

2024년 상반기 K-디지털 트레이닝

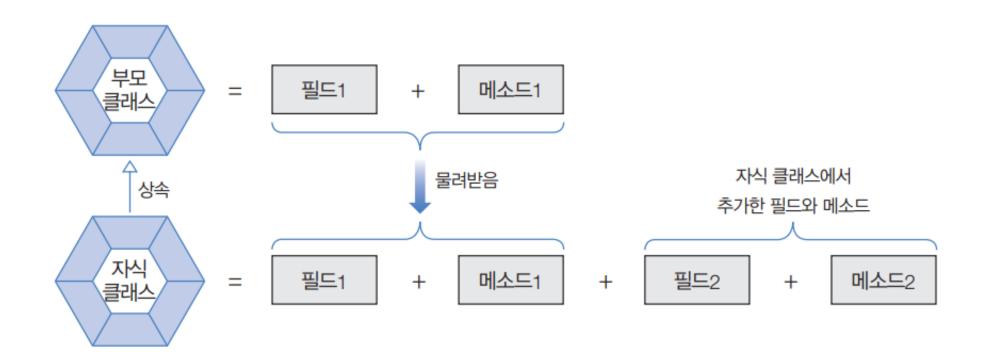
# 상속

[KB] IT's Your Life



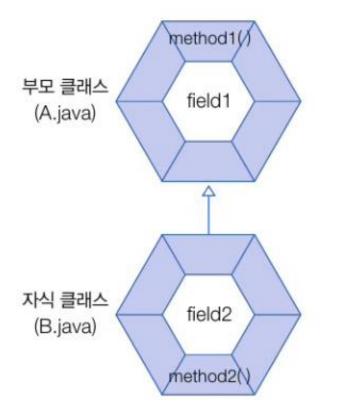
#### ☑ 상속

ㅇ 부모 클래스의 필드와 메소드를 자식 클래스에게 물려줄 수 있음



#### ♥ 상속의 이점

- o 이미 개발된 클래스를 재사용하므로 중복 코드를 줄임
- o 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

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

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

o 다중 상속 허용하지 않음. extends 뒤에 하나의 부모 클래스만 상속

```
public class 자식클래스 extends 부모클래스1, 부모글래스2 {
}
```

#### 클래스 상속

## 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();
```

# 2 클래스 상속

# 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.
```

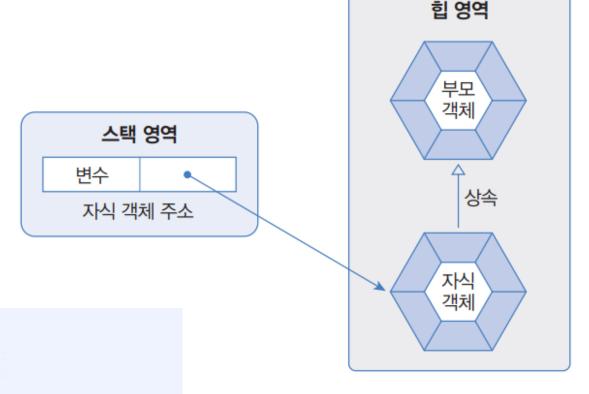
#### ♥ 부모 생성자 호출

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

```
자식클래스 변수 = new 자식클래스();
```

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

```
//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

```
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

```
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

```
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");
    }
}
```

# SmartPhoneExample.java

```
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

15

#### 메소드 오버라이딩

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

```
class Parent {
                                        class ChildExample {
 void method1() { ··· } ▼
                                          public static void main(String[] args) {
  void method2() { ··· }
                                            Child child = new Child();
                                            child.method1();
                  상속
                                            child.method2();
class Child extends Parent{
                                            child.method3();
  void method2() { ··· } *
  void method3() { ··· } ▲
```

- o 부모 메소드의 선언부(리턴 타입, 메소드 이름, 매개변수)와 동일해야 함
- o 접근 제한을 더 강하게 오버라이딩할 수 없음(public → private으로 변경 불가)
- o 새로운 예외를 throws할 수 없음

