자바 프로그래밍

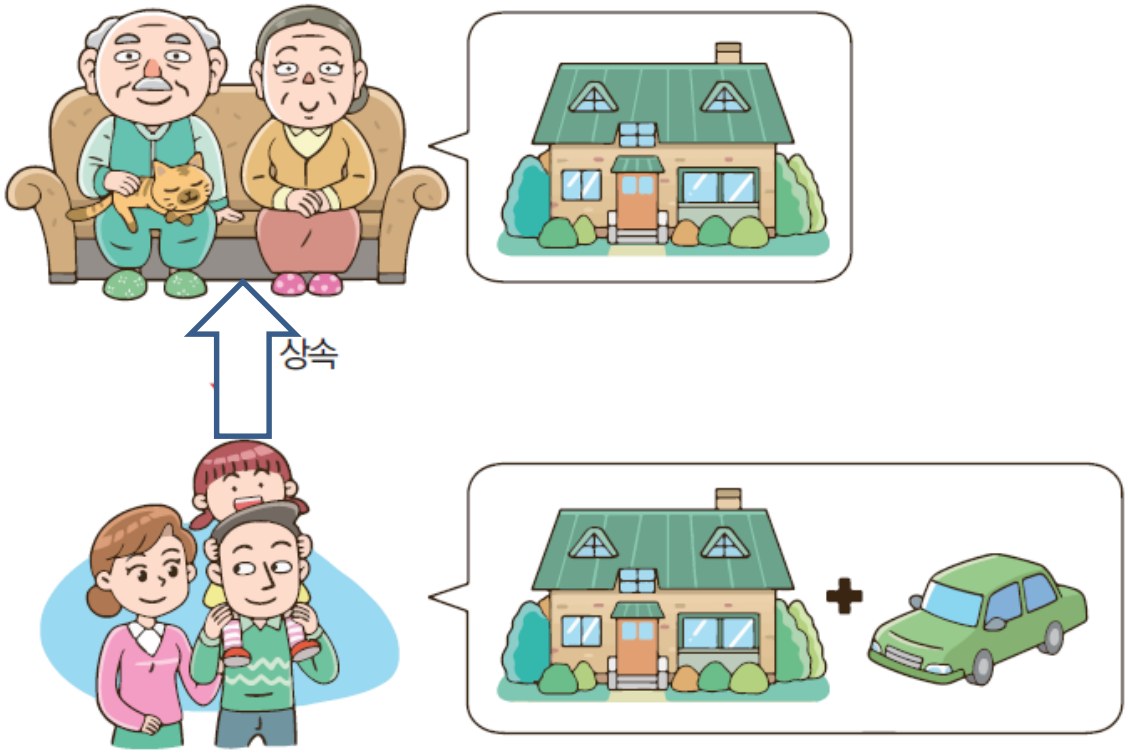
제7장 상속

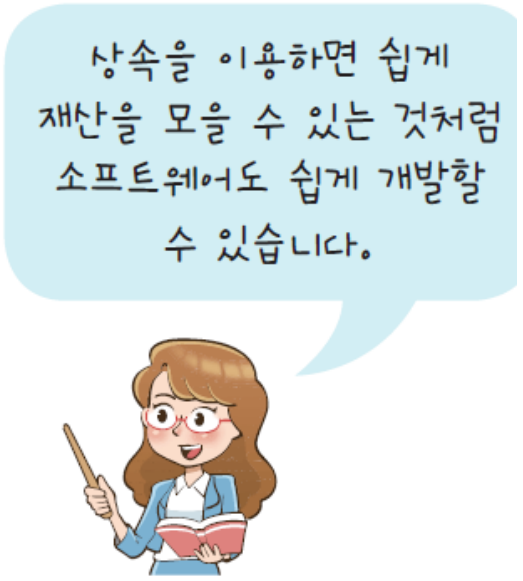
학습 내용

학습목차

상속이란?

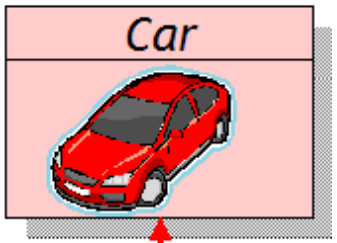
- 상속의 개념은 현실 세계에도 존재한다.



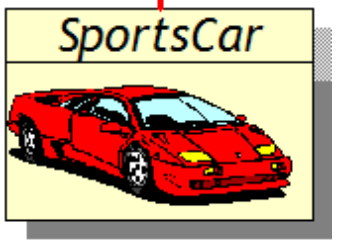


상속(inheritance)

- 클래스 간의 상속 관계는 **is-a** 관계
  - 자식은 부모의 모든 것을 계승한다.



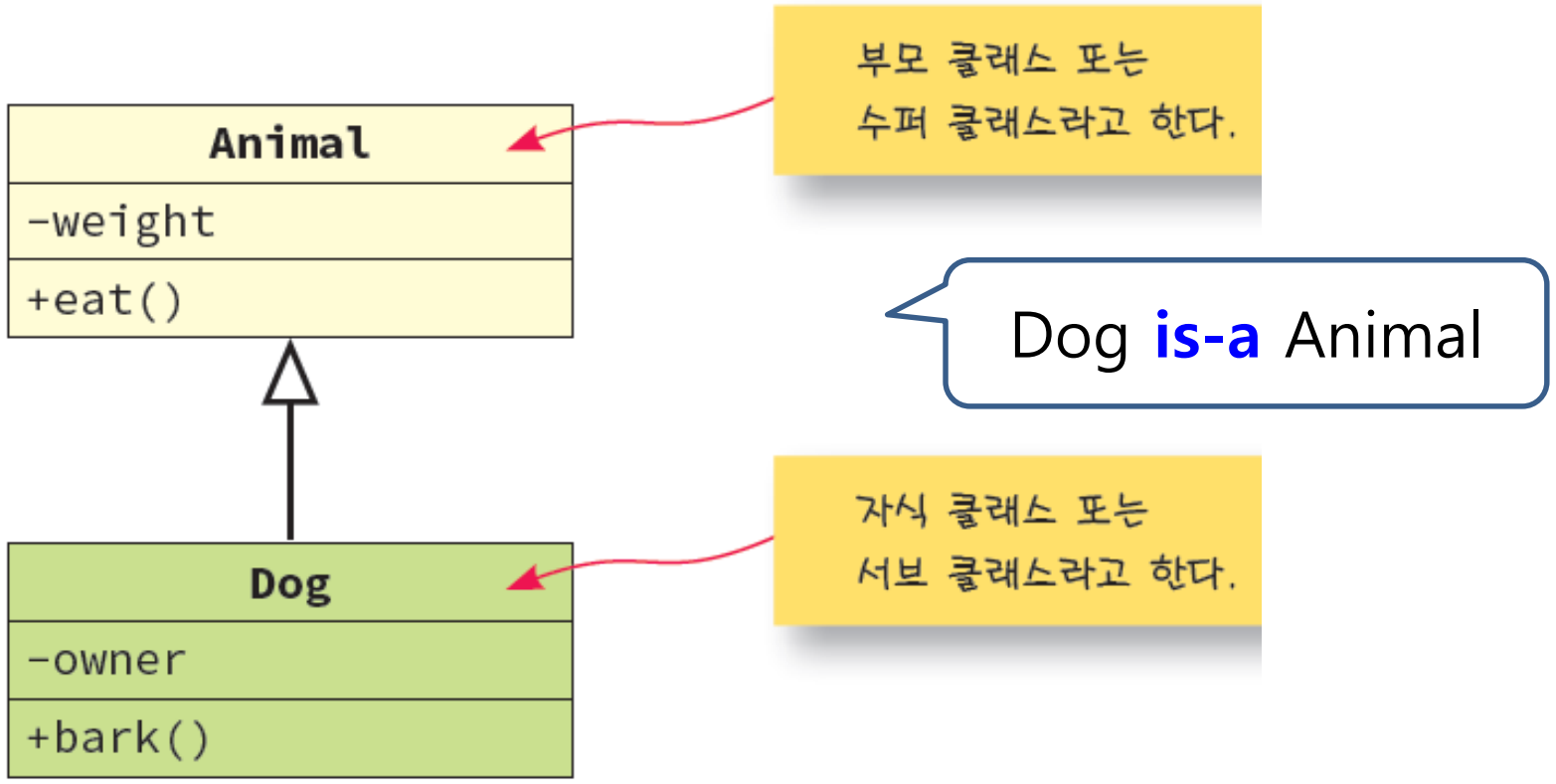
부모 클래스



자식 클래스

SportsCar **is-a** Car

부모 클래스와 자식 클래스



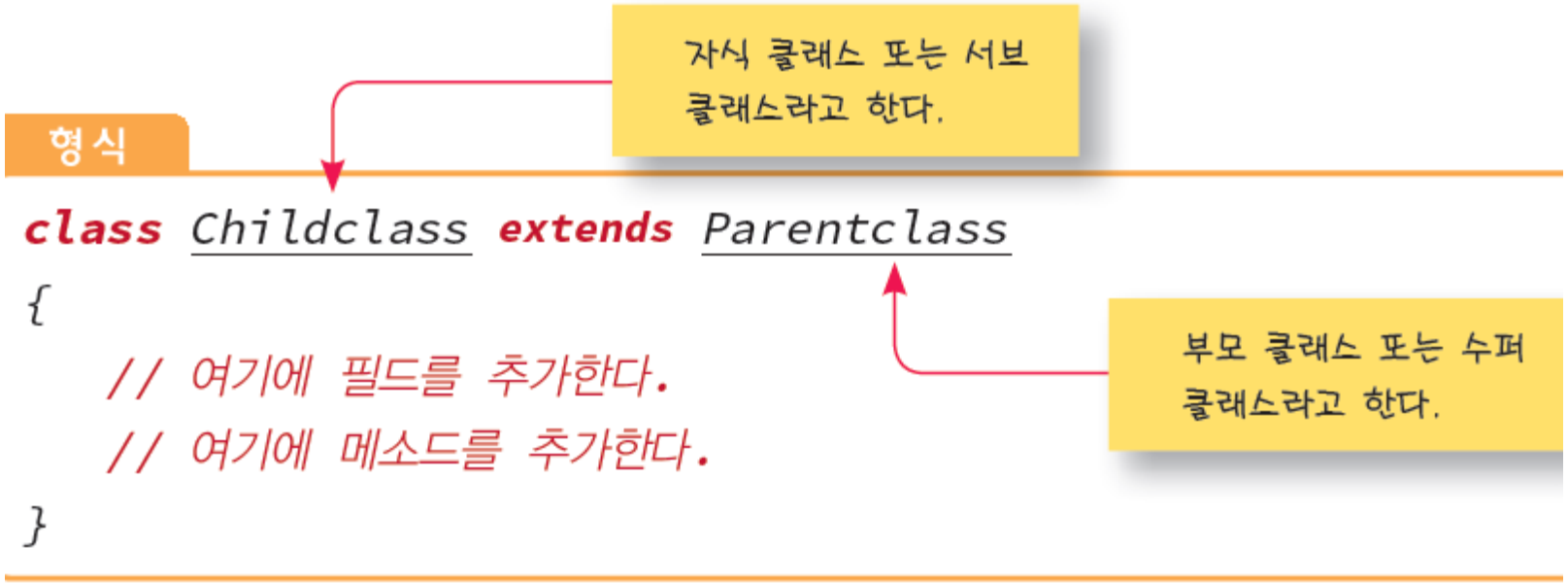
부모 클래스와 자식 클래스

- 대개 부모 클래스는 추상적이고 자식 클래스는 구체적이다.

