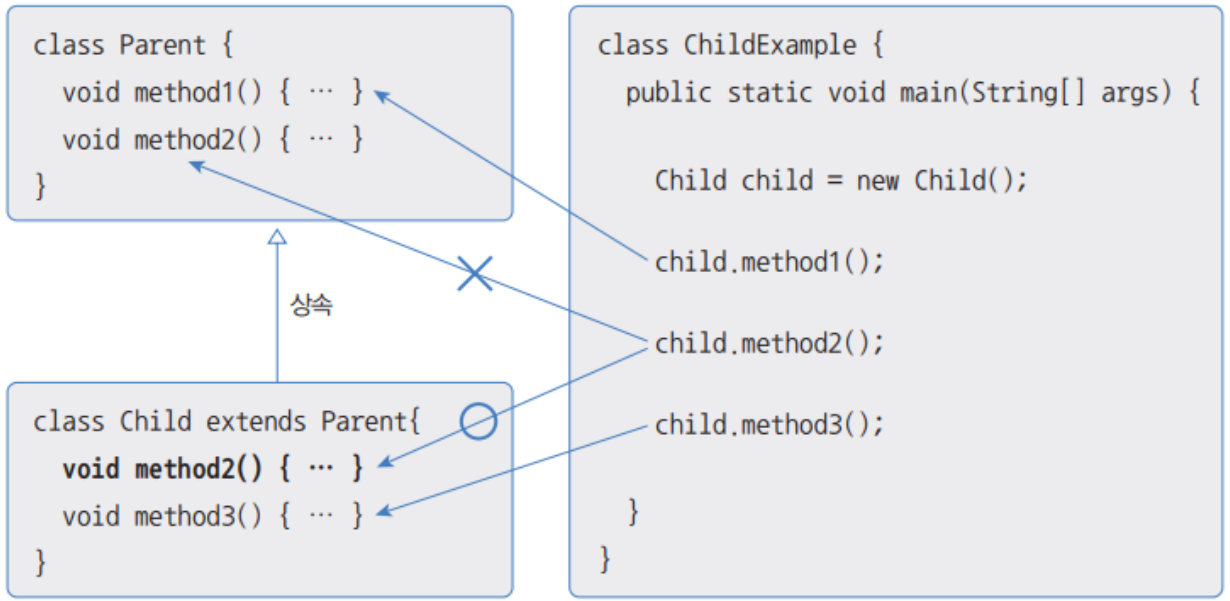
SmartPhoneExample.java

```
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
- 접근 제한을 더 강하게 오버라이딩할 수 없음(public → private으로 변경 불가)
- 새로운 예외를 throws할 수 없음

메소드를 오버라이딩 할때

접근제어자의 제한을 더 빡세게 하는 것은 허용되지 않는다.

그 반대는 가능하다

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;
  }
}
```

annotation

기능이 있는 주석.
이 경우에게는 컴파일러에게 내용을 알림.

그냥 주석은 코멘트 느낌. 설명 느낌.

인텔리제이 오버라이딩 지원 기능

ctrl+5

부모의 메서드를 조회할 수 있고
선택하면
오버라이딩 형식을 삽입해준다.

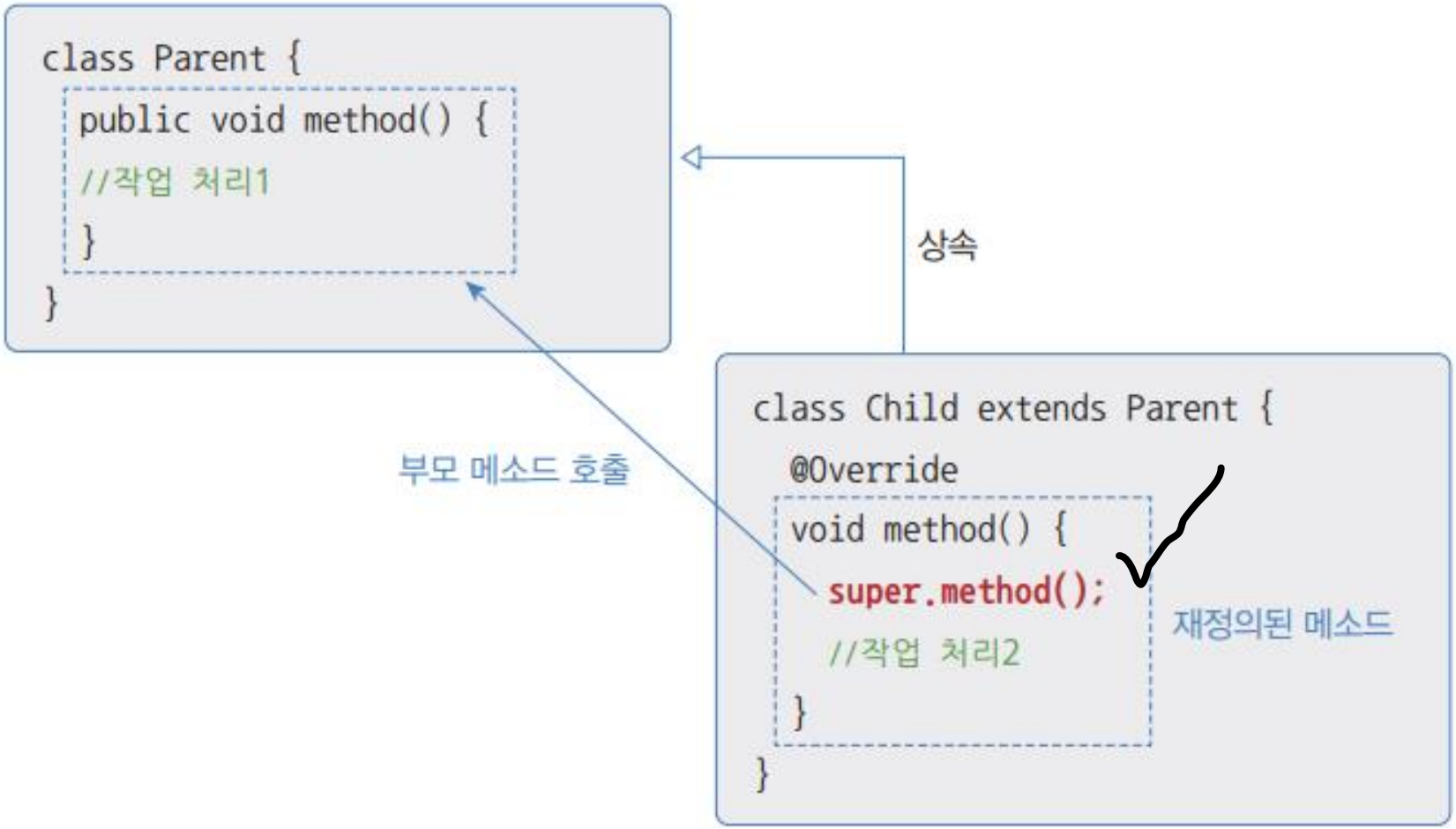
ComputerExample.java

```
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 메서드

final 클래스

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

```
public final class 클래스 { … }
```

final 메소드

- 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;
  }

}
```