## 메소드 재정의

# **Calculator.**java

```
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**

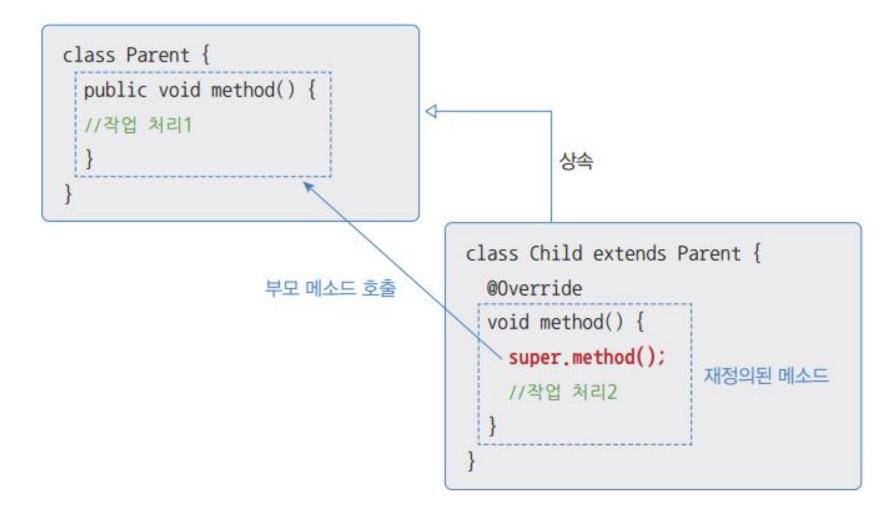
```
package ch07.sec04.exam01;
public class ComputerExample {
 public static void main(String[] args) {
   int r = 10;
   Calculator calculator = new Calculator();
   System.out.println("원 면적: " + calculator.areaCircle(r));
   System.out.println();
   Computer computer = new Computer();
   System.out.println("원 면적: " + computer.areaCircle(r));
```

```
Calculator 객체의 areaCircle() 실행
원 면적: 314.159
Computer 객체의 areaCircle() 실행
원 면적: 314.1592653589793
```

#### 4

#### ♥ 부모 메소드 호출

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



### 메소드 재정의

## 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

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

```
public final class 클래스 { ··· }
```

#### final 메소드

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

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

# 5 final 클래스, final 메서드

# 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
```

# 5 final 클래스, final 메서드

# **C**ar.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;
```

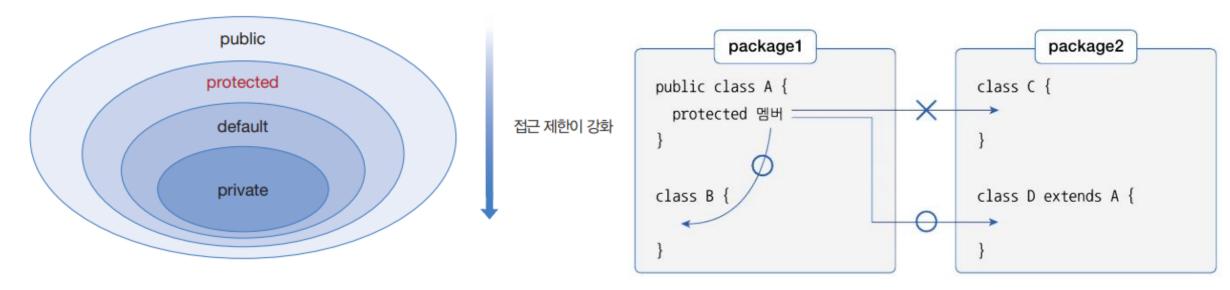
## 5 final 클래스, final 메서드

# 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

- o protected는 상속과 관련이 있고, public과 default의 중간쯤에 해당하는 접근 제한
- o protected는 같은 패키지에서는 default처럼 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

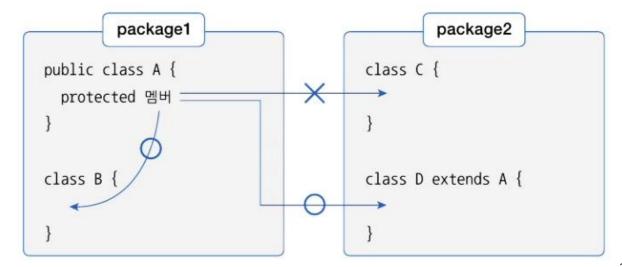
#### 6 protected 접근 제한자

# package1.A.java

```
package ch07.sec06.package1;
public class A {
 //필드 선언
 protected String field;
 //생성자 선언
 protected A() {
 //메소드 선언
 protected void method() {
```

### **Ø** package1.B.java

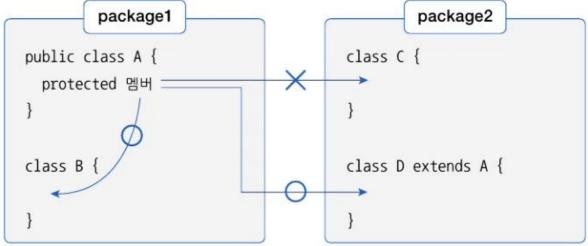
```
package ch07.sec06.package1;
public class B {
 //메소드 선언
 public void method() {
  A = new A(); //o
   a.field = "value"; //o
   a.method(); //o
```



#### 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
```

