Chapter 06 상속

6.1 상속

- 상속(Inheritance): 기존에 존재하는 클래스로부터 필드와 메서드를 이어받고, 필요한 기능을 추가할 수 있는 기법

상속의 형식

- 부모 클래스(슈퍼 클래스): 상속하는 클래스
- 자식 클래스(서브 클래스): 상속받는 클래스
- 키워드 extends를 사용한다.

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

public class ElectricCar extends Car{
    int battery;
    public void charge(int amount) {
        battery += amount;
    }
}
```

무엇이 상속되는가?

- 자식 클래스는 부모 클래스가 가지고 있는 모든 멤버들을 상속받고 자신이 필요한 멤버를 추가하기 때문에 항상 자식 클래스가 부모 클래스를 포함하게 된다.
- 상속을 나타낼 때 extends(확장)라는 용어를 사용하는 것도 이것 때문이다.

```
public class ElectricCarTest {
    public static void main(String args[]) {
        ElectricCar obj = new ElectricCar();
        obj.speed = 10; // 부모 클래스의 필드에 접근한다.
        obj.setSpeed(60); // 부모 클래스의 메소드에 접근한다.
        obj.charge(10); // 자체 메소드에 접근한다.
    }
}
```

왜 상속하는가?

1. 필드와 메소드에 대한 코드를 재사용할 수 있다.
2. 중복되는 코드를 줄일 수 있다.

```
public class Vehicle {
    int speed;
    int heading;

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public void turn(int angle) {
        heading = angle;
    }
}

class Car extends Vehicle {
    int price;
}

class Truck extends Vehicle {
    int payload;
}

class Bus extends Vehicle {
    int seat;
}
```

자바 상속의 특징

자바에서의 상속의 특징

1. 여러 클래스로부터 상속받는 다중 상속을 지원하지 않는다.
2. 상속의 횟수에 제한이 없다.
3. 상속 계층 구조의 최상위에는 java.lang.Object가 있다.

예제 6-1

```
class Animal {
    int age;
    void eat() {
        System.out.println("먹음");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("짖음");
    }
}

public class DogTest {
    public static void main(String args[]) {
```

```
        Dog obj = new Dog();
        obj.eat(); // 먹음
        obj.bark(); // 짖음
    }
}
```

예제 6-2

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

class Circle extends Shape {
    int radius;

    public Circle(int radius) {
        this.radius = radius;
        x = 0;
        y = 0;
    }

    double getArea() {
        return 3.14 * radius * radius;
    }
}

public class CircleTest {
    public static void main(String args[]) {
        Circle obj = new Circle(10);
        System.out.println("원의 중심: (" + obj.x + ", " + obj.y + ")"); //
원의 중심: (0, 0)
        System.out.println("원의 면적: " + obj.getArea()); // 원의 면적: 314.0
    }
}
```

6.2 상속과 접근 지정자

- 접근 지정자를 통해 자식 클래스가 상속받는 멤버를 지정한다.

	public	protect	default	private
동일한 클래스	O	O	O	O
동일한 패키지	O	O	O	X
자식 클래스	O	O	O	X
다른 패키지	O	X	X	X

```
class Shpae {
    protected int x, y;


    void print() {
        System.out.println("x좌펴: " + x + " y좌표: " + y);
    }
}


public class Rectangle extends Shape {
    int width, heigh;

    double calcArea() {
        return 3.14 * width * heigh;
    }

    void draw() {
        System.out.println("(" + x + ", " + y + ")" + " 위치에 " + "가로: " +
width + "세로: " + heigh);
    }
}
```

예제 6-3

```
class Person {
    // private String regnumber;
    private double weight;
    protected int age;
    String name;

    double getWeight() {
        return weight;
    }

    void setWeight(double weight) {
        this.weight = weight;
    }
}

class Student extends Person {
    int id;
}

public class StudentTest {
    public static void main(String args[]) {
        Student obj = new Student();

        // obj.regnumber = '123456 - 1234567'; // 오류
        // obj.weight = 70; // 오류
        obj.age = 23;
```

```
        obj.name = "Sewon Kim";
        obj.setWeight(70);

        System.out.println(obj.age + "세"); // 23세
        System.out.println(obj.name); // Sewon Kim
        System.out.println(obj.getWeight() + "kg"); // 70.0kg
    }
}
```