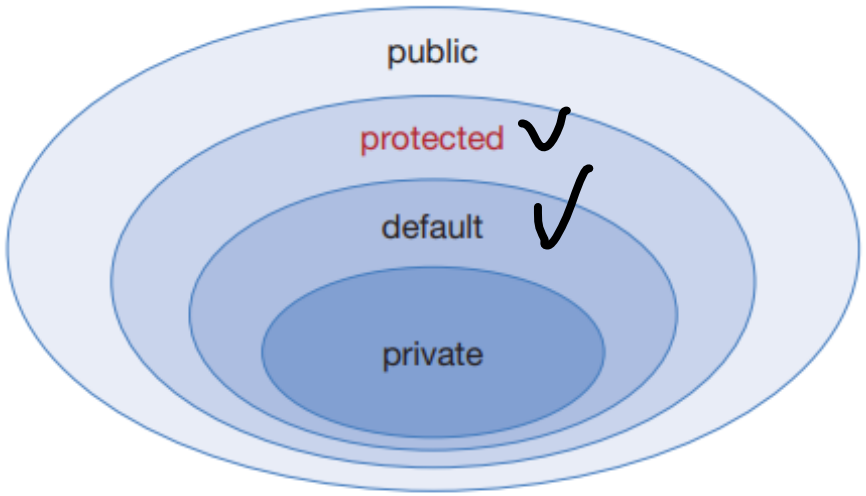
보통 부모 클래스는
회사에서 제공하거나 라이브러리에서 제공되는 클래스일 가능성이 크다.

만들어진 A클래스가 내가 필요한 기능과 필드를 가지고 있다면 상속하여 활용.

KB 국민은행

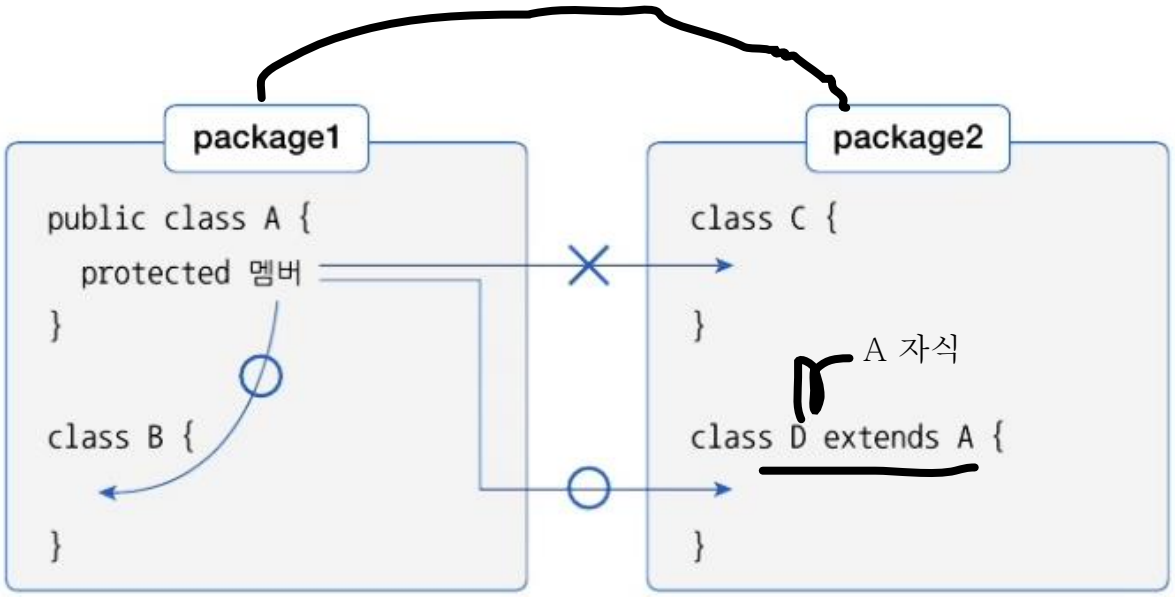
protected 접근 제한자

- protected는 상속과 관련이 있고, public과 default의 중간쯤에 해당하는 접근 제한
- protected는 같은 패키지에서는 default처럼 접근이 가능하나,
  다른 패키지에서는 자식 클래스만 접근을 허용



접근 제한이 강화

다른 패키지



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

6 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() {
  }
}
```

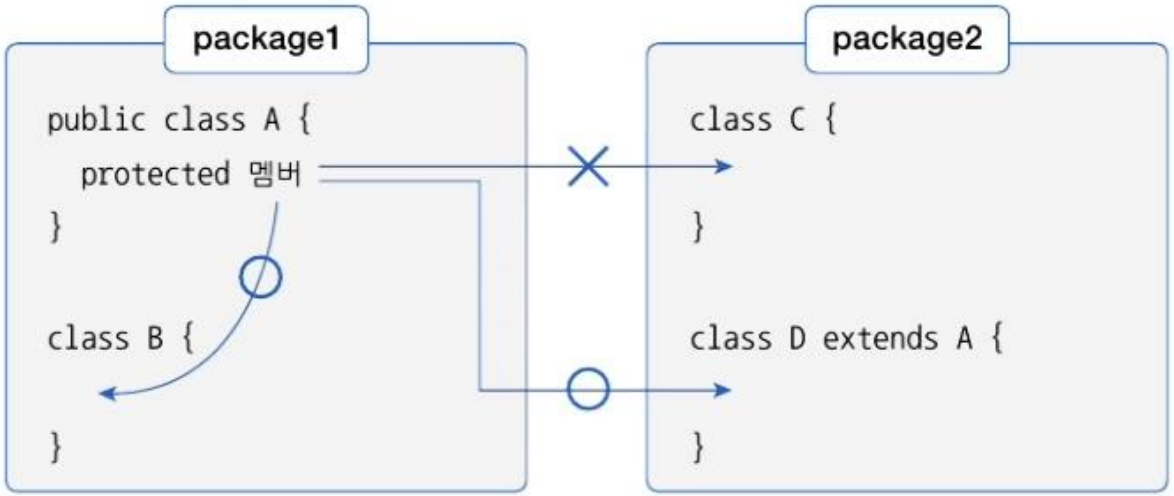
같은 패키지

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
  }
}
```