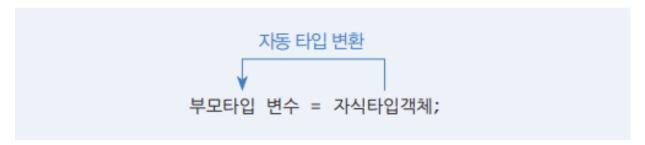


### package2.D.java

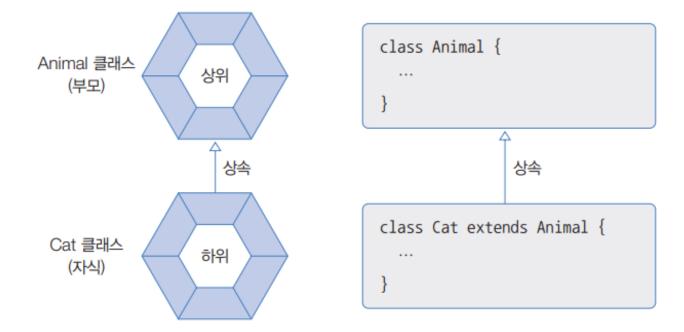
```
package ch07.sec06.package2;
import ch07.sec06.package1.A;
public class D extends A {
 public D() {
   //A() 생성자 호출
                     //o
   super();
 public void method1() {
                            상속을 통해서만 가능
   //A 필드값 변경
   this.field = "value"; //o
   //A 메소드 호출
   this.method();
                       //o
 //메소드 선언
 public void method2() {
   //A a = new A();
                       //x
                              직접 객체 생성해서
   //a.field = "value";
                       //x
                              사용하는 것은 안됨
   //a.method();
                       //x
```

#### 💟 자동 타입 변환

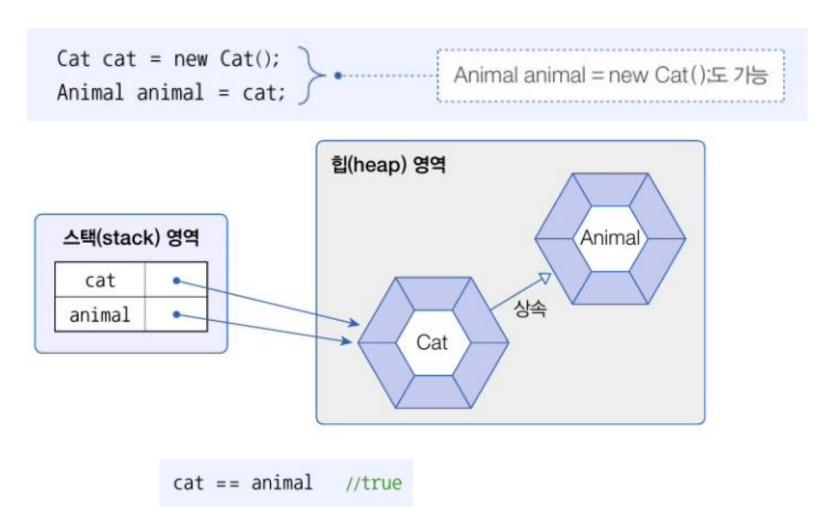
o 자동적으로 타입 변환이 일어나는 것



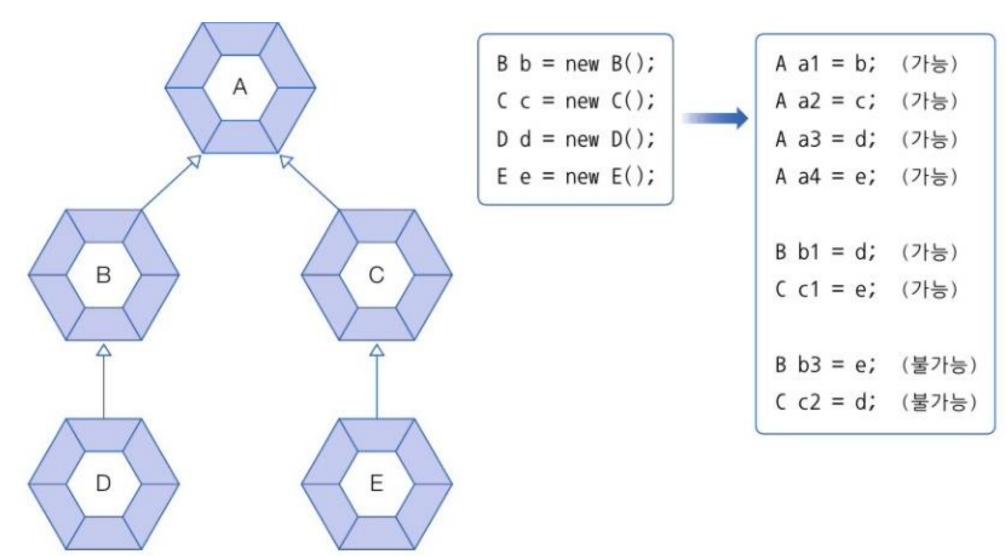
o 자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급