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

상속의 형식

- 자식 클래스를 정의할 때 **extends** 키워드 사용

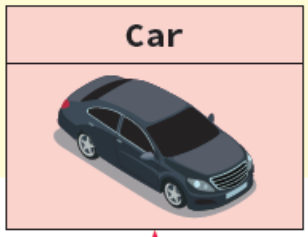


상속 예제: 자동차

Car.java

```
01 public class Car
02 {
03    int speed; // 속도
04    public void setSpeed(int speed) { // 속도 변경 메소드
05       this.speed = speed;
06    }
07 }
```

상속 설명을 위하여 private를 붙이지 않았다.



수퍼클래스 (superclass)

SportCar.java

```
01 public class SportsCar extends Car
02
03 {
04    boolean turbo;
05
06    public void setTurbo(boolean flag) { // 터보 모드 설정 메소드
07       turbo = flag;
08    }
09 }
```

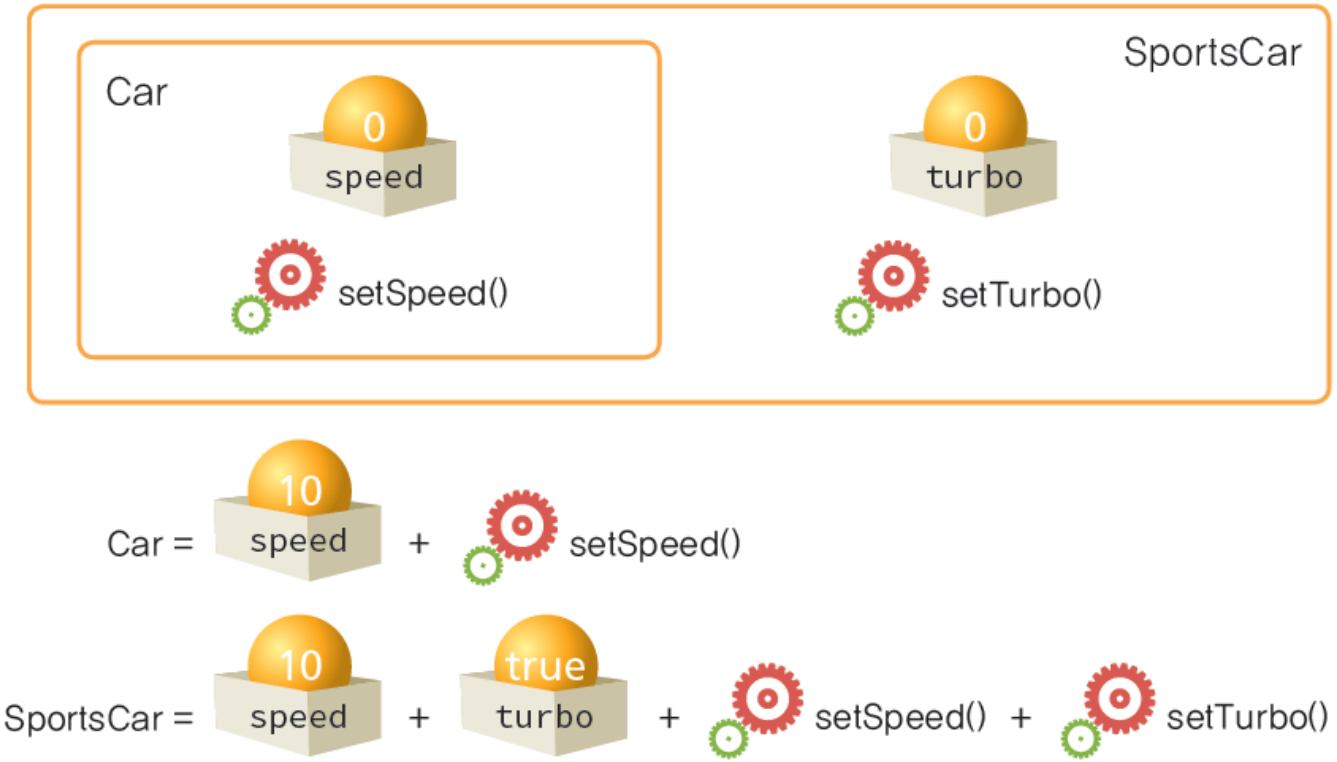
추가된 필드

서브클래스 (subclass)

추가된 메소드

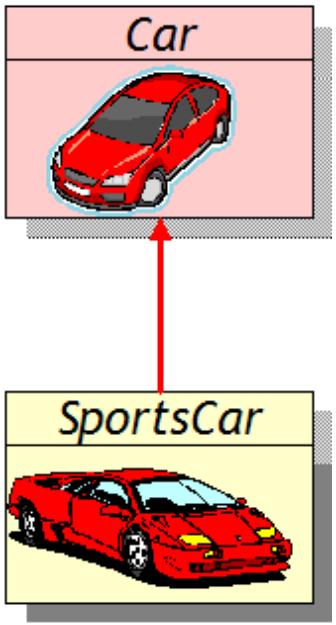
무엇이 상속되는가?

- 부모 클래스의 필드와 메소드가 자식 클래스로 상속됨
- 자식 클래스는 자신만의 필드와 메소드를 추가할 수 있음



```java
public class Car {
    int speed;
    public void setSpeed(int speed) {
        this.speed = speed;
    }
}
```

```java
public class SportsCar extends Car {
    boolean turbo;
    public void setTurbo(boolean flag) {
        turbo = flag;
    }
}
```




```java
public class SportsCarTest {
    public static void main(String[] args) {
        SportsCar s = new SportsCar();   // 자식 클래스 객체 생성
        s.speed = 10;
        s.setSpeed(60);
        s.turbo = false;
        s.setTurbo(true);
    }
}
```

부모 클래스의 필드와 메소드

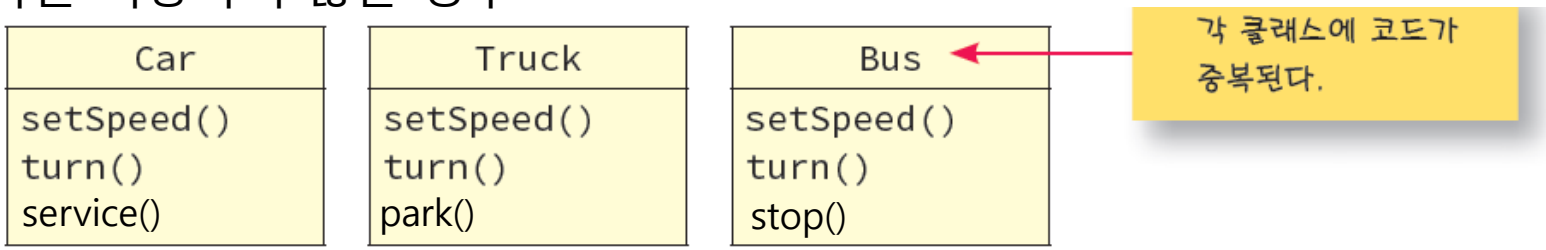
자식 클래스의 자체 필드와 메소드

상속의 장점

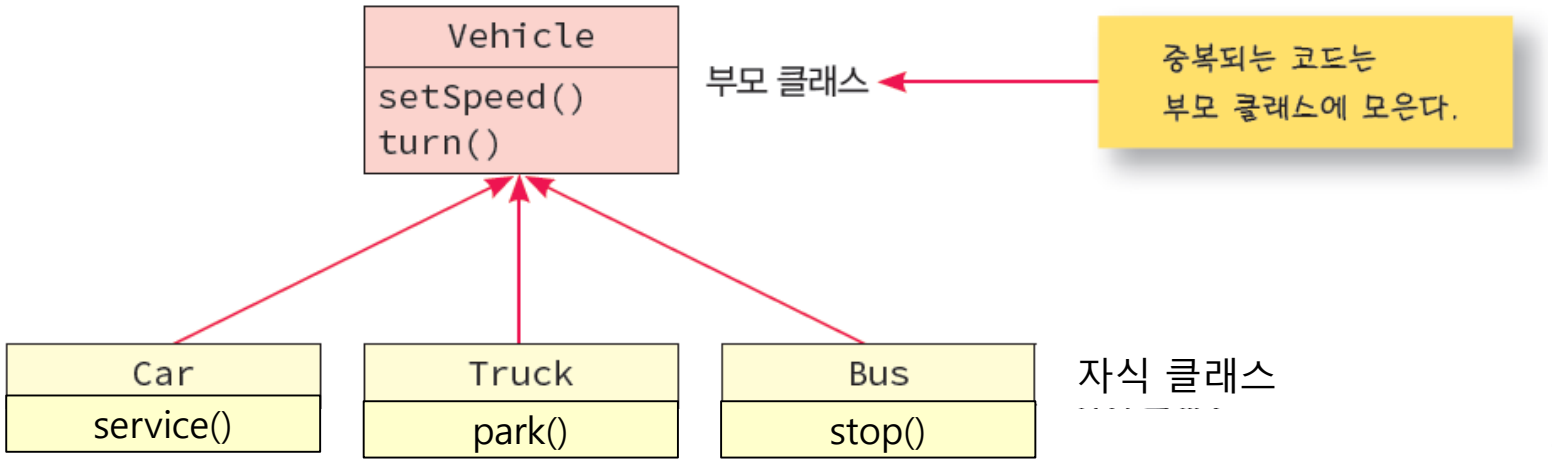
- 상속을 통하여 기존 클래스의 필드와 메소드를 재사용할 수 있음
- 이미 작성된 검증된 소프트웨어를 재사용할 수 있음
- 신뢰성 있는 소프트웨어를 손쉽게 개발, 유지 보수
- 코드의 중복을 줄일 수 있음
  - 공통적인 부분을 부모 클래스에서 정의
- 상속을 통하여 여러 클래스들을 자손으로 묶을 수 있음
  - 다형성을 이용할 수 있음
  - 자손들이 가져야 하는 규약을 명시할 수 있음