private 필드는 상속이 안되는 것이 아니다.
상속은 되지만 직접적인 접근 불가일뿐

햇갈리지 말어

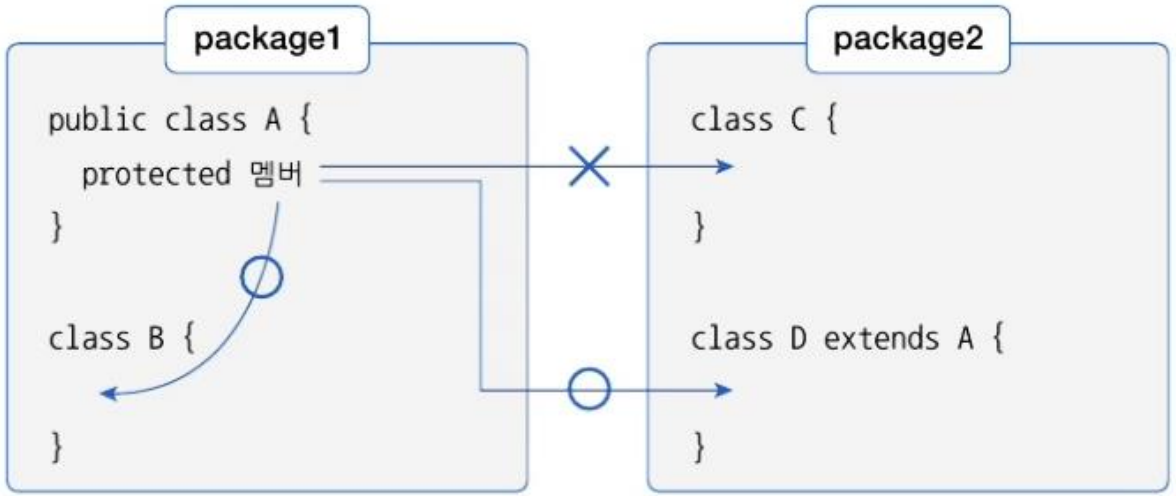


6 protected 접근 제한자

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
  }
```

다른 패키지 & 관계 없음



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
  }

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

다른 패키지 but 부모 자식 관계

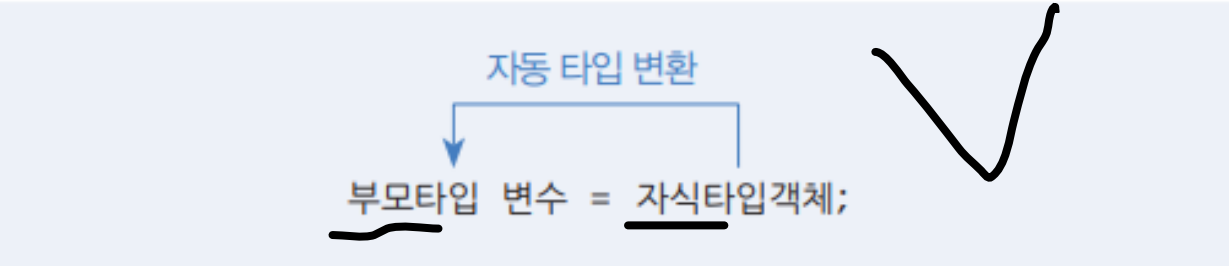
상속을 통해서만 가능

직접 객체 생성해서 사용하는 것은 안됨

참조형 타입 캐스팅.

자동 타입 변환

자동적으로 타입 변환이 일어나는 것



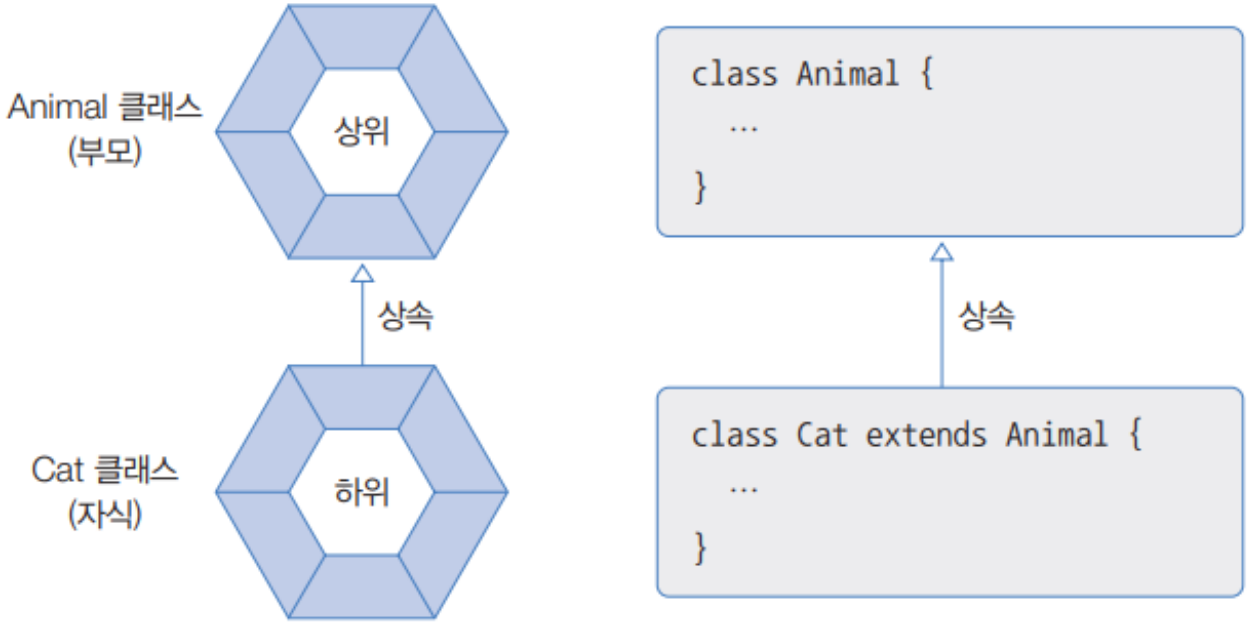