#### ♡ 자동 타입 변환

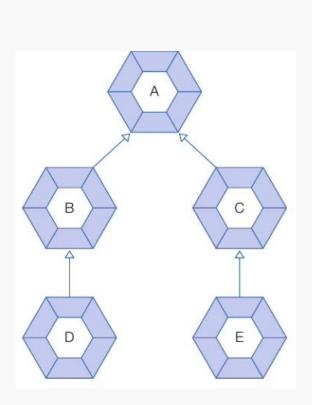


#### ♡ 자동 타입 변환



### **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;
               자동 타입 변환(상속 관계에 있음)
   B b1 = d;
   C c1 = e;
   // B b3 = e;
                컴파일 에러(상속 관계에 있지 않음)
   // C c2 = d;
```

#### 💟 자동 타입 변환

```
class Parent {
                                          class ChildExample {
 void method1() { ··· }
                                            public static void main(String[] args) {
 void method2() { ··· }
                                              Child child = new Child();
                                              Parent parent = child;
                                              parent.method1();
                  상속
                                              parent.method2();
class Child extends Parent{
                                              parent.method3(); (호출 불가능)
 void method2() { ··· } //오버라이딩
 void method3() { ··· } ◀
```

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

#### ☑ 강제 타입 변환

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

```
강제 타입 변환

▼
자식타입 변수 = (자식타입) 부모타입객체;

캐스팅 연산자
```

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

#### ◎ 강제 타입 변환

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

```
class Parent {
                                        class ChildExample {
  String field1;
                                          public static void main(String[] args) {
  void method1() { ··· }
                                            Parent parent = new Child();
 void method2() { ··· }
                                            parent.field1 = "xxx";
                                            parent.method1();
                                            parent.method2();
                                            parent.field2 = "yyy"; (불가능)
                  상속
                                            parent.method3();
                                                             (불가능)
                                            Child child = (Child) parent;
class Child extends Parent{
                                            child.field2 = "yyy"; (가능)
  String field2; 4
                                            child.method3();
                                                             (가능)
  void method3() { ···
```

# 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)
                        //(가능)
   child.method3();
```

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

#### 다형성

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



#### ☑ 필드 다형성

o 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

#### 다형성

# **☑** Tire.java

```
package ch07.sec08.exam01;

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

# HankookTire.java

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

### KumhoTire.java

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

# **C**ar.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**

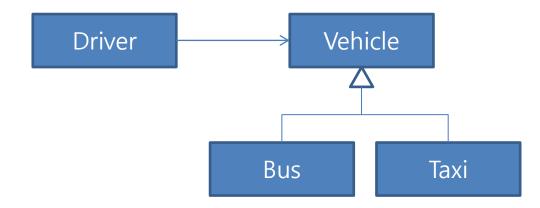
```
package ch07.sec08.exam01;
public class CarExample {
 public static void main(String[] args) {
   //Car 객체 생성
   Car myCar = new Car();
   //Tire 객체 장착
   myCar.tire = new Tire();
   myCar.run();
   //HankookTire 객체 장착
   myCar.tire = new HankookTire();
   myCar.run();
   //KumhoTire 객체 장착
                                        회전합니다.
   myCar.tire = new KumhoTire();
                                        한국 타이어가 회전합니다.
   myCar.run();
                                         금호 타이어가 회전합니다.
```

#### 매개변수 다형성

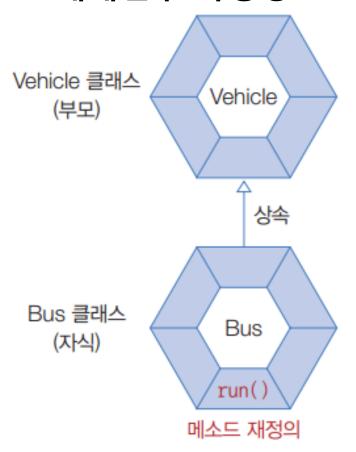
- o 메소드가 클래스 타입의 매개변수를 가지고 있을 경우,
  - 호출할 때 동일한 타입의 자식 객체를 제공할 수 있음
- o 어떤 자식 객체가 제공되느냐에 따라서 메소드의 실행 결과가 달라짐(전략 strategy 패턴)