상속 – 코드의 중복을 줄임

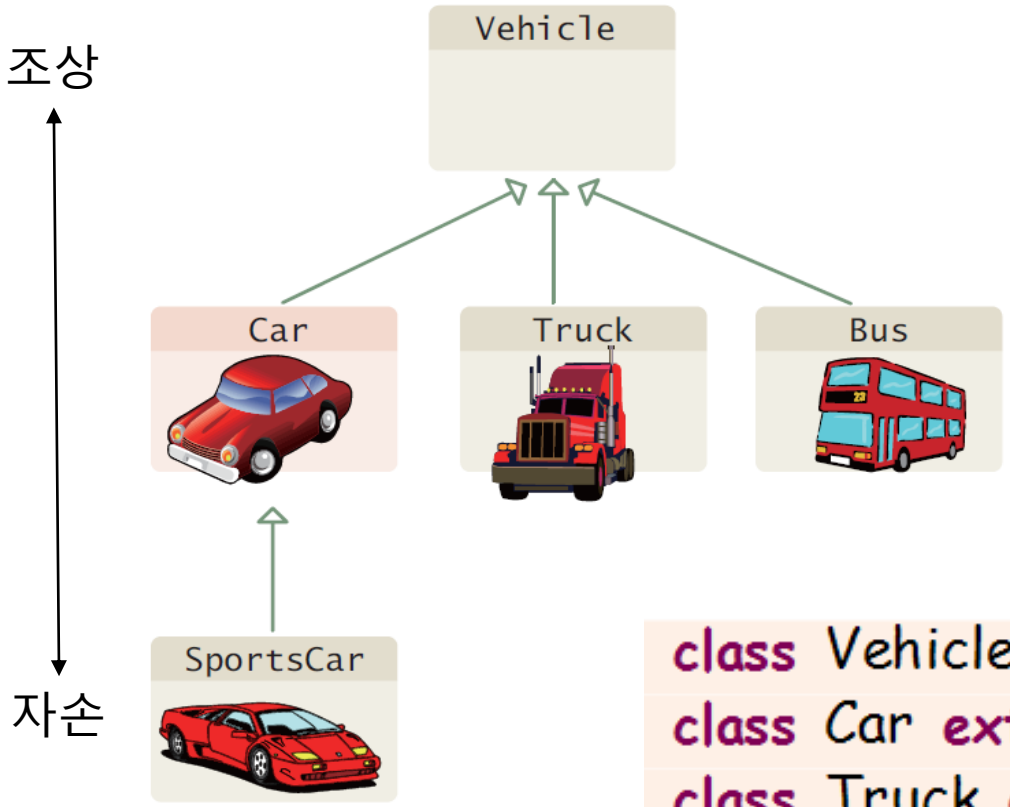
상속을 이용하지 않은 경우



상속을 이용한 경우

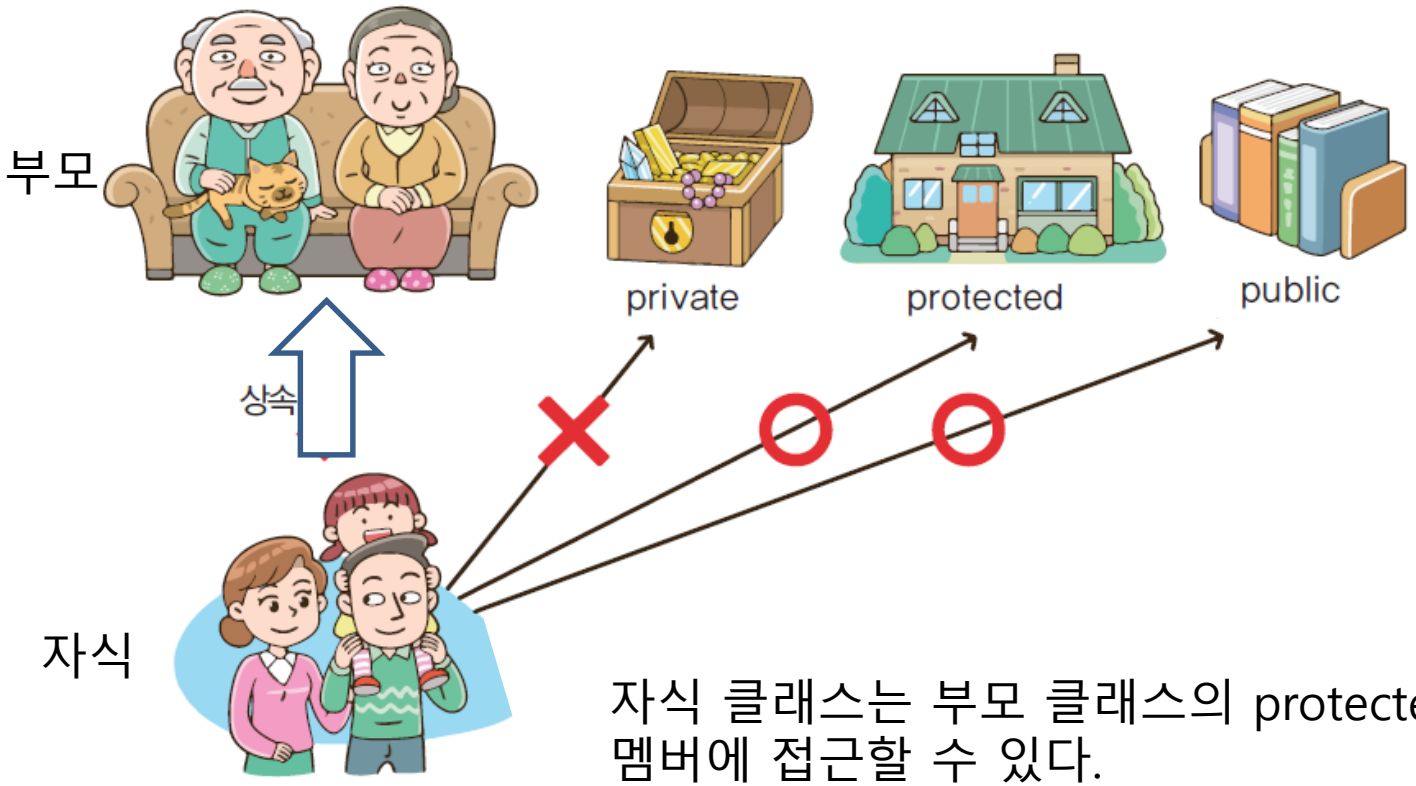


상속의 계층 구조




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

상속과 접근제어



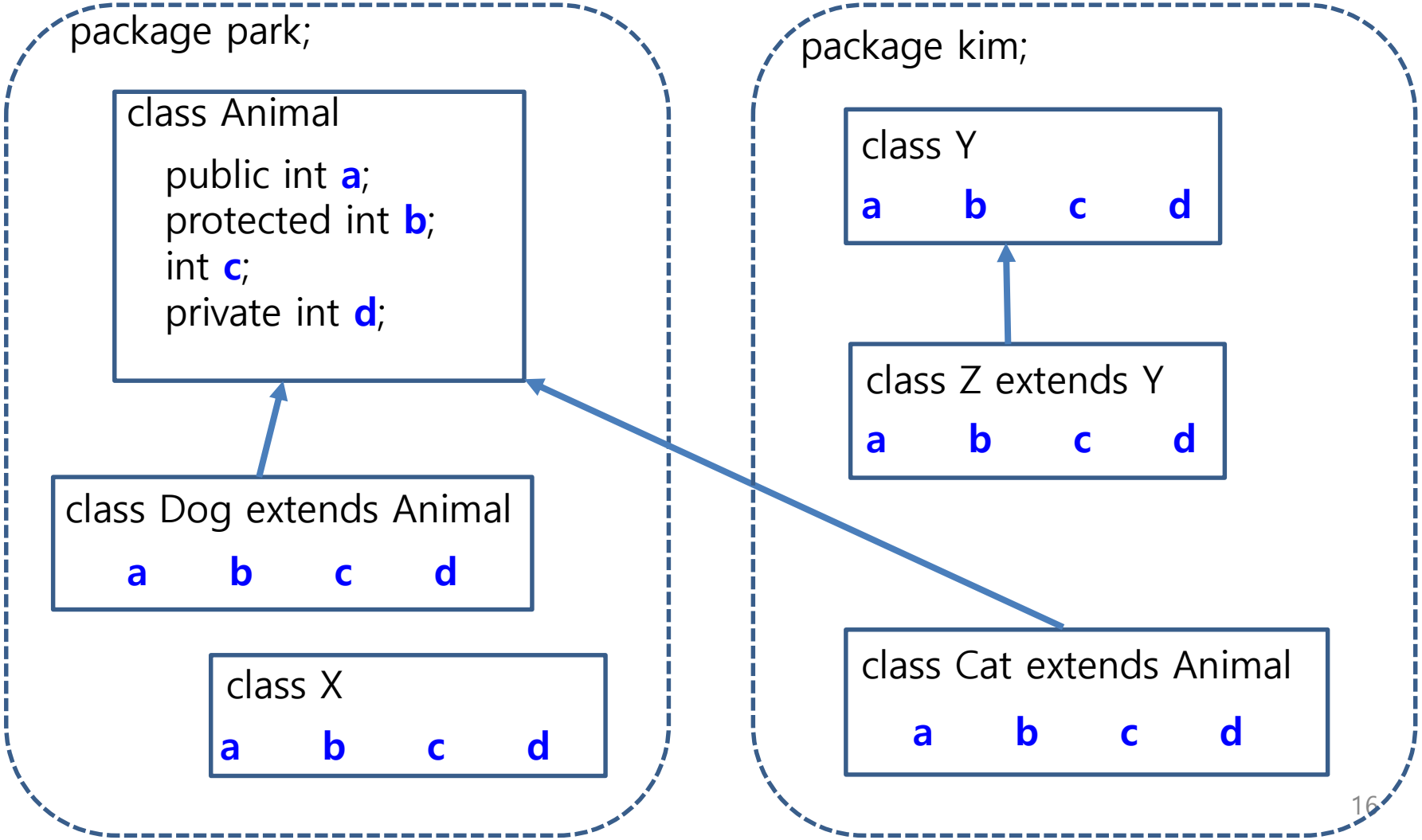
자식 클래스는 부모 클래스의 protected 멤버에 접근할 수 있다.

접근 제어 지정자

접근 지정자	클래스	패키지	**자식 클래스**	전체 세계
public	O	O	O	O
protected	**O**	**O**	**O**	**X**
없음(디폴트)	O	O	X	X
private	O	X	X	X

접근 제어 지정자

- 각 클래스에서 Animal의 a, b, c, d 중 접근 가능한 것은?



도형 예제

```
public class Shape {
        private int x;
        private int y;
}
```

```
public class Rectangle extends Shape {
        private int width;
        private int height;
        public void draw() {
                System.out.println(width + " " + height);
                System.out.println(x + " " + y);
        }
}
```

에러! 부모 클래스의 **private** 멤버인 x, y는 접근 불가능

```
public class Main {
        public static void main(String[] args) {
                Shape s = new Shape();
                System.out.println(s.x + " " + s.y);
        }
}
```

에러! 다른 클래스의 **private** 멤버인 x, y는 접근 불가능

도형 예제 : 에러 해결 방법

```
public class Shape {
    private int x;
    private int y;
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

```
public class Rectangle extends Shape {
    private int width;
    private int height;
    public void draw() {
        System.out.println(width + " " + height);
        System.out.println(getX() + " " + getY());
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Shape s = new Shape();
        System.out.println(s.getX() + " " + s.getY());
    }
}
```

도형 예제 : protected

```
public class Shape {
    protected int x;
    protected int y;
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

```
public class Rectangle extends Shape {
    private int width;
    private int height;
    public void draw() {
        System.out.println(width + " " + height);
        System.out.println(x + " " + y);
    }
}
```

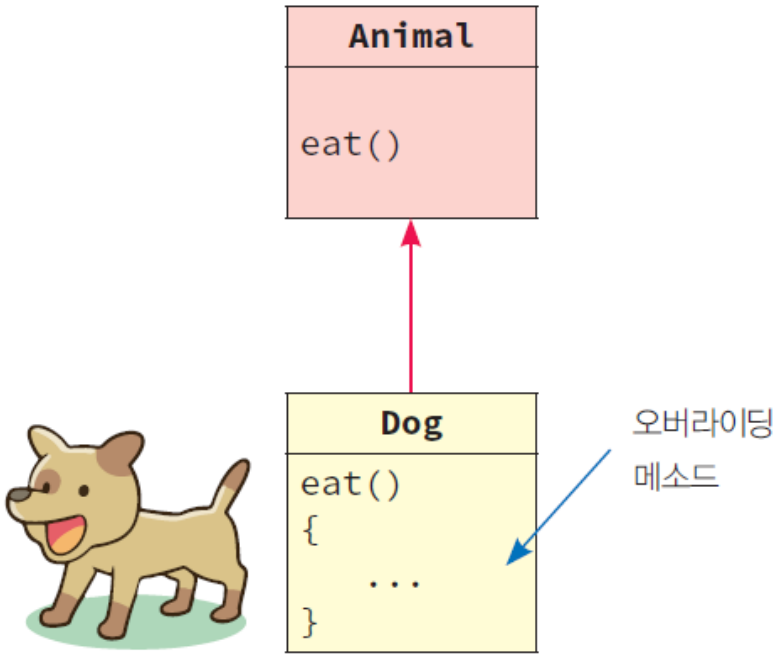
OK! 부모 클래스의 **protected** 멤버인 x, y는 접근 가능