경우의 2개

1) 참조 부모 타입 = 자식 실체 객체

2) 참조 자식 타입 = 부모 실체 객체

1은 되고 2는 안되죠

자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급

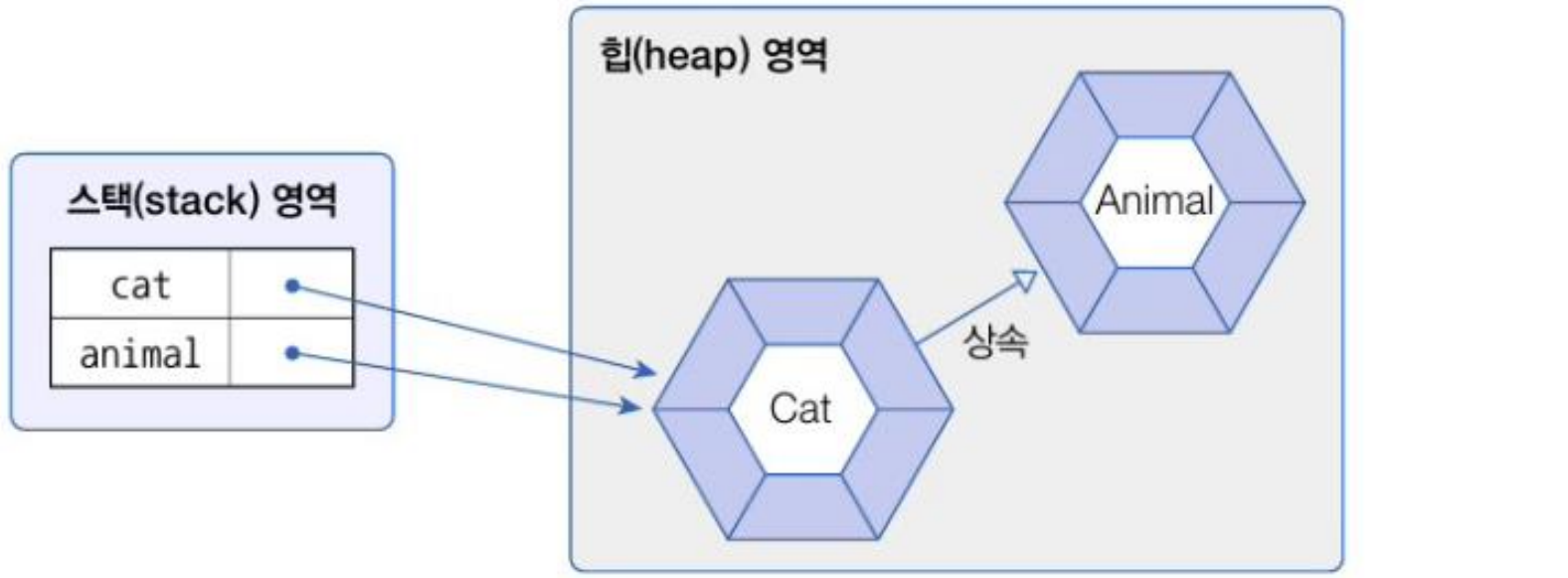


강제로 개발자가
명시적으로 캐스팅하여
2가 되게끔 할 수 있음
하지만 개발자 책임

자동 타입 변환

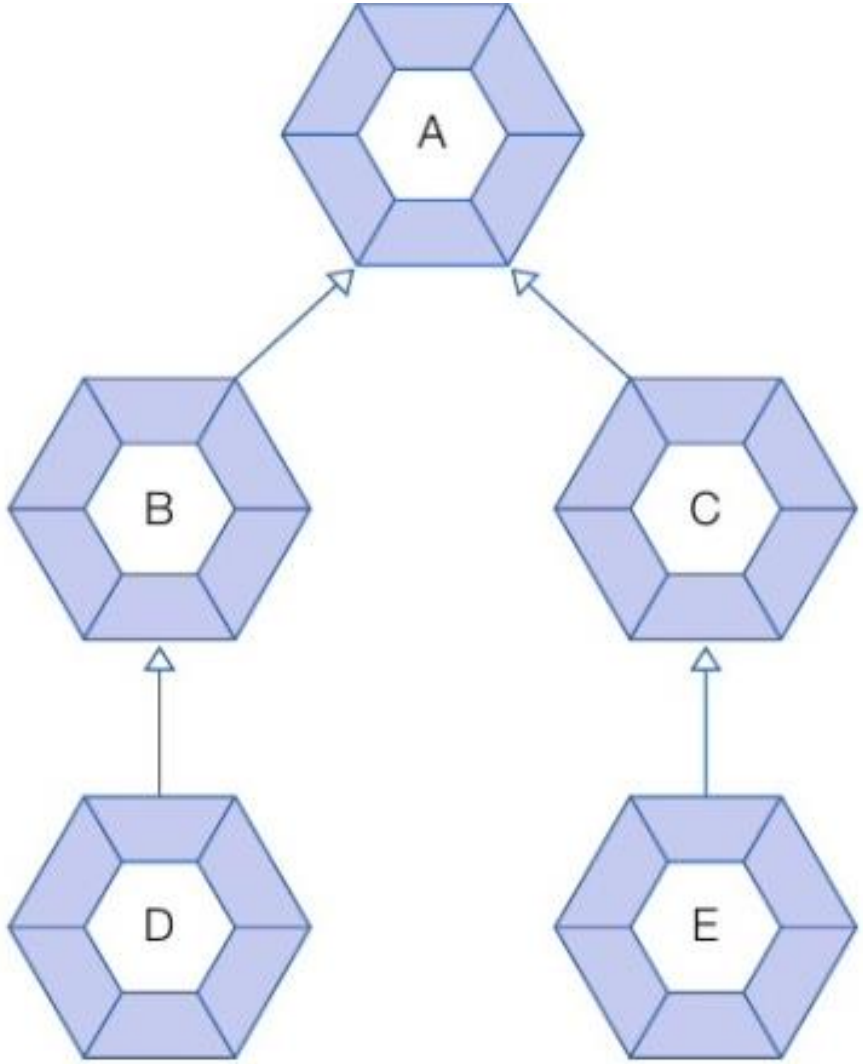
```
Cat cat = new Cat();
Animal animal = cat;
```

Animal animal = new Cat();도 가능




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

자동 타입 변환

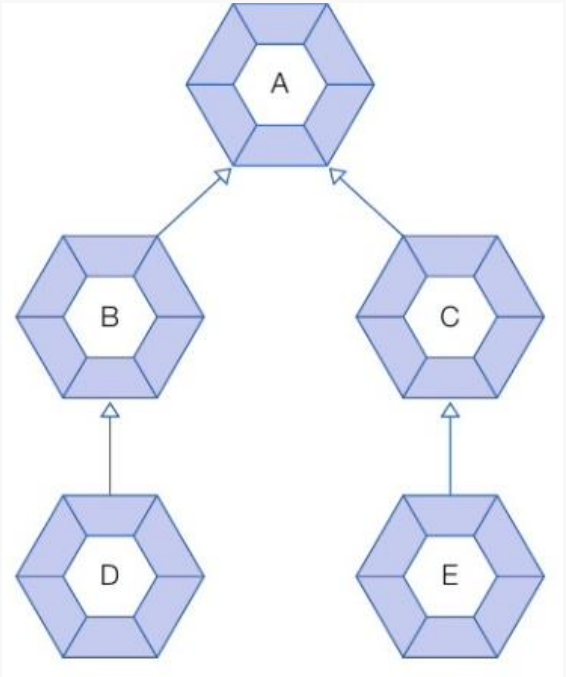