6.3 상속과 생성자

- 자식 클래스 객체 안에는 부모 클래스에서 상속된 부분이 들어 있기 때문에 자식 클래스 객체를 생성하면 부모 클래스의 생성자도 호출된다.
- 자식 클래스 객체는 부모 클래스에서 상속된 부분을 초기화하기 위하여 먼저 부모 클래스의 생성자를 호출하고 부모 클래스의 생성자 호출이 끝나면 자식 클래스가 추가한 부분을 초기화하기 위하여 자식 클래스의 생성자가 실행된다.

```
class Base {
    Base() {
        System.out.println("Base() 생성자 호출");
    }
};

class Derived extends Base {
    Derived() {
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
Base() 생성자 호출
Derived() 생성자 호출
 */
```

명시적인 호출

- 키워드 super를 이용해 자식 클래스의 생성자에서 명시적으로 부모 클래스의 생성자를 호출할 수 있다.

```
class Base {
    Base() {
        System.out.println("Base() 생성자 호출");
    }
};
```

```
class Derived extends Base {
    Derived() {
        super();
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
Base() 생성자 호출
Derived() 생성자 호출
 */
```

묵시적인 호출

- 자바에서는 명시적으로 부모 클래스의 생성자를 호출하지 않아도 자식 클래스의 객체가 생성될 때 자동으로 부모 클래스의 기본 생성자가 호출된다.

```
class Base {
    Base() {
        System.out.println("Base() 생성자 호출");
    }
};

class Derived extends Base {
    Derived() {
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
Base() 생성자 호출
Derived() 생성자 호출
 */
```

오류가 발생하는 경우

- 묵시적인 부모 클래스 생성자 호출을 사용하려면 부모 클래스에 매개변수가 없는 기본 생성자가 반드시 정의되어 있거나 생성자가 명시적으로 정의되지 않아야 한다.

```
// 기본 생성자가 정의된 경우
class Base {
    Base() {
        System.out.println("Base() 생성자 호출");
    }
};

class Derived extends Base {
    Derived() {
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
Base() 생성자 호출
Derived() 생성자 호출
 */
```

```
// 생성자가 명시적으로 정의되지 않아 기본 생성자가 자동으로 생성되는 경우
class Base {
};

class Derived extends Base {
    Derived() {
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
Derived() 생성자 호출
 */
```

```
// Base 클래스에는 이미 int 형의 인수를 가지는 생성자가 선언되어 있어서 컴파일러가 기본 생성자
를 만들지 않으므로 오류가 발생한다.
// 이를 해결하기 위해 키워드 super로 명시적으로 자식 클래스의 생성자 첫 부분에 부모 클래스의 생
성자를 호출하는 문장을 넣어야 한다.
class Base {
    Base(int x) {
```

```
        System.out.println("Base() 생성자 호출");
    }
};

class Derived extends Base {
    Derived() {
        // super(100);
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
error
 */
```

부모 클래스의 생성자 선택

- 부모 클래스에 여러 개의 생성자가 정의된 경우 키워드 super의 인자에 들어가는 매개변수의 수를 통해 부모 클래스의 생성자를 선택한다.

```
class Base {
    Base() {
        System.out.println("Base() 생성자 호출");
    }

    Base(int x) {
        System.out.println("Base(int x) 생성자 호출");
    }
};

class Derived extends Base {
    Derived() {
        super(100);
        System.out.println("Derived() 생성자 호출");
    }
};

public class Test {
    public static void main(String args[]) {
        Derived obj = new Derived();
    }
}
/*
Base(int x) 생성자 호출
Derived() 생성자 호출
*/
```

예제 6-4 Person 클래스와 Student 클래스 만들어보기

```
class Person {
    String name;
    public Person() { };
    public Person(String name) {
        this.name = name;
    }
}

class Employee extends Person {
    String id;
    public Employee() {
        super();
    }

    public Employee(String name) {
        super(name);
    }

    public Employee(String name, String id) {
        super(name);
        this.id = id;
    }
}

public class EmployeeTest {
    public static void main(String args[]) {
        Employee obj = new Employee("Sewon", "2022440025");

        System.out.println("Employ [id: " + obj.id + ", name: " + obj.name
+ "]"); // Employ [id: 2022440025, name: Sewon]
    }
}
```

6.4 메소드 오버라이딩

메소드 오버라이딩이란?