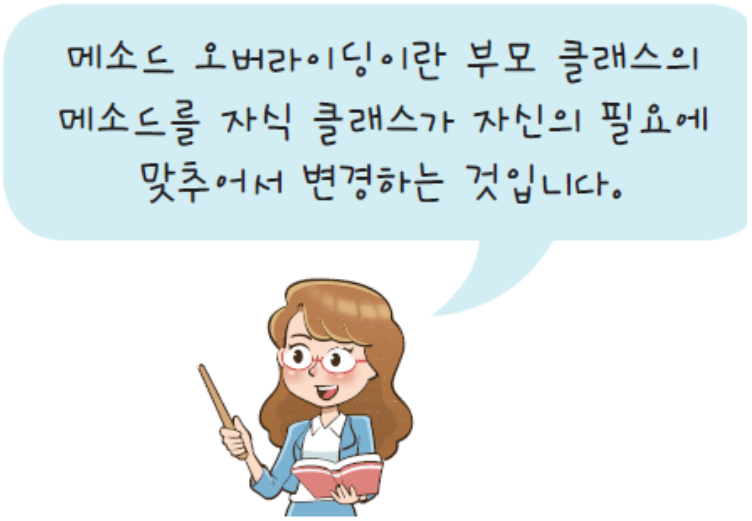
```
public class Main {
    public static void main(String[] args) {
        Shape s = new Shape();
        System.out.println(s.getX() + " " + s.getY());
        System.out.println(s.x + " " + s.y);
    }
}
```

에러! 다른 클래스의 **protected** 멤버인 x, y는 접근 불가능

메소드 오버라이딩

- 메소드 오버라이딩(method overriding)
  - 메소드 재정의
  - 상속받은 메소드를 자식 클래스가 필요에 따라 다시 정의하는 것





메소드 오버라이딩 예

```
public class Animal {
    public void eat() {
        System.out.println("동물이 먹고 있습니다.");
    }
}
```

```
public class Dog extends Animal {

    public void eat() {
        System.out.println("강아지가 먹고 있습니다.");
    }
}
```

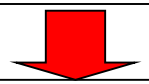
부모로부터 상속받은 eat() 메소드를 override

```
public class DogTest {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();                // 출력은?
    }
}
```

메소드 오버라이딩 유의사항

- 부모 클래스의 private 메소드는 오버라이드 할 수 없다
- 가시성(visibility)이 축소되도록 오버라이드 할 수 없다.
  - 예를 들어 public 메소드를 protected로 오버라이드 불가능
- 메소드의 이름, 반환형, 매개 변수의 개수와 데이터 타입이 일치하여야 한다.

```
public class Animal {
    public void eat() {
        System.out.println("동물이 먹고 있습니다.");
    }
}
```



메소드 오버라이딩이 아님!

```
public class Dog extends Animal {
    public void eat(int a) {
        System.out.println("강아지가 먹고 있습니다. " );
        return 0;
    }
}
```

어노테이션(annotation)

- 오버라이드 실수한 예 : 오버라이드를 잘못했는데도 오류가 발생하지 않으므로, 잘못된 것을 모른다.

```
public class Animal {
    public void eat() {
        System.out.println("동물이 먹고 있습니다.");
    }
}
```

메소드 오버라이딩이 아님!

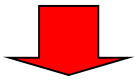
```
public class Dog extends Animal {
    public void eet() {
        System.out.println("강아지가 먹고 있습니다.");
    }
}
```

에러 발생하지 않음:
eat() 와는 무관한 다른 메소드로 인식

어노테이션(annotation)

- 오버라이드하는 메소드 앞에 어노테이션 **@Override**를 두면 오버라이드가 잘못된 경우 오류를 발생시켜 알려준다.

```
public class Animal {
      public void eat() {
            System.out.println("동물이 먹고 있습니다.");
      }
}
```



메소드 오버라이딩이 아님!

```
public class Dog extends Animal {
      @Override
      public void eet() {
            System.out.println("강아지가 먹고 있습니다.");
      }
}
```

에러! -- The method eet() of type Dog must override or implement a supertype method

super 키워드

- **super** 는 수퍼 클래스의 메소드나 필드를 명시적으로 참조하기 위해 사용
- super 사용 예
  - 서브 클래스에서 메소드를 오버라이드할 때 수퍼 클래스의 메소드를 완전히 대치하는 경우보다 내용을 추가하는 경우가 많다.
  - 이 경우 super 키워드를 이용하여 수퍼 클래스의 메소드를 호출한 후, 자신이 필요한 부분을 추가하면 된다.

super 사용 예1

```
public class Animal {
    public void eat() {
        System.out.println("동물이 먹고 있습니다.");
    }
}

public class Dog extends Animal {
    public void eat() {
        super.eat();              // 부모 클래스의 eat() 호출
        System.out.println("강아지가 먹고 있습니다.");
    }
}

public class DogTest {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();                  // Dog 클래스의 eat() 호출
    }
}
```

실행결과

```
동물이 먹고 있습니다.
강아지가 먹고 있습니다.
```

super 사용하지 않은 예

```
public class Employee {
    private int salary = 100;
    public int getIncome() {
        return salary;
    }
}

public class Manager extends Employee {
    private int bonus = 20;
}

public class ManagerTest {
    public static void main(String[] args) {
        Employee e = new Employee();
        Manager m = new Manager();
        System.out.println(e.getIncome());
        System.out.println(m.getIncome());
    }
}
```

실행결과

super 사용 예2

```
class Employee {
    private int salary = 100;
    public int getIncome() {
        return salary;
    }
}
class Manager extends Employee {
    private int bonus = 20;
    public int getIncome() {
        return super.getIncome() + bonus;
    }
}
public class ManagerTest {
    public static void main(String[] args) {
        Employee e = new Employee();
        Manager m = new Manager();
        System.out.println(e.getIncome());
        System.out.println(m.getIncome());
    }
}
```

실행결과

상속과 생성자

- 서브 클래스의 객체가 생성될 때, 서브 클래스의 생성자만 호출될까? 아니면 수퍼 클래스의 생성자도 호출될까?

```
class Shape {
    public Shape() {
        System.out.println("Shape 생성자");
    }
}
class Rectangle extends Shape {
    public Rectangle() {
        System.out.println("Rectangle 생성자");
    }
}
public class RectangleTest {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
    }
}
```

생성자의 실행 순서는
(1) 부모 클래스의 생성자
(2) 자식 클래스의 생성자

실행결과

```
Shape 생성자
Rectangle 생성자
```

명시적인 생성자 호출

- **super()**를 이용하여 명시적으로 수퍼 클래스의 생성자 호출

```
class Shape {
    public Shape() {
        System.out.println("Shape 생성자");
    }
}
class Rectangle extends Shape {
    public Rectangle() {
        super();  // 반드시 생성자의 첫번째 줄
        System.out.println("Rectangle 생성자");
    }
}
public class RectangleTest {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
    }
}
```

실행결과

```
Shape 생성자
Rectangle 생성자
```

묵시적인 생성자 호출

- 명시적으로 수퍼 클래스의 생성자를 호출하지 않으면 자동으로 수퍼 클래스의 디폴트 생성자가 호출된다.

```
class Shape {
    public Shape() {
        System.out.println("Shape 생성자");
    }
}
class Rectangle extends Shape {
    public Rectangle() {
        System.out.println("Rectangle 생성자");
    }
}
public class RectangleTest {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
    }
}
```

컴파일러가 자동으로 super(); 를 넣어주는 것으로 보면 됨

실행결과

```
Shape 생성자
Rectangle 생성자
```

생성자 호출 유의사항

- 다음 프로그램의 오류 발생 이유는?

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

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

public class RectangleTest {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
    }
}
```

생성자가 하나라도 정의되어 있으면 기본 생성자가 자동으로 추가되지 않음

오류: 묵시적으로 Shape()를 호출할 수 없음

super() 사용 예

```
public class Employee {
    private int salary;

    public Employee(int salary) {
        this.salary = salary;
    }
    public int getIncome() {
        return salary;
    }
}
```

```
public class Manager extends Employee {
    private int bonus;

    public Manager(int salary, int bonus) {
        super(salary);   // 부모의 생성자 호출
        this.bonus = bonus;
    }
    public int getIncome() {
        return super.getIncome() + bonus;
    }
}
```

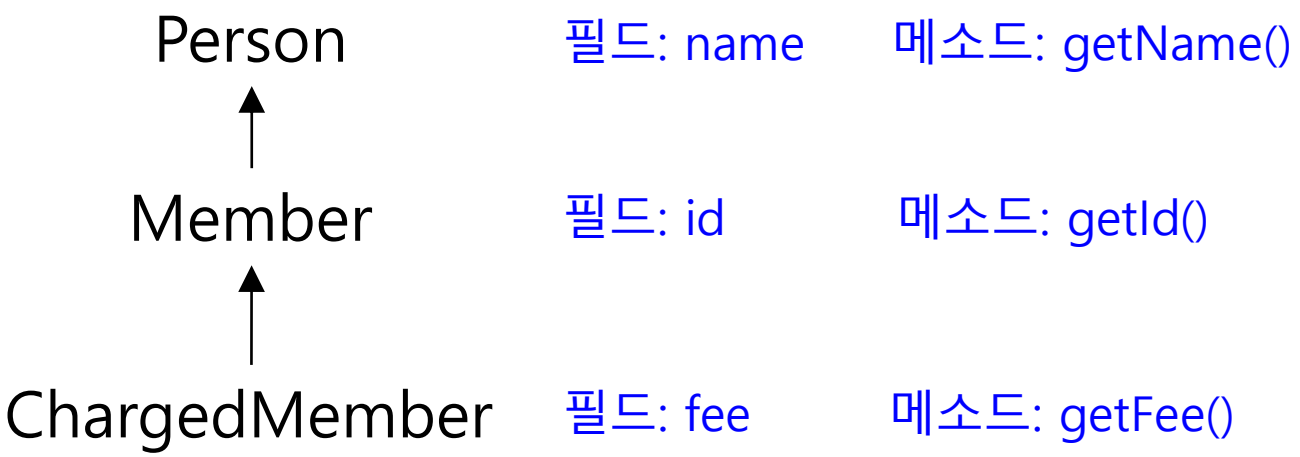
```
public class ManagerTest {
    public static void main(String[] args) {
        Manager m = new Manager(100, 20);
        System.out.println(m.getIncome());
    }
}
```

실행결과

```
120
```

상속 계층 구조

- Person의 자식 클래스 Member
- Member의 자식 클래스 ChargedMember
- 이 경우 ChargedMember는 Person과 Member로부터 모두 상속받음



```java
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

```java
public class Member extends Person {
    private int id;
    public Member(String name, int id) {
        super(name);
        this.id = id;
    }
    public int getId() {
        return id;
    }
}
```

```java
public class ChargedMember extends Member {
    private int fee;
    public ChargedMember(String name, int id, int fee) {
        super(name, id);
        this.fee = fee;
    }
    public int getFee() {
        return fee;
    }
}
```

```java
public class ManagerTest {
    public static void main(String[] args) {
        ChargedMember m = new ChargedMember("kim", 111, 9000);
        System.out.println(m.getName() + m.getId() + m.getFee());   // kim1119000
    }
}
```

추상 클래스

미완성인 메소드를 지닌 클래스로서, 메소드가 미완성이므로 추상클래스로는 객체를 만들 수 없다.