```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
```

```
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 {
}
```

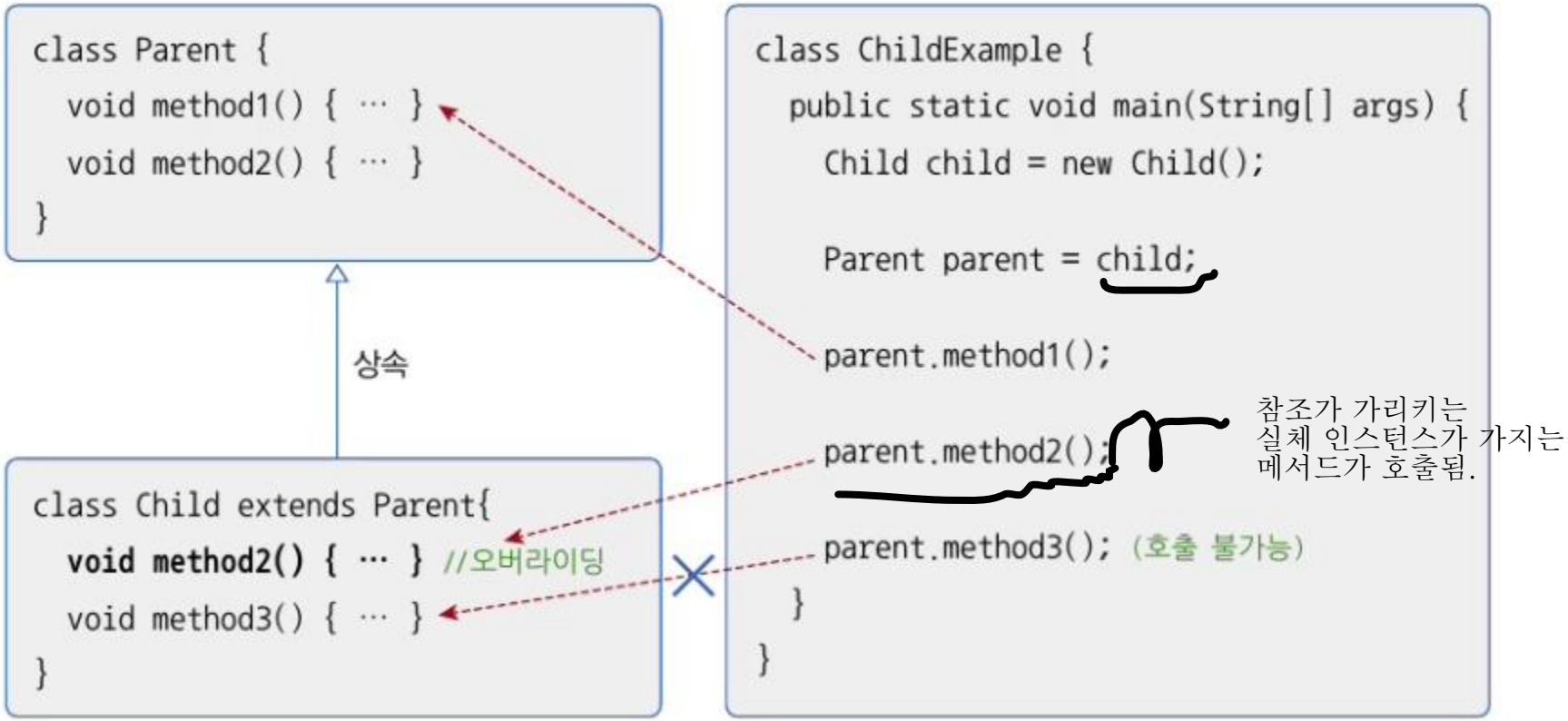



```
public class PromotionExample {
  public static void main(String[] args) {
    B b = new B();
    C c = new C();
    D d = new D();
    E e = new E();

    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)

자동 타입 변환



다양한 자식 타입들을 받는 함수는
매개변수 타입이 각각 다르게 여러개 오버로딩할 필요없이 부모타입으로 모두 받을 수 있다.

받은 인자로.
겉모습(참조형과 코드)은 같지만 실체가 달라 호출된 함수의 동작이 다르게 되는
다형성, 폴리모피즘이 가능해진다!!!!!!!!!!!!!!

Parent.java

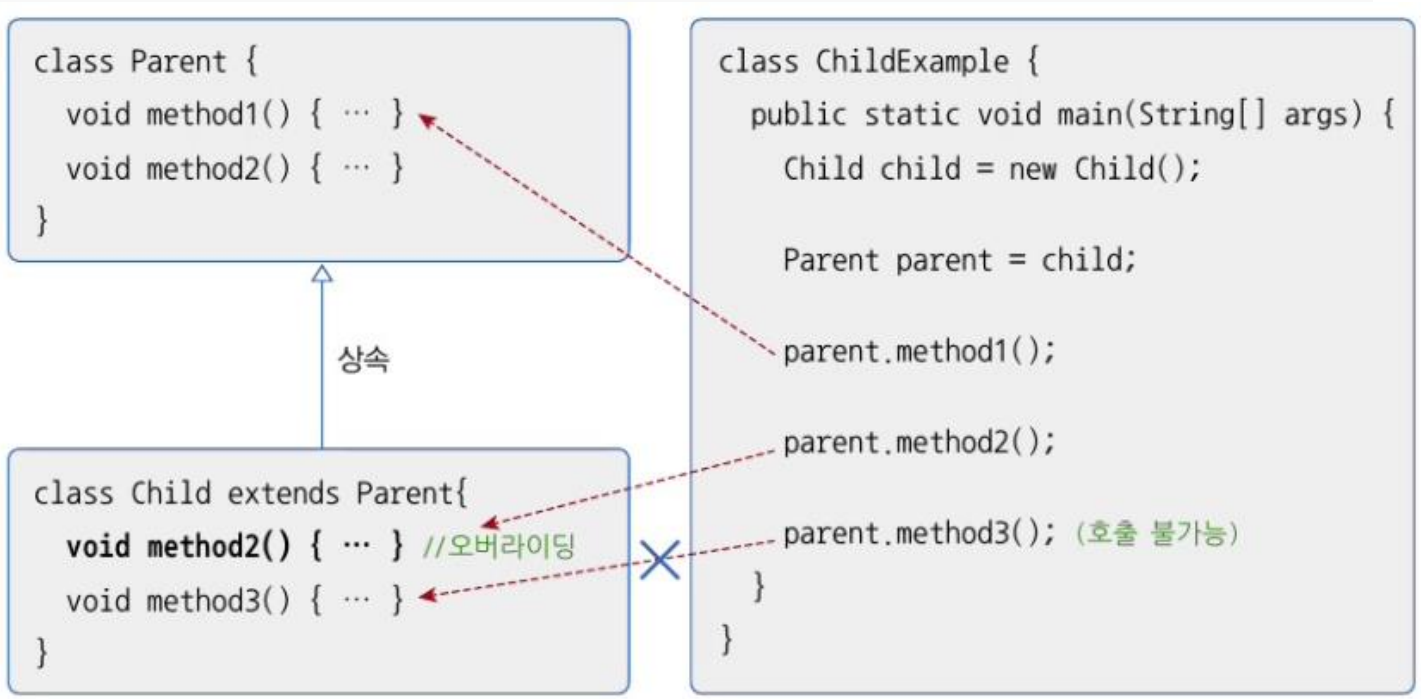
```
package ch07.sec07.exam02;

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()");
  }
}
```

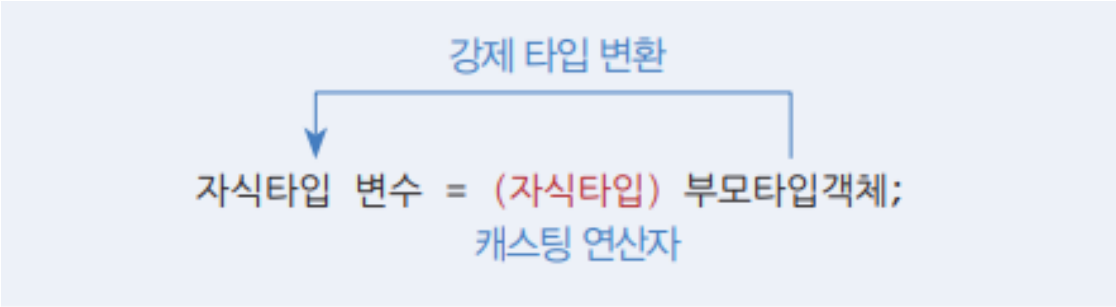
Parent.java

```
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)
  }
}
```