- 메소드 오버라이딩(method overriding): 자식 클래스가 부모 클래스의 메소드를 자신의 필요에 맞추어서 재정의하는 것. 이때 메소드의 이름이나 매개변수, 반환형은 동일해야 한다.

```
class Shpae {
    public void draw() { System.out.println("Shape을 그립니다."); };
}

class Circle extends Shpae {
    public void draw() {System.out.println("Circle을 그립니다."); };
}
```

```
class Rectangle extends Shape {
    public void draw() {System.out.println("Rectangle을 그립니다."); };
}

class Triangle extends Shape {
    public void draw() {System.out.println("Triangle을 그립니다."); };
}

public class ShapeTest {
    public static void main(String args[]) {
        Rectangle obj = new Rectangle();
        obj.draw(); // Rectangle을 그립니다.
    }
}
// Rectangle 클래스의 객체에 대하여 draw()가 호출되면 Shape의 draw()가 아니라
Rectangle 클래스 안에서 오버라이딩된 draw()가 호출된다.
```

- @Override: @Override 이노테이션은 부모 클래스에 있는 메소드와 일치하는 메소드가 없다면 컴파일 오류를 발생시킨다.

```
class Shpae {
    public void draw(String color) { System.out.println("Shape을 그립니다.");
};
}

class Circle extends Shpae {
    @Override
    public void draw() {System.out.println("Circle을 그립니다."); };
}

public class ShapeTest {
    public static void main(String args[]) {
        Rectangle obj = new Rectangle();
        obj.draw();
    }
}

// method does not override or implement a method from a supertype
```

키워드 super를 사용하여 부모 클래스 멤버 접근

- 부모 클래스의 메소드를 오버라이딩한 경우에 super를 사용하면 자식 클래스의 오버라이딩 메소드에서 부모 클래스의 메소드를 호출할 수 있다.
- 보통 메소드를 오버라이딩할 때 부모 클래스의 메소드를 완전히 대체하기보다 내용을 추가하는 경우가 많으므로 super 키워드를 이용해 부모 클래스의 메소드를 호출한 후 자신이 필요한 부분을 추가하는 것이 좋다.

```
class Shpae {
    public void draw() { System.out.println("Shape 중 하나를 그릴 예정입니다.");
```

```
};
}

class Circle extends Shpae {
    @Override
    public void draw() {
        super.draw();
        System.out.println("Circle을 그립니다.");
    };
}

public class ShapeTest {
    public static void main(String args[]) {
        Circle obj = new Circle();
        obj.draw();
    }
}
/*
Shape 중 하나를 그릴 예정입니다.
Circle을 그립니다.
 */
```

오버라이딩 vs 오버로딩

- 오버로딩(overloading): 한 클래스에서 같은 메소드명을 가진 여러 개의 메소드를 작성하는 것
- 오버라이딩(overriding): 부모 클래스의 메소드를 자식 클래스에서 재정의하는 것
- 모두 이름을 재사용하는 것이다.
- 모두 다형성과 관련이 있다.
- 오버로딩은 컴파일 시간에서의 다형성을 지원하고 메소드 오버라이딩은 실행 시간에서의 다형성을 지원한다.

정적 메소드를 오버라이드하면 어떻게 될까?

- 동일한 시그니처(이름, 반환형, 매개변수의 수)를 가지는 정적 메소드가 오버라이드되면 호출하는 객체의 참조 변수에 따라 호출되는 메소드가 결정된다.

```
class Animal {
    public static void A() {
        System.out.println("static method in Animal");
    }
}

public class Dog {
    // @Override
    public static void A() {
        System.out.println("static method in dog");
    }

    public static void main(String args[]) {
```

```
        Dog obj = new Dog();
        Animal obj2 = new Animal();

        // Dog obj3 = new Animal(); (X)
        // Animal obj4 = new Dog(); (X)

        obj.A(); // static method in dog
        obj2.A(); // static method in Animal
    }
}
```

예제 6-5 Employee 클래스

```
class Employee {
    public int baseSalary = 3000000;
    public int getSalary() {
        return baseSalary;
    }
}

class Manager extends Employee{
    @Override
    public int getSalary() {
        return (baseSalary + 2000000);
    }
}

class Programmer extends Employee{
    @Override
    public int getSalary() {
        return (baseSalary + 3000000);
    }
}

public class Salary {
    public static void main(String args[]) {
        Employee obj0 = new Employee();
        Manager obj1 = new Manager();
        Programmer obj2 = new Programmer();
        Employee obj4 = new Manager();

        System.out.println("직원의 월급: " + obj0.getSalary()); // 직원의 월급:
3000000
        System.out.println("관리자의 월급: " +obj1.getSalary()); // 관리자의 월
급: 5000000
        System.out.println("프로그래머의 월급: " + obj2.getSalary()); // 프로그래
머의 월급: 6000000
        System.out.println(obj4.getSalary()); // 5000000
    }
}
```