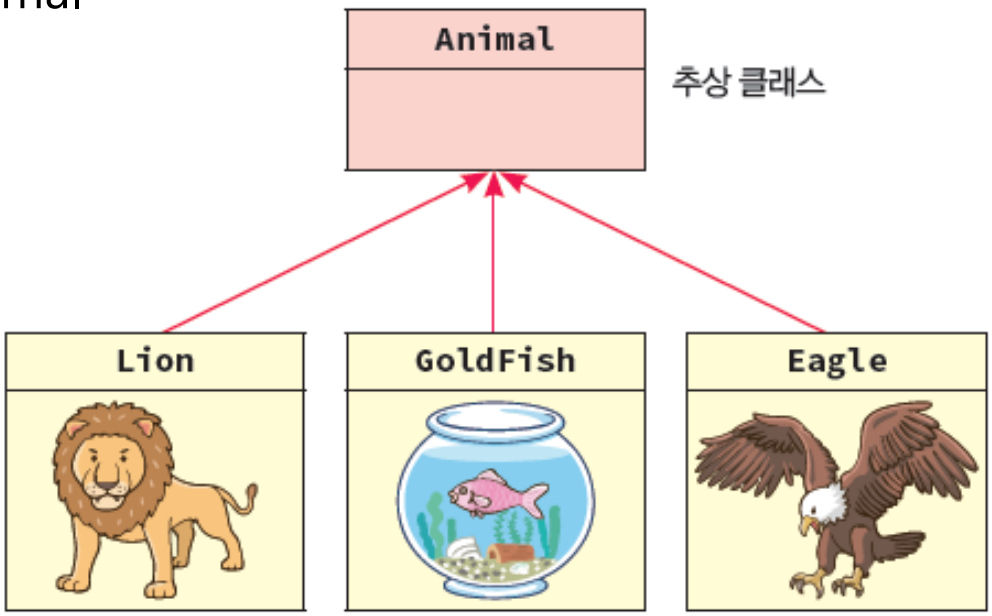
- 추상 클래스(abstract class)
  - abstract로 선언된 클래스
  - 추상 메소드(abstract method)를 가질 수 있다.
    - 추상 메소드는 메소드 몸체(method body)가 없이 abstract로 선언된 메소드
  - 필드와 일반 메소드를 가질 수 있지만, 객체 생성(instantiation)이 불가능하다.

```
public abstract class Animal {              // 추상 클래스 Animal
        public abstract void move();        // 추상 메소드 move()
                    …
}
```

move() 메소드 정의에서 몸체 { } 가 없이 ; 기호로 끝난다.

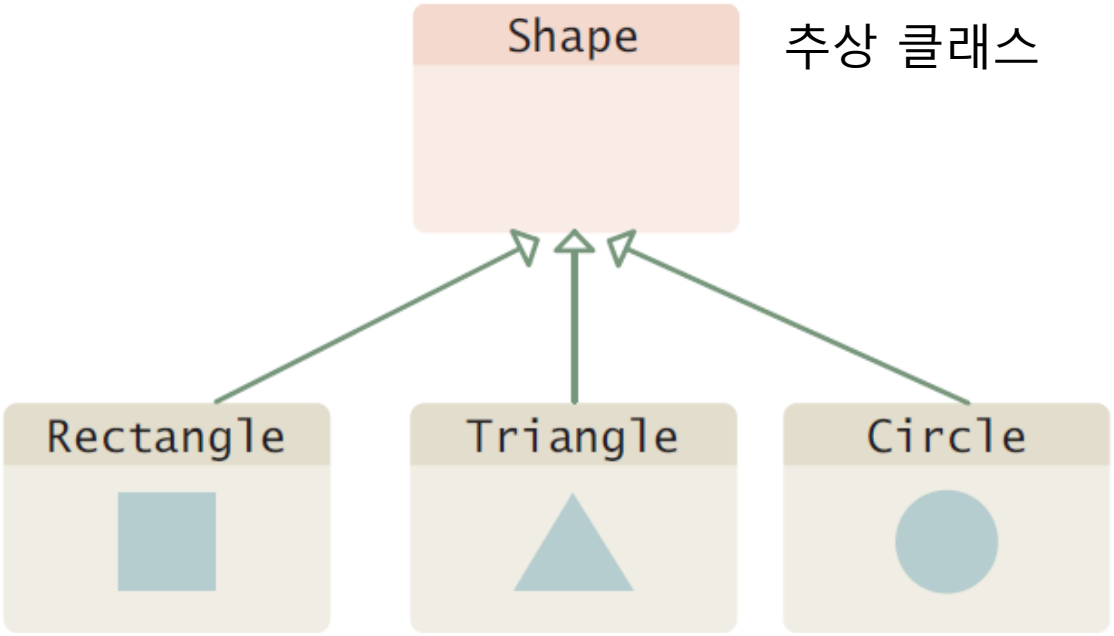
추상 클래스

- 추상 클래스는 추상적인 개념을 표현하는 데 적당하다.
  - 예) Animal



- 따라서 추상 클래스는 객체를 생성할 수 없다.
  - 예) Animal animal = new Animal(); // 오류

추상 클래스 예



```
abstract class Shape {
    int x, y;
    public void move(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public abstract double area();
}

class Rectangle extends Shape {
    double width, height;
    public double area() {
        return width * height;
    }
}

class Circle extends Shape {
    double radius;
    public double area() {
        return 3.14 * radius * radius;
    }
}
```

추상클래스 Shape

추상 클래스에 일반 메소드를 둘 수 있다.

area() 메소드를 추상메소드로 선언 – "도형"은 추상적인 개념이므로 면적을 구하는 것이 무의미하므로 메소드 body가 없는 추상메소드로 선언함

서브클래스에서 area()를 구현 – 사각형 면적 공식에 따라

서브클래스에서 area()를 구현 – 원 면적 공식에 따라

다형성이란?

- 다형성(polymorphism)이란 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 서로 다른 동작을 하는 것

자바에서의 자료형 검사

- 자바의 자료형 검사는 엄격하다.
  - 클래스도 일종의 자료형

```
class A {

}

class B {

}

public class TypeTest1 {
        public static void main(String[] args) {
                A a = new B();    // 에러!
        }
}
```

클래스 A형 변수로 클래스 B 객체를 참조할 수 없다.

상향 형변환

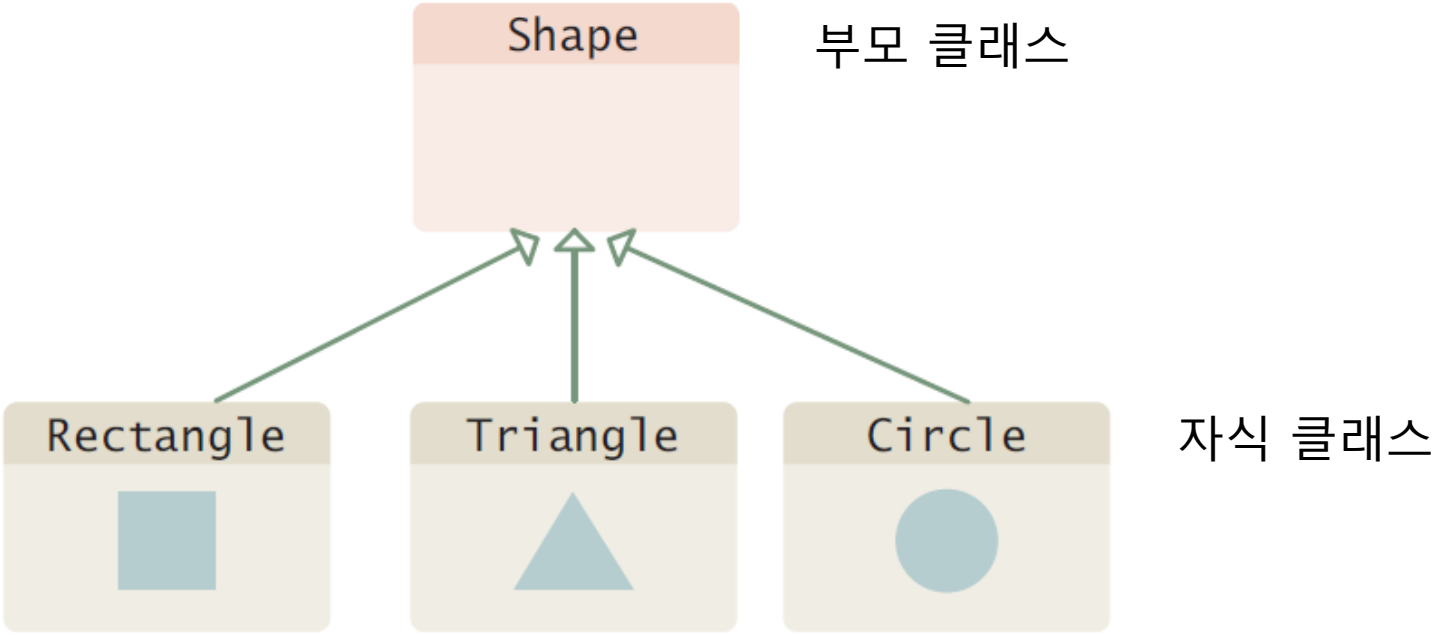
- 그러나 예외는 있다.
- 부모 클래스의 참조 변수는 자식 클래스의 객체를 참조할 수 있다!

```
class A {

}

class B extends A {

}

public class TypeTest2 {
        public static void main(String[] args) {
                A a = new B();     // OK!
        }
}
```

상향 형변환

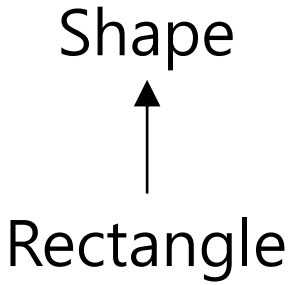
상속과 다형성

- Rectangle, Triangle, Circle의 부모 클래스가 Shape 클래스라고 하자.



상향 형변환

```
class Shape {
    int x
    int y;
}
class Rectangle extends Shape {
    int width;
    int height;
}
```



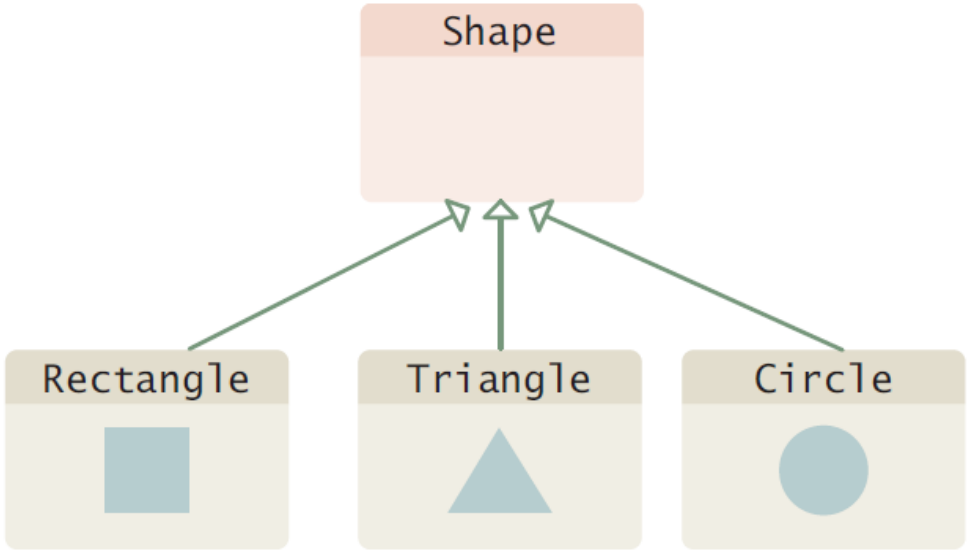

```
public class ShapeTest {
    public static void main(String[] arg) {
        Shape s1, s2;
        s1 = new Shape();       // ① 당연히 가능
        s2 = new Rectangle(); // ② Shape 변수로 Rectangle 객체 참조 가능
    }
}
```

자동적으로 Rectangle 객체 레퍼런스가
Shape형 레퍼런스로 상향 형변환됨

왜 그럴까?