강제 타입 변환

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



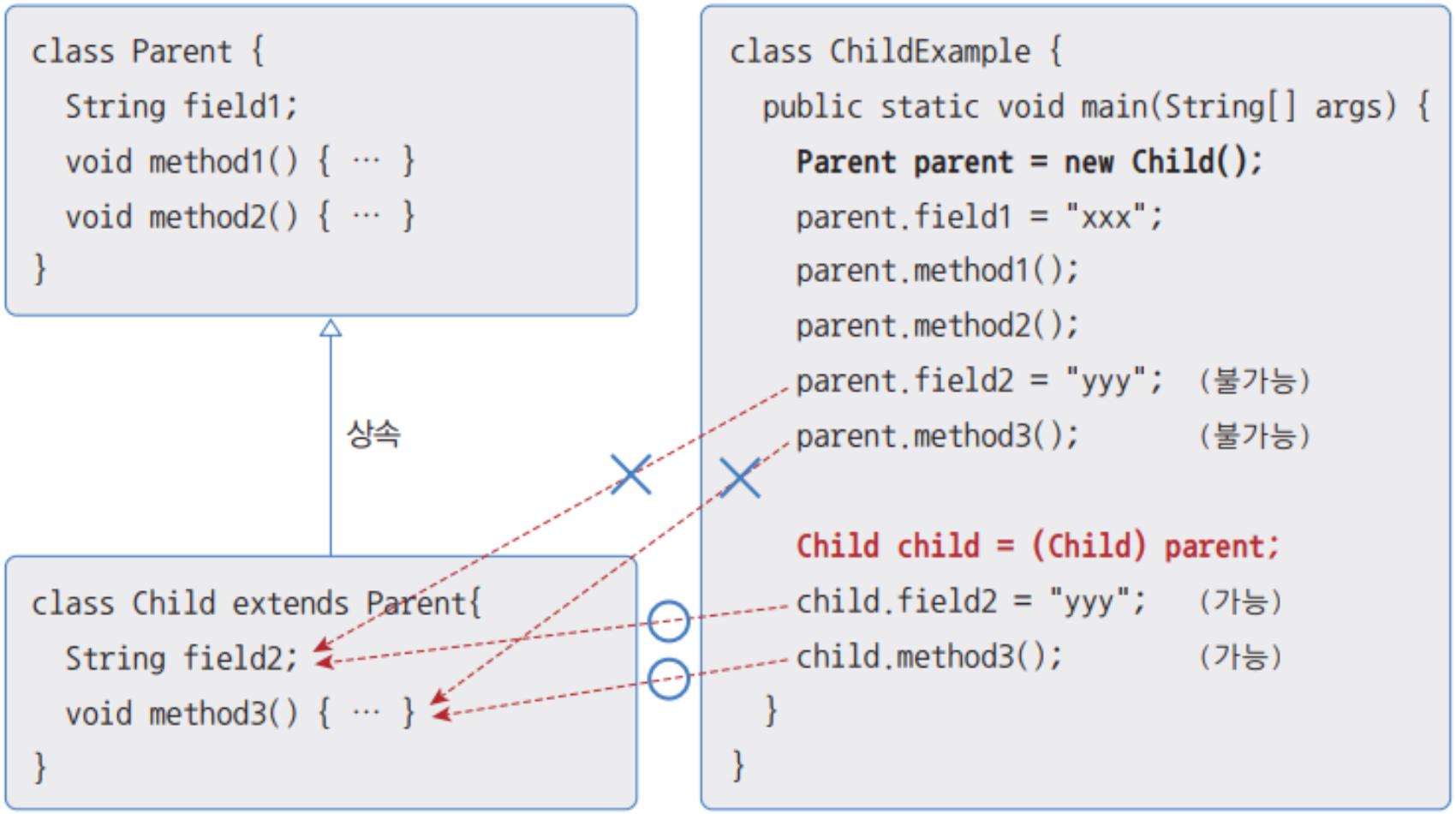

```
Parent parent = new Child();     //자동 타입 변환
Child child = (Child) parent;    //강제 타입 변환
```

본래 타입으로 다시 잠깐 써야 할 때.

이번 강제 타입 변환은
결과가
실체와 타입이 아다리가 맞죠?
이렇게 사용해야 합니다

강제 타입 변환

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



Parent.java

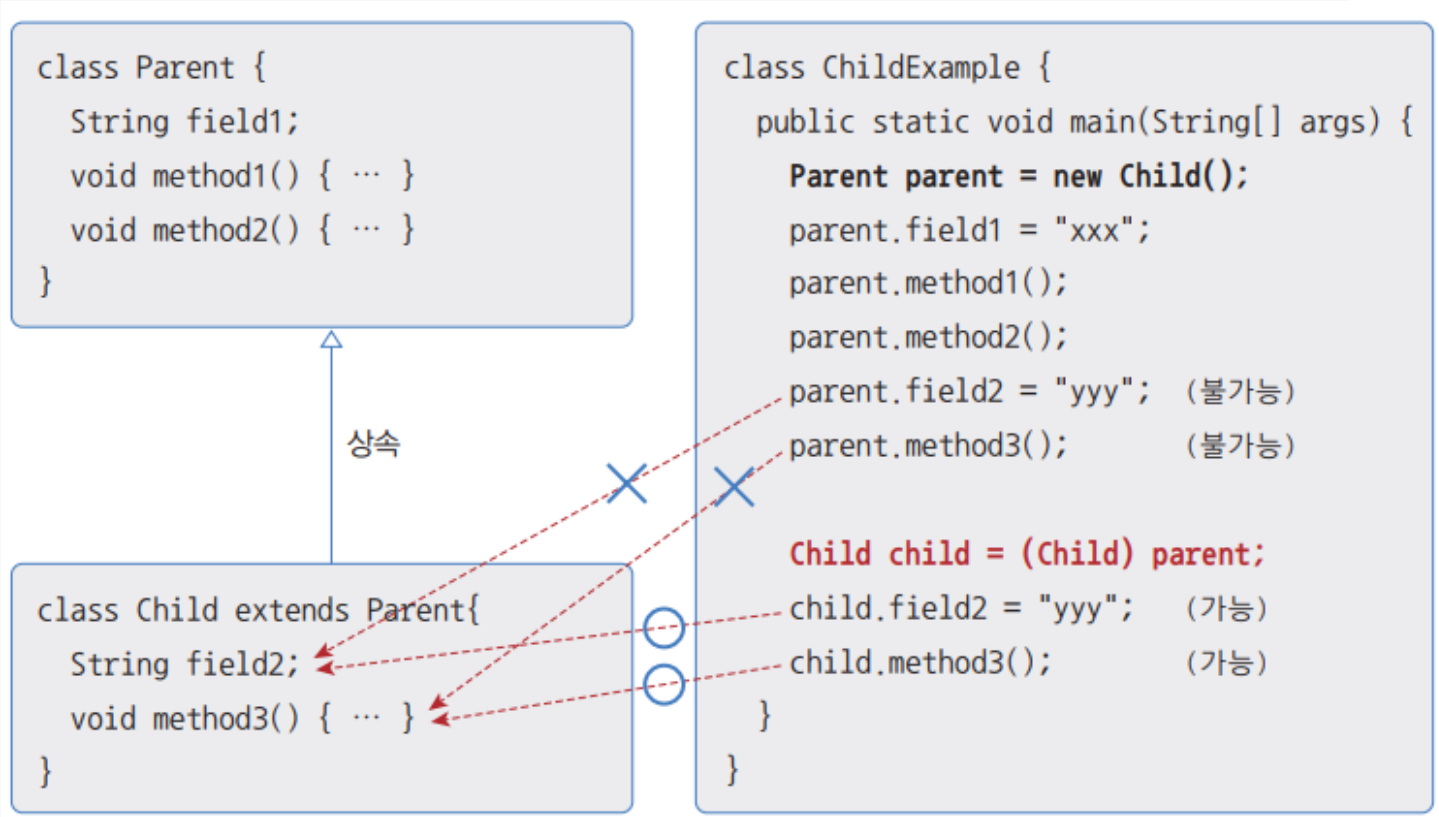
```
package ch07.sec07.exam03;

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)
  }
}
```



겉모습은 같지만 실제 행동은 다르게 작동하는 성질

코드와 참조형은 같지만 실체에 따라 행동은 다르게 하는 성질.