```
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
```

#### 다형성

# **☑** Vehicle.java

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

## Bus.java

```
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

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

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

#### ☑ instanceof 연산자

o 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("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);

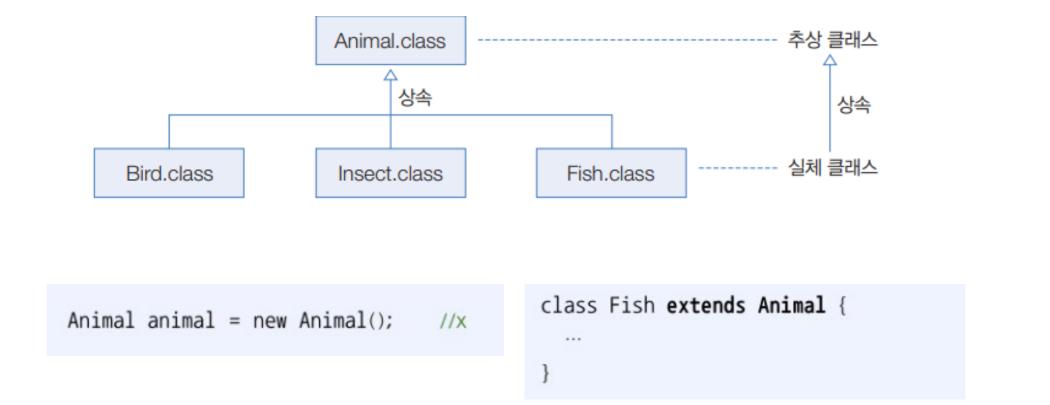
    System.out.println();

    //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

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



#### ♥ 추상 클래스 선언

- o 클래스 선언에 abstract 키워드를 붙임
- o 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

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

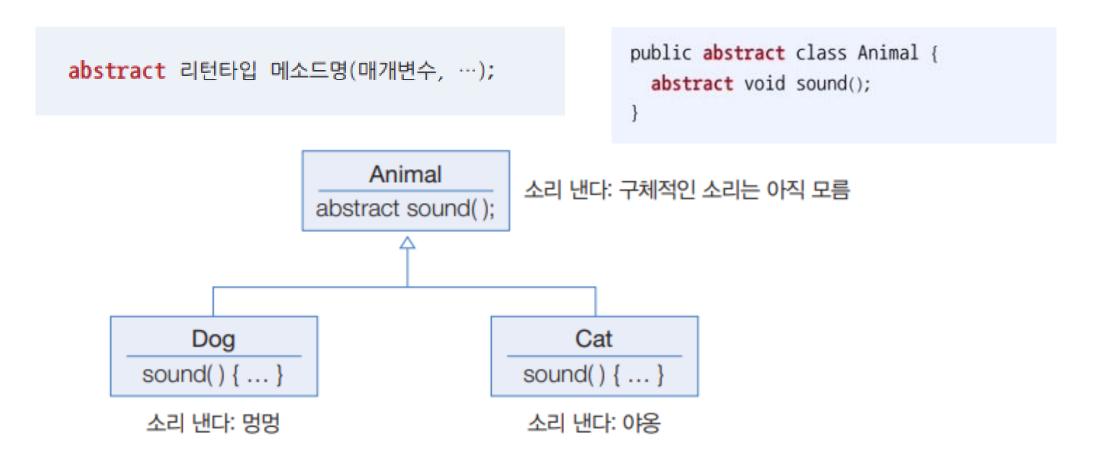
    SmartPhone smartPhone = new SmartPhone("홍길동");

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

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

#### 추상 메소드와 재정의

- o 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
- o 일반 메소드 선언과의 차이점은 abstract 키워드가 붙고, 메소드 실행 내용인 중괄호 { }가 없다.



## Animal.java

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

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

# **Dog.**java

```
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
```

#### 11 봉인된 클래스

#### sealed 클래스

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

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

public final class Employee extends Person { ... }

public non-sealed class Manager extends Person { ... }
```