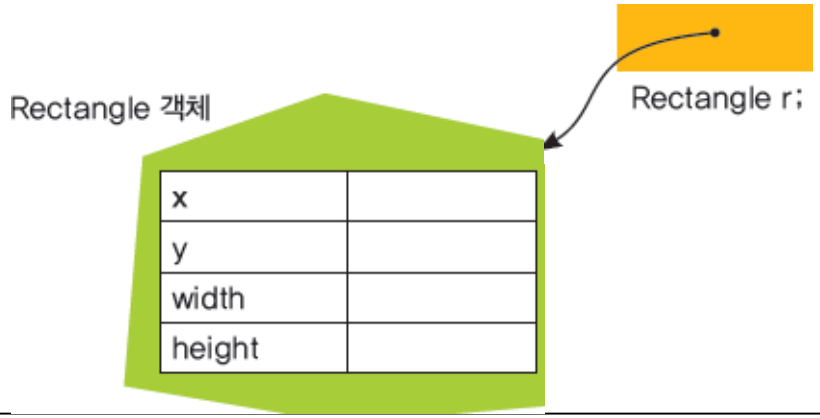
- 자식 클래스 객체도 일종의 부모 클래스 객체이기 때문이다.
  - Rectangle **is-a** Shape
  - Rectangle 객체는 일종의 Shape 객체
  - 다음 코드는 모두 OK

```
Shape s;
s = new Shape();
s = new Rectangle();
s = new Triangle();
s = new Circle();
```



상향 형변환 유의사항

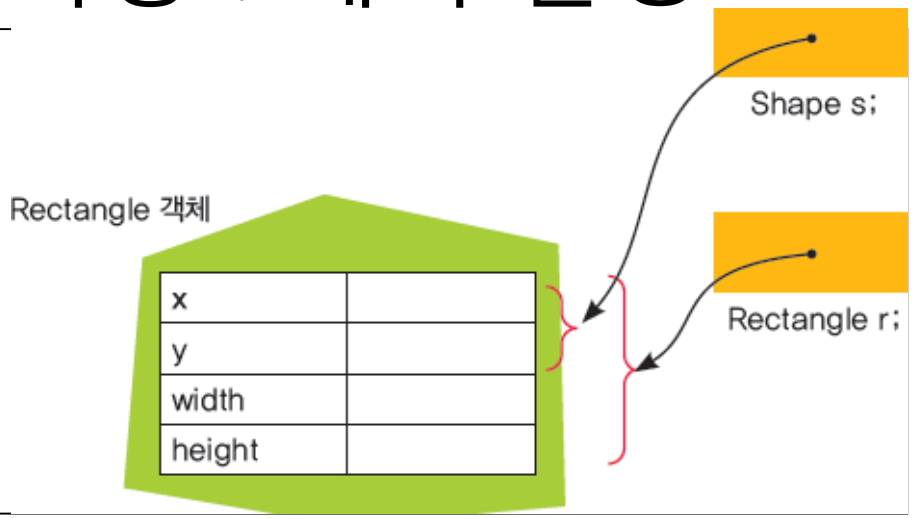
```
class Shape {
    int x
    int y;
}
class Rectangle extends Shape {
    int width;
    int height;
}
```




```
public class ShapeTest {
    public static void main(String[] arg) {
        Rectangle r;
        r = new Rectangle();        // 형변환이 필요 없음
        r.x = 0;
        r.y = 0;
        r.width = 100;
        r.height = 100;
    }
}
```

상향 형변환 유의사항 : 에러 발생

```
class Shape {
    int x
    int y;
}
class Rectangle extends Shape {
    int width;
    int height;
}
```




```
public class ShapeTest2 {
    public static void main(String[] arg) {
        Shape s;
        s = new Rectangle();  // 상향 형변환
        s.x = 0;
        s.y = 0;
        s.width = 100;    // 컴파일 에러!
        s.height = 100;   // 컴파일 에러!
    }
}
```

s가 가리키는 객체는 Rectangle 객체이지만, s가 Shape형 참조 변수이므로 s를 통해서는 Shape의 필드만 접근할 수 있다.

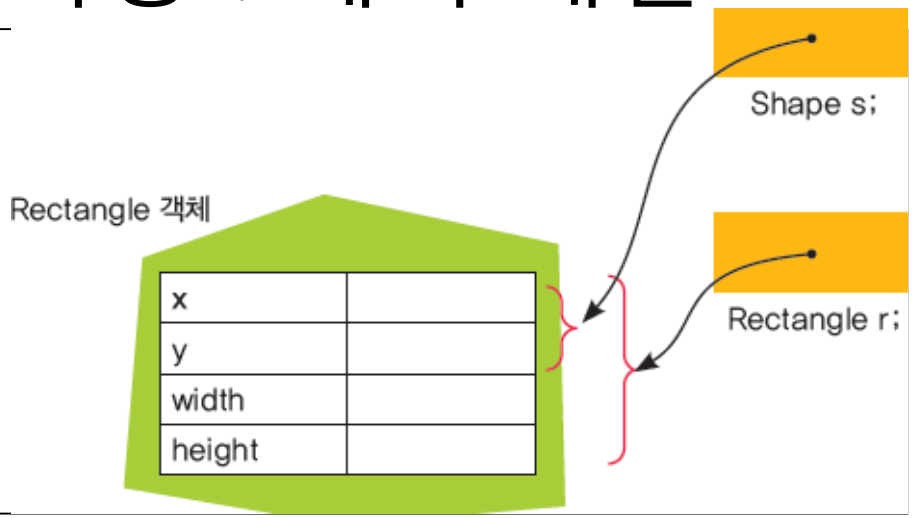
하향 형변환

- 수퍼 클래스를 서브 클래스로 하향 형변환 가능한 경우도 있다.
- 예) Shape를 Rectangle로 형변환
  - 앞의 ShapeTest2 예제에서 s를 통하여 Rectangle 클래스의 필드와 메소드를 사용하고자 할 때 형변환이 필요함

```
Shape s = new Rectangle();
s.width = 100;                      // 에러
((Rectangle) s).width = 100;        // OK! 하향 형변환
```

상향 형변환 유의사항 : 에러 해결

```
class Shape {
    int x
    int y;
}
class Rectangle extends Shape {
    int width;
    int height;
}
```




```
public class ShapeTest2 {
    public static void main(String[] arg) {
        Shape s;
        s = new Rectangle();  // 상향 형변환
        s.x = 0;
        s.y = 0;
        ((Rectangle) s).width = 100;          // OK!
        ((Rectangle) s).height = 100;         // OK!
    }
}
```

하향 형변환

- 수퍼 클래스 변수가 수퍼 클래스 객체를 참조하면 하향 형변환 하더라도 실행시간 에러 발생

```
Shape s = new Shape();
Rectangle r = (Rectangle) s;        // 실행시간 에러!
```

- 수퍼 클래스 변수가 서브 클래스 객체를 참조하는 경우, 수퍼 클래스 변수의 참조값을 서브 클래스 형으로 하향 형변환 가능

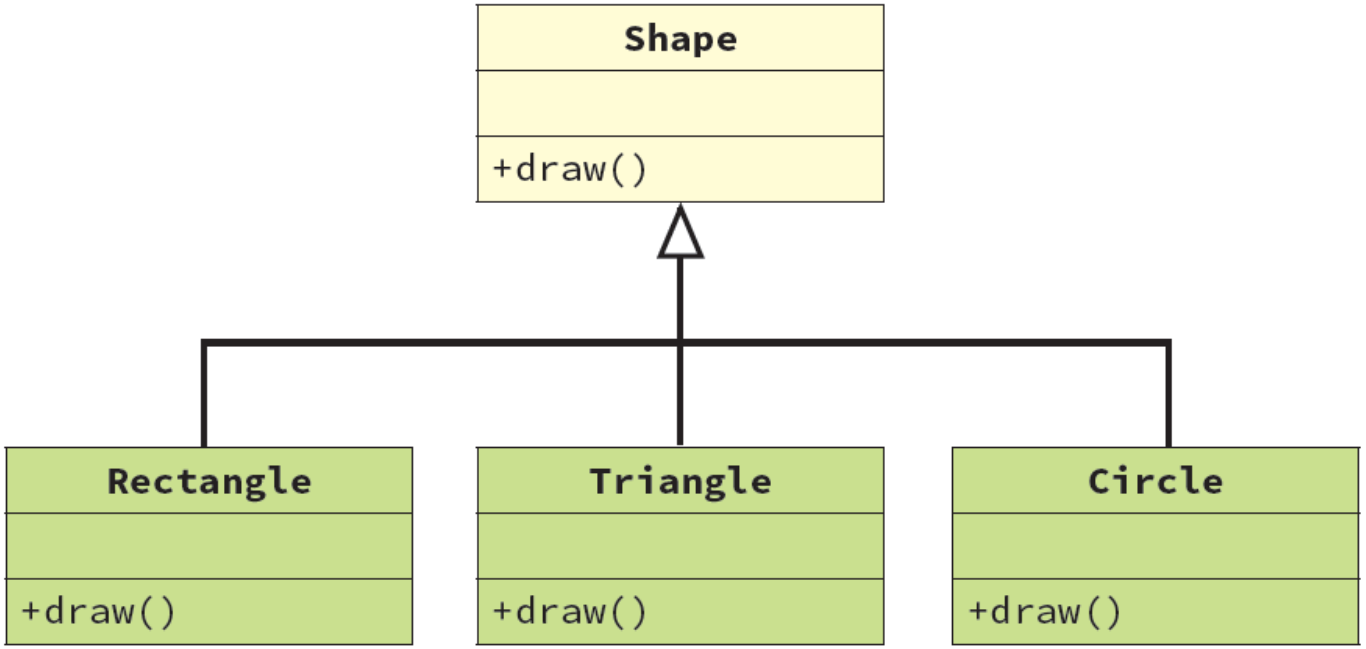
```
Shape s = new Rectangle();
Rectangle r = (Rectangle) s;        // OK!
r.width = 100;
r.height = 100;
```

동적 바인딩

- 메소드 호출을 실제 메소드와 연결하는 것을 바인딩(binding)이라고 한다.
- 상속 관계에서 오버라이드된 같은 이름의 메소드들 중에서 어떤 메소드를 호출할 것인가?
  - 자바 가상 머신(JVM)은 실행 단계에서 객체의 타입을 보고 적절한 메소드를 호출한다. 이것을 동적 바인딩(dynamic binding)이라고 한다.

동적 바인딩과 다형성

- 객체들이 동일한 메시지(예를 들어 draw())를 받더라도 각 객체의 타입에 따라서 서로 다른 동작을 한다. → 다형성



예1

```
public class Shape {
    protected int x, y;
    public int getX() { return x; }
    public int getY() { return y; }
    public void draw() {
        System.out.println("Shape " + x + " " + y);
    }
}
```

```
public class Rectangle extends Shape {
    private int width, height;
    public void draw() {
        System.out.println("Rectangle "+ x + " " + y + " " + width + " " + height);
    }
}
```

```
public class Triangle extends Shape {
    private int base, height;
    public void draw() {
        System.out.println("Triangle " + x + " " + y + " " + base + " " + height);
    }
}
```

예1 - 계속

```
public class ShapeTest {
    public static void main(String[] arg) {
        Shape s1, s2, s3;
        s1 = new Shape();
        s2 = new Rectangle();
        s3 = new Triangle();
        s1.draw();              // 동적 바인딩
        s2.draw();              // 동적 바인딩
        s3.draw();              // 동적 바인딩
    }
}
```

실행시간에 s3이 Triangle 객체를 가리키므로, Triangle의 draw()를 호출

```
Shape 0 0
Rectangle 0 0 0 0
Triangle 0 0 0 0
```

만일 동적 바인딩이 아니라면 (즉, 정적 바인딩이라면) 출력은?