6.5 다형성

- 다형성(polymorphism: poly 많은 morph 모양): 객체들이 동일한 메시지를 받더라도 각자의 실제 타입에 따라서 서로 다른 동작을 하는 것

업캐스팅

- 업캐스팅(upcasting, 상형 형변환): 부모 클래스 변수로 자식 클래스 객체를 참조하는 것
- 업캐스팅을 했을 때 자식 클래스 중에서 부모 클래스 중에서 부모 클래스로부터 상속받은 부분만을 접근할 수 있다.
- 즉 객체의 타입이 아니라 변수의 타입에 의해 접근할 수 있는 멤버가 결정된다.

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

class Rectangle extends Shape{
    int width, height;
    @Override
    public void draw() {
        System.out.println("Reactangle draw");
    }
}

class Triangle extends Shape {
    int base, height;
    @Override
    public void draw() {
        System.out.println("Triangle draw");
    }
}

class Circle extends Shape {
    int radius;
    @Override
    public void draw() {
        System.out.println("Circle draw");
    }
}

public class ShapeTesting extends Shape {
    public static void main(String args[]) {
        Shape obj1;
        obj1 = new Shape();
        Shape obj2 = new Rectangle();

        obj1.x = 10;
        obj1.y = 20;

        obj1.width = 30;
```

```
        obj1.height = 40; // error: cannot find symbol
    }
}
```

업캐스팅 vs 다운캐스팅

업캐스팅	다운캐스팅
부모 참조 변수로 자식 참조 변수를 참조하는 것	자식 참조 변수로 부모 객체를 참조하는 것
묵시적으로 수행될 수 있다.	명시적으로 수행되어야 한다.

자식 클래스의 멤버에 접근할 수 없고 부모 클래스의 멤버에 접근할 수 있다.

```
class Parent {
    public void print() {
        System.out.println("Parent overrided method call");
    }

    public void noprint() {
        System.out.println("Parent no overrided method call");
    }
}

class Child extends Parent {
    @Override
    public void print() {
        System.out.println("Child overriding method call");
    }
}

public class Casting {
    public static void main(String args[]) {
        Parent obj1 = new Parent();
        obj1.print(); // Parent overrided method call
        obj1.noprint(); // Parent no overrided method call

        Child obj2 = new Child();
        obj2.print(); // Child overriding method call
        obj2.noprint(); // Parent no overrided method call

        Parent obj3 = new Child();
        obj3.noprint(); // Parent no overrided method call
        obj3.print(); // Child overriding method call

        Child obj4 = (Child)obj3; // 다운캐스팅: 부모 객체를 자식 객체로 형변환
        obj4.print(); // Child overriding method call
        obj4.noprint(); // Parent no overrided method call

        Child obj5 = new Parent(); // error: incompatible types: Parent
cannot be converted to Child
```

```
    }
}
```

동적 바인딩

- 부모 참조 변수를 가지고 자식 객체를 참조하는 것이 어디에 필요할까?
- 여러 가지 객체를 하나의 자료 구조 안에 모아서 처리하는 경우에 필요하다.

```
class Shpae {
    public void draw() { System.out.println("Shape 중 하나를 그릴 예정입니다.");
};
}

class Rectangle extends Shape{
    @Override
    public void draw() { System.out.println("Rectangle을 그립니다."); }
}

class Triangle extends Shape {
    @Override
    public void draw() { System.out.println("Triangle을 그립니다."); }
}

public class ShapeTest {
    public static void main(String args[]) {
        Shape [] arrayOfShapes;
        arrayOfShapes = new Shape[3];

        arrayOfShapes[0] = new Rectangle();
        arrayOfShapes[1] = new Triangle();

        for (int i = 0; i < arrayOfShapes.length; i++ ) {
            arrayOfShapes[i].draw();
        }
    }
}