=>>>>코드 생산성, 수정용이성, 등등 증가. 여튼 좋음.

다형성

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

자동 타입 변환 + 메소드 오버라이딩 → 다형성

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();
  }
}
```

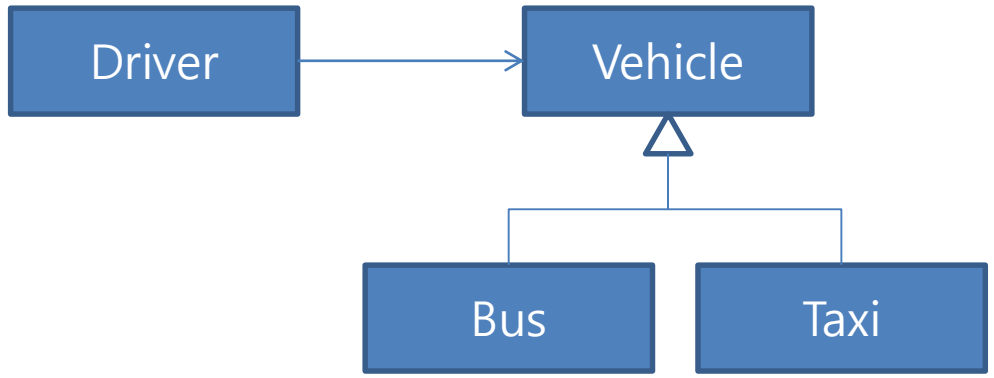
```
회전합니다.
한국 타이어가 회전합니다.
금호 타이어가 회전합니다.
```

매개변수 다형성

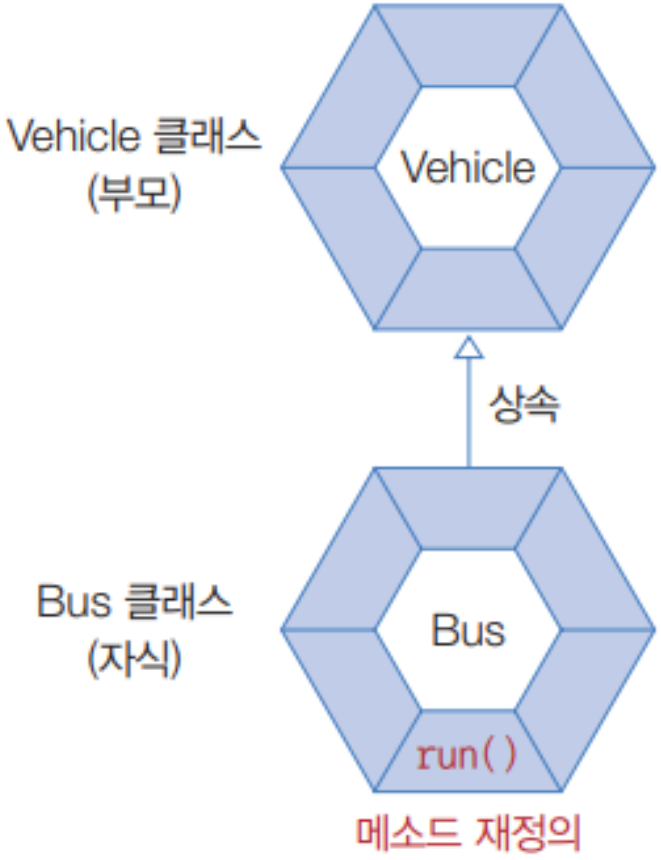
- 메소드가 클래스 타입의 매개변수를 가지고 있을 경우,
  - 호출할 때 동일한 타입의 자식 객체를 제공할 수 있음
- 어떤 자식 객체가 제공되느냐에 따라서 메소드의 실행 결과가 달라짐(전략 strategy 패턴)

```
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

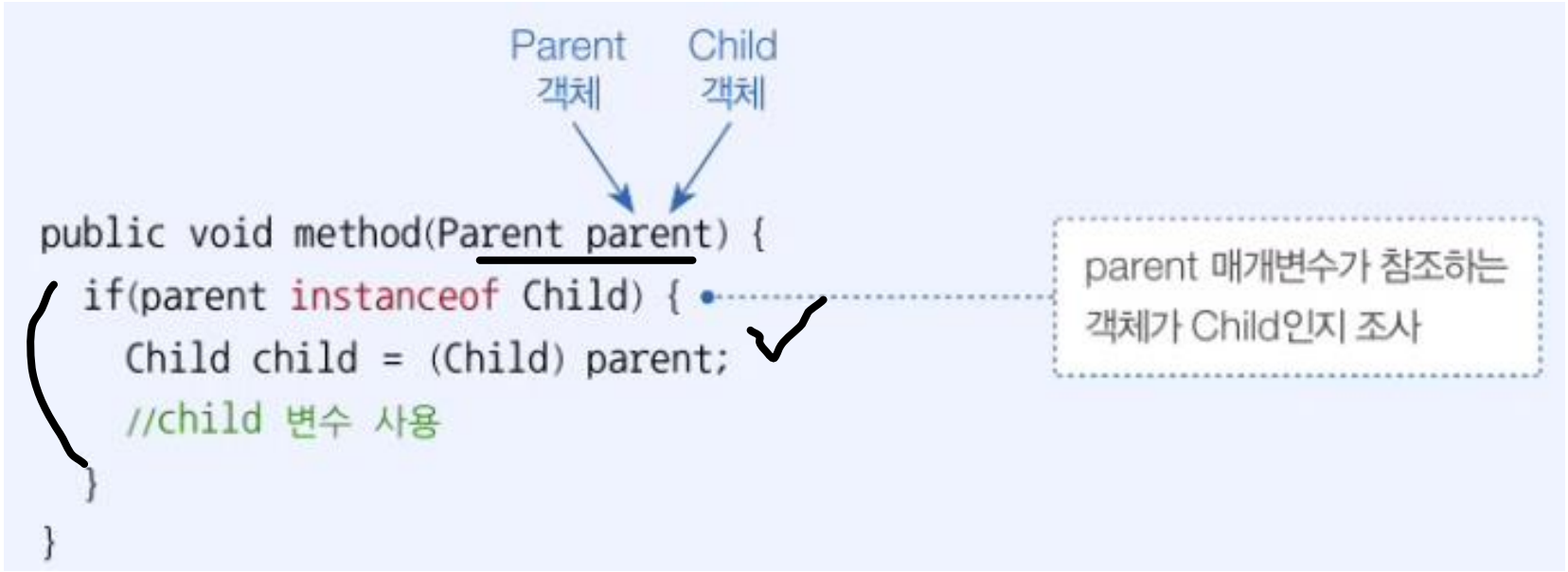
```
package ch07.sec08.exam02;

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;
```



instanceof 연산자

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

트루일때 자동변환후 대입됨 실체가

```
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("걷습니다.");
  }
}
```