예2

```
public class ShapeTest {
    private static Shape[] shapes;

    public static void main(String[] arg) {
        init();
        drawAll();
    }
    private static void init() {
        shapes = new Shape[3];
        shapes[0] = new Rectangle();
        shapes[1] = new Triangle();
        shapes[2] = new Triangle();
    }
    private static void drawAll() {
        for (int i = 0; i < shapes.length; i++) {
            shapes[i].draw();
        }
    }
}
```

```
Rectangle 0 0 0 0
Triangle 0 0 0 0
Triangle 0 0 0 0
```

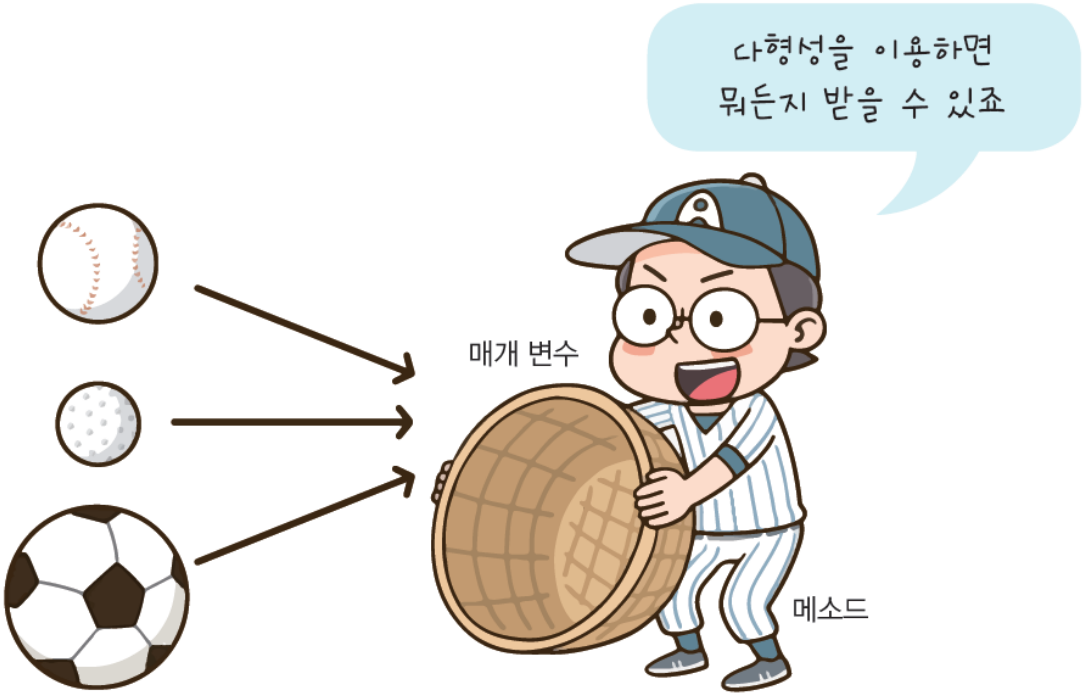
어떤 장점이 있을까?

```
public class Circle extends Shape {
    private int radius;

    public void draw(){
        System.out.println("Circle " + x + " " + y + " " + radius);
    }
}
```

- 위와 같은 새로운 클래스가 추가되더라도 다른 코드는 변경할 필요가 없다.

메소드의 매개 변수

- 매개변수를 부모 클래스 타입으로 선언하면 훨씬 넓은 범위의 객체를 받을 수 있다. → 다형성을 이용하는 전형적인 방법



매개변수와 다형성 예

- 형식 매개변수가 부모 클래스인 경우, 실 매개변수로 자식 클래스 사용 가능

```
public class ShapeTest {
    public static void main(String[] arg) {
        Rectangle r = new Rectangle();
        Circle c = new Circle();

        printShape(r);   // 매개변수 전달시 상향 형변환
        printShape(c);   // 매개변수 전달시 상향 형변환
    }

    private static void printShape(Shape s) {
        s.draw();
        System.out.println(s.getX() + " " + s.getY());
    }
}
```

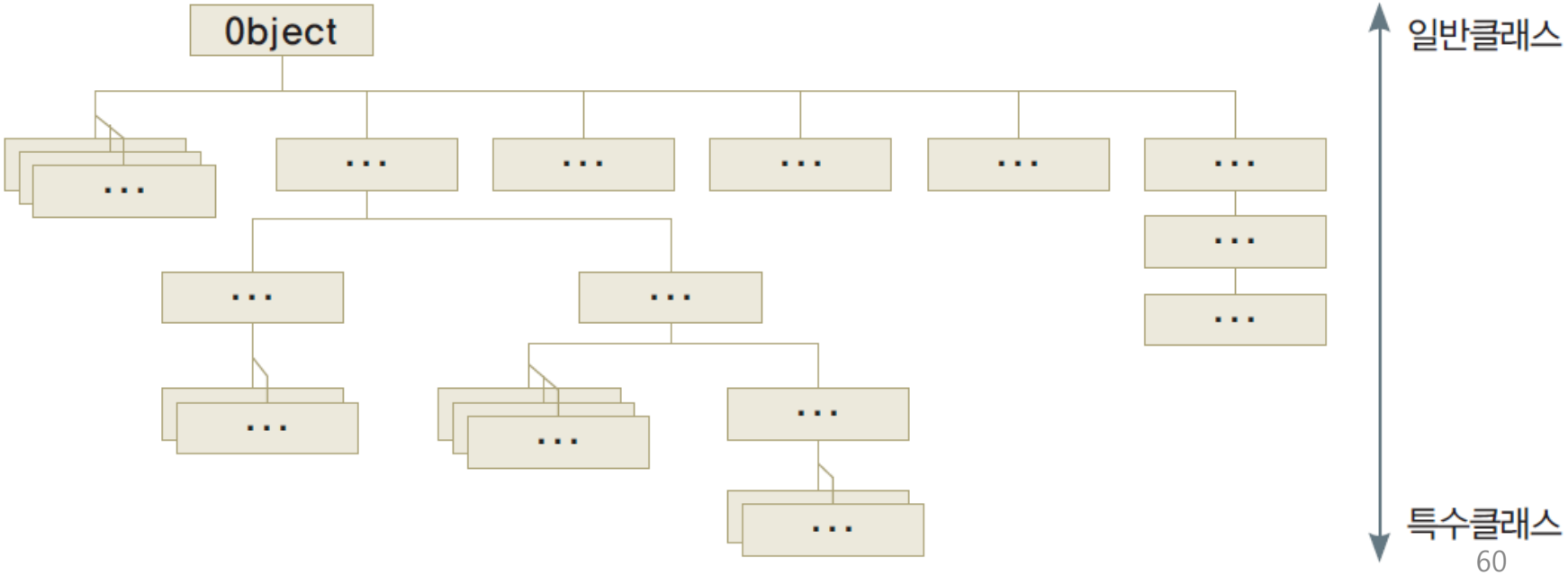
instanceof 연산자

- 객체가 특정 클래스(타입)인가를 검사

```
public class ShapeTest {
    public static void main(String[] arg) {
        Rectangle r = new Rectangle();
        Circle c = new Circle();

        printShape(r);   // 매개변수 전달시 상향 형변환
        printShape(c);   // 매개변수 전달시 상향 형변환
    }

    private static void printShape(Shape s) {
        s.draw();
        if(s instanceof Circle)     // s가 Circle 타입인가? true/false
            System.out.println("원주율은 3.14");
    }
}
```

Object 클래스

- Object 클래스는 자바 클래스 계층 구조에서 맨 위에 위치
  - 모든 클래스의 조상
  - java.lang 패키지에 들어 있음
- 부모 클래스를 명시적으로 지정하지 않으면 Object 클래스의 자식 클래스로 간주됨



Object 클래스의 메소드

메소드	설명
`Object clone( )`	객체 자신의 복사본을 생성하여 반환한다.
`boolean equals(Object obj)`	obj가 현재 객체와 같은지를 반환한다.
`void finalize()`	사용되지 않는 객체가 제거되기 직전에 호출된다.
`class getClass()`	실행 시간에 객체의 클래스 정보를 반환한다.
`int hashCode( )`	객체에 대한 해쉬 코드를 반환한다.
`String toString( )`	객체를 기술하는 문자열을 반환한다.

- 모든 클래스가 Object의 자손이므로 모든 클래스가 위의 메소드들을 상속받음
- 자식 클래스가 용도에 맞게 오버라이드 할 수 있음

getClass() 메소드

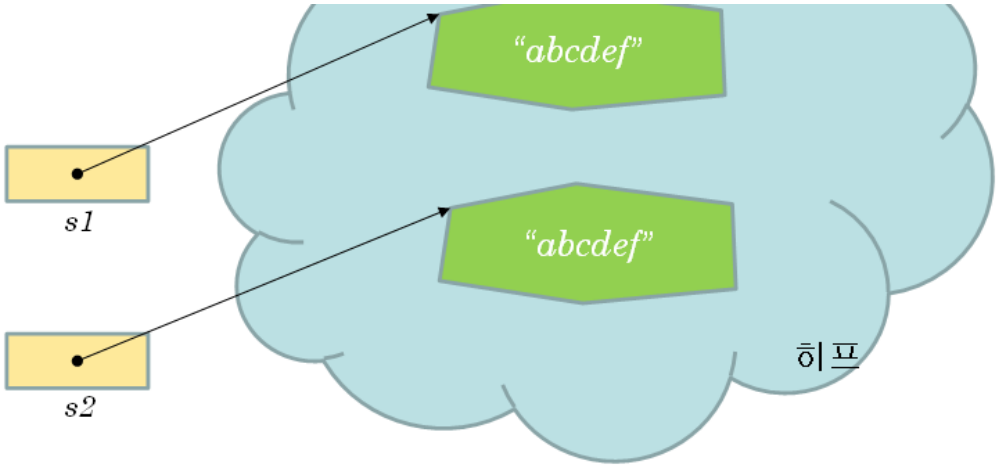
- 객체가 어떤 클래스로 생성되었는지에 대한 정보를 반환

```
class Car {
}

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

```
Car
```

equals() 메소드

- Object 클래스에서 제공하는 equals()
  - 연산자 ==를 사용하여 객체의 참조값이 동일한지를 검사하고 그 결과(true 또는 false)를 반환한다.
- 하지만 많은 클래스에서 객체가 동일한지 검사할 때 이와 다른 기준을 적용하기를 원한다.
  - equals()를 오버라이드하면 됨
- String 클래스 예



s1 == s2 ➔ false

s1.equals(s2) ➔ true

equals()를 오버라이드 하지 않은 경우