/*
Rectangle을 그립니다.
Triangle을 그립니다.
-> 객체의 타입은 Shape이지만 객체가 가리키고 있는 객체의 타입이 Rectangle이기 때문에 오버라이
딩의 경우에 자식 클래스의 메소드가 호출된다.
 */
```

- 바인딩(binding): 메소드 호출을 실제 메소드의 몸체와 연결하는 것
- C 언어에서는 컴파일 단계에서 모든 바인딩이 완료되지만 Java에서는 바인딩이 실행 시까지 연기된다.
- 동적 바인딩(dynamic binding): 자바 가상 머신(JVM)이 실행 단계에서 변수가 참조하는 객체의 실제 타입을 보고 적절한 메소드를 호출하는 것
- 동적 바인딩: 오버라이드된 메소드 호출이 컴파일 시간이 아닌 실행 시간에 결정되는 메커니즘

- 오버라이드된 메소드가 부모 클래스 참조를 통하여 호출되는 경우에 객체의 타입에 따라서 서로 다른 메소드가 호출되는 메커니즘
- 객체의 실제 타입이 호출되는 메소드를 결정한다.

동적 바인딩의 장점

- 다형성을 사용하면 시스템에 최소한의 영향을 미치면서 새로운 유형의 객체를 쉽게 추가하여 시스템을 확장할 수 있다.
- 자바는 오버로드된 메소드에 대해 정적 바인딩을 사용하고 오버라이드된 메소드에 대해 동적 바인딩을 사용한다. 즉 오버로드된 메소드는 컴파일할 때 결정되지만 오버라이드된 메소드는 실행 시간까지 바인딩이 연기되었다가 실행 시간에 실제 타입을 보고 어떤 메소드를 호출할 것인지 결정한다.

업캐스팅의 활용

- 업캐스팅은 동적 바인딩의 용도로 많이 사용하지만 메소드의 매개 변수를 선언할 때도 이용한다.
- 메소드의 매개 변수를 부모 타입으로 선언하면 부모 클래스에서 파생된 모든 타입의 객체를 받을 수 있다.

instanceof 연산자

- 자바에서는 동적 바인딩으로 인해 변수의 타입만으로는 변수가 가리키는 실제 타입을 알 수 없다.
- instanceof 연산자: 변수가 가리키는 객체의 실제 타입을 알려준다.

종단 클래스와 종단 메소드

- 종단 클래스(final class): 상속시킬 수 없는 클래스
- 자바에서는 이론상 중요한 클래스의 서브 클래스를 만들어 서브 클래스로 하여금 시스템을 파괴할 수 있도록 하기 때문에 자바 시스템은 중요한 클래스에 대하여 종단 클래스를 선언하여 보안을 유지한다.
- 키워드 final: 클래스 앞에 붙인다.
- 종단 메소드(final method): 종단 클래스가 아닌 일반 클래스에서 재정의될 수 없는 특별한 메소드

6.6 상속 vs 구성

상속	구성
한 객체가 클래스를 상속받아서 부모 객체의 속성과 동작을 획득하는 기법	클래스가 다른 클래스의 인스턴스를 클래스의 필드로 가지는 기법
한 클래스에서 다른 클래스를 파생시킨다.	하나의 클래스를 다른 클래스의 합으로 정의
IS-A 관계	HAS-A 관계
상속에서는 하나의 클래스만을 상속할 수 있으므로 하나의 클래스에서만 코드를 재사용할 수 있다.	여러 클래스에서 코드를 재사용할 수 있다.
상속은 컴파일 시간에 결정된다.	구성은 실행 시간에 결정될 수 있다.
final로 선언된 클래스의 코드를 재사용할 수 없다.	final로 선언된 클래스의 코드도 재사용할 수 있다.
부모 클래스의 public 및 protected 메소드를 모두 노출한다.	아무것도 노출하지 않고 공개 인터페이스만을 사용하여 상호작용한다.

is-a 관계

- is-a 관계: A is a B라 할 때 A는 B의 일종이다.
- is-a 관계가 있으면 상속을 한다. 상속 계층 구조를 올바르게 설계하였는지 판단하려면 is-a 관계가 성립하는지 생각해 보면 된다.

has-a 관계

- has-a 관계: A has a B라 할 때 A는 B를 가지고 있다.
- has-a 관계가 있으면 상속을 하면 안 되고 하나의 클래스 안에 다른 하나의 클래스를 포함하는 구성을 해야 한다.
- 구성: 자동차는 엔진으로 구성된다.
- 집합: 연못은 오리를 소유한다.

```
class Vehicle {}
class Engine {}
class Brake {}

public class Car extends Vehicle {
    private Engine obj1;
    private Brake obj2;

    public Car() {
        obj1 = new Engine();
        this.obj2 = new Brake();
    }
}
```

구성 vs 집합

잘못 사용하는 경우

상속 vs 구성