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();
    }
  }
```

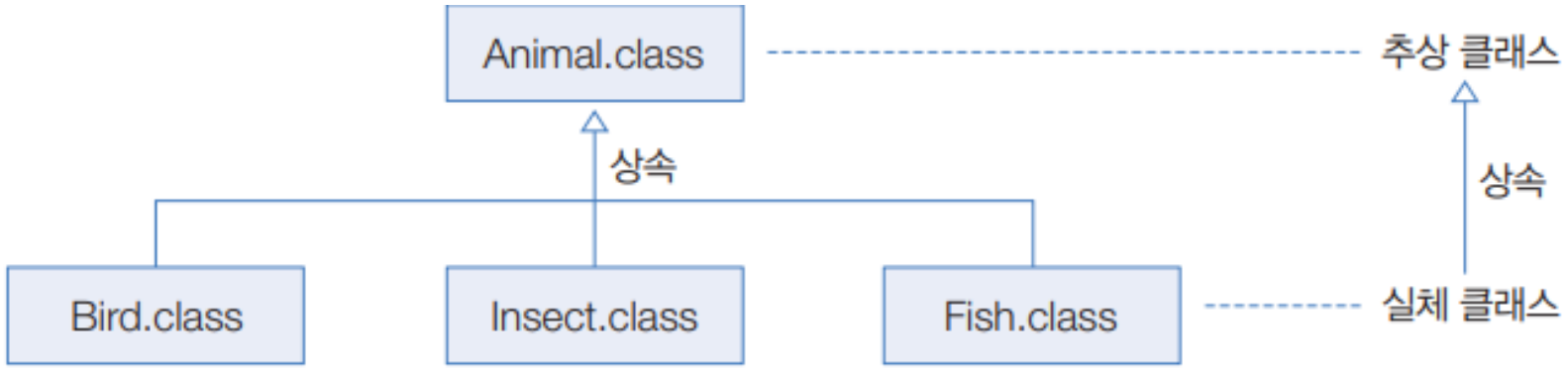
InstanceofExample.java

```
  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
걷습니다.

name: 김길동
걷습니다.
studentNo: 10
공부를 합니다.
```

추상 클래스

- 객체를 생성할 수 있는 실체 클래스들의 공통적인 필드나 메소드를 추출해서 선언한 클래스
- 추상 클래스는 실체 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();
  }
}
```

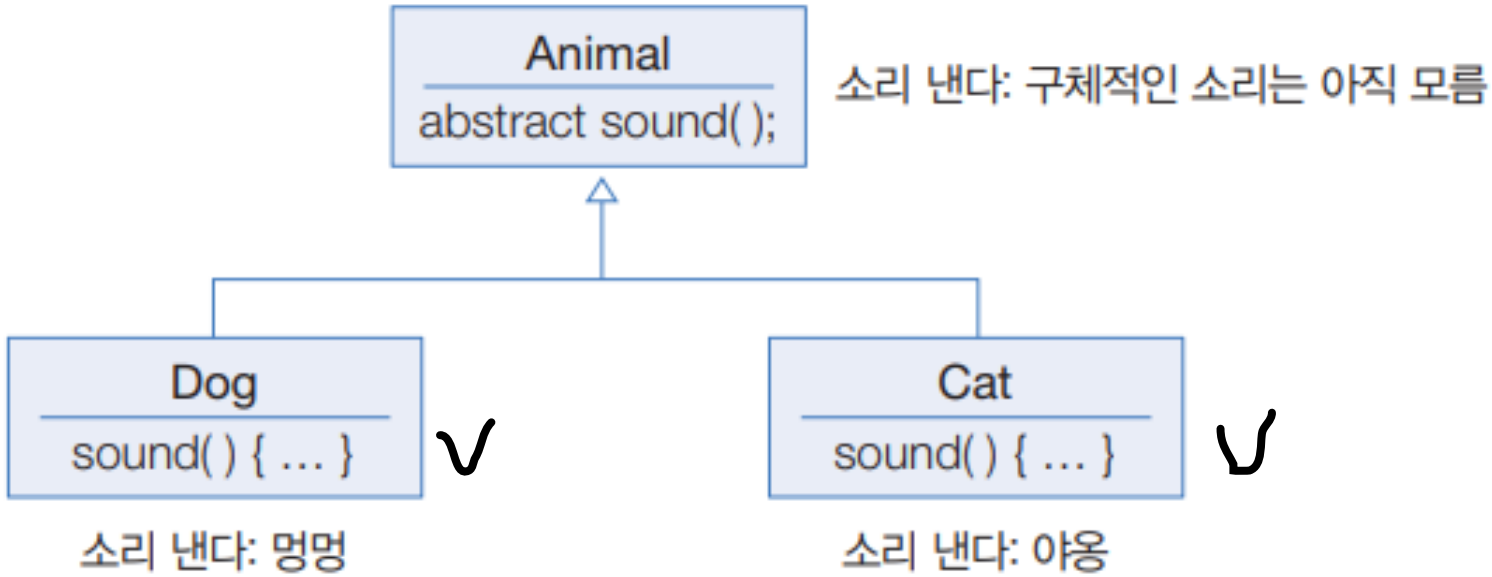
```
폰 전원을 켭니다.
인터넷 검색을 합니다.
폰 전원을 끕니다.
```

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");
  }
}
```

푸상클래스와 추상 메소드는 인텔리 제이에서 점선 그림으로 표현된다

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());
  }

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

자동 타입 변환

인텔리제이 기능 추상클래스
필수 구현

퓌해야하는 액션 클릭

메소드를 완성해주거나

해당 클래스도 추상으로 만들거나.

인텔리제이에서 문제가 생겼을 때
추가액션 을 워한다면

alt+enter해라.
제안해줌,

```
멍멍
야옹
멍멍
야옹
```

sealed 클래스

final과 유사하지만 약간 다름.
봉인을 일부 해제시켜주는 키워드

- Java 15부터 무분별한 자식 클래스 생성을 방지하기 위해 봉인된 클래스가 도입
- sealed 키워드를 사용하면 permits 키워드 뒤에 상속 가능한 자식 클래스를 지정
- final은 더 이상 상속할 수 없다는 뜻이고, non-sealed는 봉인을 해제한다는 뜻

```
public sealed class Person  permits Employee, Manager { … }
```

Employee manager클래스 빼고는 상속못하게끔.

```
public final class Employee extends Person { … }
public non-sealed class Manager extends Person { … }
```