```
class Car {
    private String model;
    public Car(String model) {
        this.model= model;
    }
}


public class CarTest {
    public static void main(String[] args) {
        Car car1 = new Car("HMW520");
        Car car2 = new Car("HMW520");
        if (car1.equals(car2))
            System.out.println("동일한 종류의 자동차");
        else
            System.out.println("다른 종류의 자동차");
    }
}
```

다른 종류의 자동차

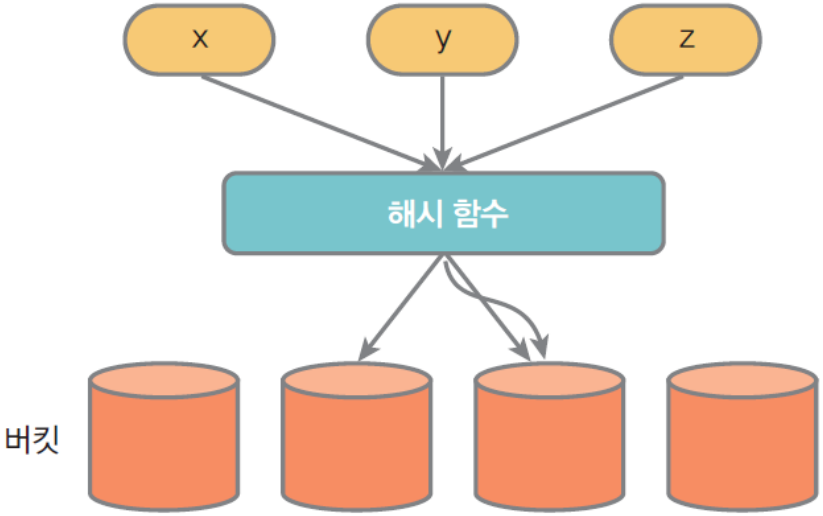
```
class Car {
    private String model;
    public Car(String model) {
        this.model= model;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Car)
            return model.equals(((Car) obj).model);
        else
            return false;
    }
}

public class CarTest {
    public static void main(String[] args) {
        Car car1 = new Car("HMW520");
        Car car2 = new Car("HMW520");
        if (car1.equals(car2))
            System.out.println("동일한 종류의 자동차");
        else
            System.out.println("다른 종류의 자동차");
    }
}
```

Object로부터 상속받은 equals() 메소드를 오버라이드

```
동일한 종류의 자동차
```

hashCode() 메소드

- hashCode()는 해싱(hashing)이라는 탐색 알고리즘에서 필요한 해시값을 생성하는 메소드
  - 두 객체가 동일하면 동일한 해시값을 가져야 한다.
  - 두 객체가 동일한 해시값을 가지면 객체가 동일할 수도 있고 아닐 수도 있다.(충돌 때문에)



이클립스의 자동코드생성 기능을 이용하여 equals()와 hashCode()를 오버라이드해보자.

toString() 메소드

- Object 클래스에서 제공하는 toString()은 다음과 같은 객체의 문자열 표현을 리턴

  getClass().getName() + '@' + Integer.toHexString(hashCode())

- 이와 달리 toString() 메소드가 객체의 필드값 정보를 리턴하도록 하려면 toString()를 오버라이드하면 된다.

toString()을 오버라이드 하지 않은 경우

```
class Car {
    private String model;
    public Car(String model) {
        this.model= model;
    }
}



public class CarTest {
    public static void main(String[] args) {
        Car car1 = new Car("HMW520");
        System.out.println(car1.toString());
    }
}
```

Car@1db9742

```
class Car {
    private String model;
    public Car(String model) {
        this.model= model;
    }
    @Override
    public String toString() {
         return "모델: " + model;
    }
}

public class CarTest {
    public static void main(String[] args) {
        Car car1 = new Car("HMW520");
        System.out.println(car1.toString());
    }
}
```

Object로부터 상속받은
toString() 메소드를 오버라이드

```
모델: HMW520
```

IS-A 관계

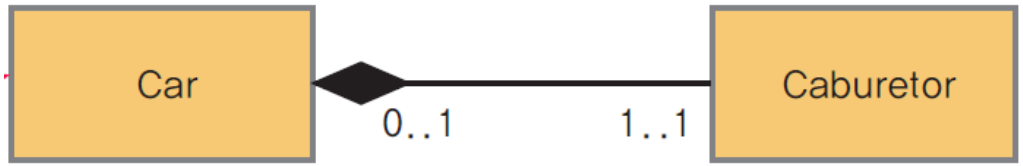
- is-a 관계
  - "~은 ~이다" 와 같은 관계
  - "~은 일종의 ~이다"
- 상속은 is-a 관계
- 예
  - 자동차는 탈것이다(Car is-a Vehicle).
  - 강아지는 동물이다(Dog is-a Animal).

HAS-A 관계

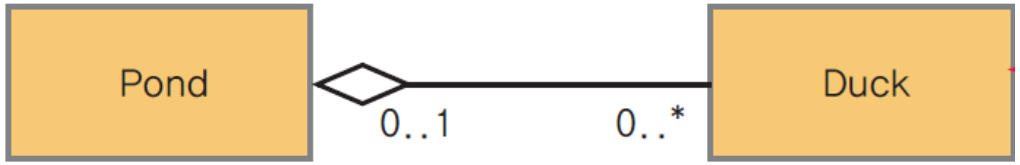
- has-a 관계
  - "~은 ~을 가지고 있다"와 같은 관계
- 예
  - 도서관은 책을 가지고 있다(Library has-a Book).
  - 거실은 소파를 가지고 있다(Livingroom has-a Sofa).

HAS-A 관계

- 객체 지향 프로그래밍에서 has-a 관계는 구성 관계(composition) 또는 집합 관계(aggregation)를 나타낸다.
  - 자동차를 카뷰레타를 가지고 있다(composition)



  - 연못은 오리를 가지고 있다(aggregation)

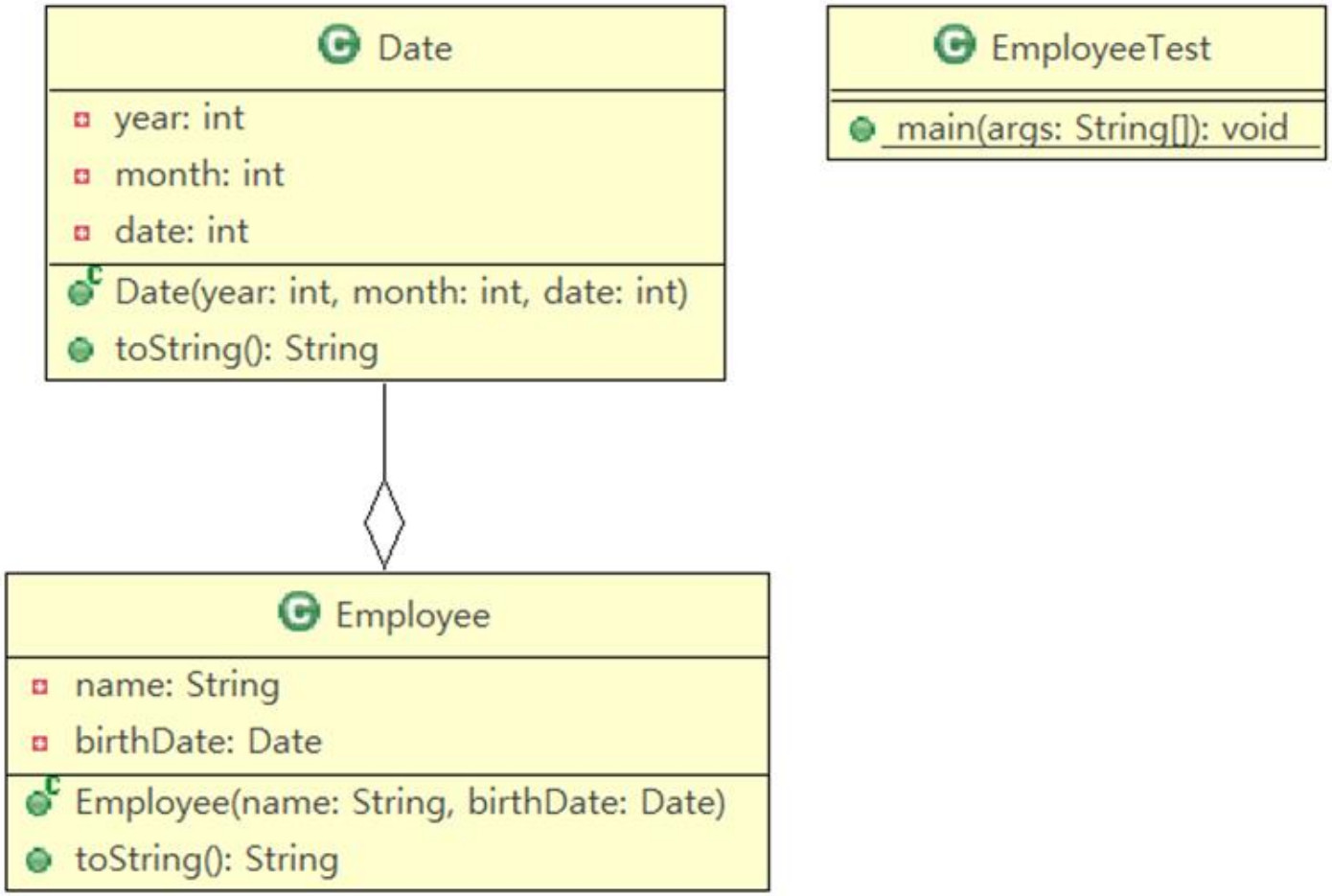


IS-A와 HAS-A 관계의 구현

- Car is-a Vehicle
- Car has-a Carburetor

```
class Vehicle {
}

class Carburetor {
}

class Car extends Vehicle{
        private Carburetor cb;
}
```

HAS-A 관계 예제


Date
year: int
month: int
date: int
Date(year: int, month: int, date: int)
toString(): String
EmployeeTest
main(args: String[]): void
Employee
name: String
birthDate: Date
Employee(name: String, birthDate: Date)
toString(): String

```java
public class Date {
    private int year;
    private int month;
    private int date;
    public Date(int year, int month, int date) {
        this.year = year;
        this.month = month;
        this.date = date;
    }
    @Override
    public String toString() {
        return year + "/" + month + "/" + date;
    }
}
```

```java
public class EmployeeTest {
    public static void main(String[] args) {
        Date birth = new Date(1990, 1, 2);
        Employee employee =
                new Employee("홍길동", birth);
        System.out.println(employee);
    }
}
```

```java
public class Employee {
    private String name;
    private Date birthDate;
    public Employee(String name, Date birthDate) {
        this.name = name;
        this.birthDate = birthDate;
    }
    @Override
    public String toString() {
        return name + " " + birthDate;
    }
}
```

```
홍길동 1990/1/2
```

종단 클래스(final class)

- final class는 자식 클래스를 정의할 수 없는 클래스
  - final class가 필요한 것은 주로 보안상의 이유 때문이다.
- final 키워드 사용

자식 클래스를 정의할 수 없도록 final로 선언

```
public final class MyFinal {
    …
}

public class ThisIsWrong extends MyFinal {      // 에러!
    …
}
```

종단 메소드(final method)

- final method는 자식 클래스에서 오버라이드할 수 없는 메소드
- final 키워드 사용

```
public class Baduk {
    enum BadukPlayer { WHITE, BLACK }
    …

    final BadukPlayer getFirstPlayer() {
        return BadukPlayer.BLACK;
    }
}
```

자식 클래스에서 재정의할 수 없도록 final로 선언

정적 바인딩

- 일반적으로 메소드 호출은 동적 바인딩(dynamic binding)이지만, 다음과 같은 예외가 있다.
  - 정적 메소드는 정적 바인딩(static binding)
  - 참고 : final 메소드, private 메소드도 정적 바인딩
    - 오버라이드할 수 없으므로 컴파일 시간에 결정 가능

```java
public class Animal {
    public static void eat() {
        System.out.println("Animal의 정적 메소드 eat()");
    }
    public void sound() {
        System.out.println("Animal의 인스턴스 메소드 sound()");
    }
}
```

```java
public class Cat extends Animal {
    public static void eat() {
        System.out.println("Cat의 정적 메소드 eat()");
    }
    public void sound() {
        System.out.println("Cat의 인스턴스 메소드 sound()");
    }

    public static void main(String[] args) {
        Animal animal = new Cat();
        Animal.eat();
        animal.eat();          // 정적 바인딩
        animal.sound();        // 동적 바인딩
    }
}
```

```
Animal의 정적 메소드 eat()
Animal의 정적 메소드 eat()
Cat의 인스턴스 메소드 sound()
```