Chapter 07 추상 클래스, 인터페이스, 중첩 클래스

7.1 추상 클래스

- Abstract Class(추상 클래스): 완전하게 구현되지 않은 메소드를 가진 클래스
- abstract 키워드를 사용한다.
- 추상 메소드: 몸체가 없는 메소드
- 추상 메소드는 세미콜론(;)로 종료되어야 한다.
- 추상 클래스는 하나 이상의 추상 메소드를 가져야 하고, 추상 클래스를 상속받는 자식 클래스에서는 모든 추상 메소드를 재정의해야 한다.

예제 7-1

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

    public abstract void draw();
}

class Rectangle extends Shape{
    int width, height;
    public void draw() {
        System.out.println("직사각형 그리기 메소드");
    }
}

class Circle extends Shape {
    int radius;
    public void draw() {
        System.out.println("원 그리기 메소드");
    }
}

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

        // Shape shape = new shape(); // 컴파일 오류
        Shape circle = new Circle();

        circle.draw();
    }
}
```

추상 클래스의 용도

- 부모 클래스에서 void draw() {}와 같이 내용이 없는 일반 메소드로 정의하는 것은 자식 클래스에서 오버라이드하지 않아도 컴파일 에러가 발생하지 않지만, 추상 클래스로 정의하면 서브 클래스에서 반드시 구현해야 하므로 구현을 강요하는 면에서 장점이 있다.

7.2 인터페이스

인터페이스의 용도

- 추상화: 인터페이스는 메소드 몸체(세부 구현)가 없는 메소드 시그니처만 저장한다. 메소드 시그니처만 공개하는 것은 사용자에게 메소드 구현을 숨김으로써 추상화를 달성한다. 추상화는 객체 지향 프로그래밍 기술에서 중요한 개념이다.
- 다중 상속: 기존 방식은 심각한 모호성 문제(다이아몬드 문제) 때문에 다중 클래스 상속이 불가능하다. 인터페이스는 필드의 정의가 금지되므로 이러한 문제를 해결할 수 있다.
- 느슨한 결합: 결합(커플링)이란 하나의 클래스가 다른 클래스에 종속되는 정도를 말한다. 인터페이스를 사영하면 메소드와 메소드 시그니처를 따로 정의할 수 있어서 클래스들이 완전히 독립적인 상태에서 느슨한 결합을 이룰 수 있다.

인터페이스의 필요성과 예

인터페이스 정의

- 키워드 interface를 이용하여 인터페이스를 정의한다.
- 인터페이스는 추상 메소드들과 디폴트 메소드로 이루어진다.
- 인터페이스 안에서 필드(변수)는 선언될 수 없고 상수는 정의될 수 있다.
- 인터페이스 안에서 선언되는 메소드는 묵시적으로 public abstract이기 때문에 별도의 수식어를 사용하지 않아도 된다.

인터페이스 구현

- 인터페이스 안에는 구현되지 않은 메소드가 존재하기 때문에 인터페이스만으로 객체를 생성할 수 없고, 다른 클래스에 의하여 구현될 수 있다.
- 인터페이스를 구현한다는 것은 인터페이스에 정의된 추상 메소드의 몸체를 정의한다는 것이다.
- 클래스가 인터페이스를 구현하기 위해서는 키워드 implement를 사용한다.

인터페이스 vs 추상 클래스

- 인터페이스와 추상 클래스는 객체화될 수 없다.
- 주로 구현이 안 된 메소드들로 이루어진다.
- 추상 클래스에서는 인터페이스와 달리 일반적인 필드도 선언할 수 있으며 일반적인 메소드도 정의할 수 있다.
- 인터페이스에서 추상 클래스와 달리 모든 메소드는 public, abstract으로 선언되며 여러 개의 인터페이스갸 동시에 구현될 수 있다.

추상 클래스를 사용하는 경우

- 관련된 클래스 사이에서 코드를 공유하고 싶은 경우
- 공통적인 필드나 메소드의 수가 많은 경우, 또는 public 이외의 접근 지정자를 사용해야 하는 경우
- 정적이 아닌 필드나 상수가 아닌 필드를 선언하는 경우

인터페이스를 사용하는 경우

- 관련 없는 클래스에서 동일한 동작을 하는 경우
- 누가 구현하는지 신경쓸 필요가 없는 경우

- 다중 상속을 하는 경우

인터페이스와 타입

- 인터페이스 자료형처럼 사용하여 이 인터페이스를 구현한 참조 변수를 정의할 수 있다.

예제 7-2

```
interface RemoteControl {
    public abstract void turnOn();
    public abstract void turnOff();
    public default void printBrand() {
        System.out.println("Remote Control TV");
    }
}

class Television implements RemoteControl {
    boolean on;
    public void turnOn() {
        on = true;
        System.out.println("TV turn on");
    }

    public void turnOff() {
        on = false;
        System.out.println("TV turn off");
    }

    @Override
    public void printBrand() {
        System.out.println("PowerJava TV");
    }
}

public class TestInterface {
    public static void main(String[] args) {
        RemoteControl rc = new Television();
        rc.turnOff();
        rc.turnOff();
        rc.printBrand();
    }
}
```

7.3 인터페이스를 이용한 다중 상속

인터페이스끼리도 상속이 가능하다

- 다른 프로그래머가 사용하고 있던 인터페이스를 변경하면 이 인터페이스를 구현하는 모든 클래스가 동작하지 않게 된다.
- 이 경우를 대비하여 인터페이스도 상속을 받아서 확장할 수 있다.
- we can inherit interface using keyword extends.

인터페이스를 이용한 다중 상속

- Multiple inheritance(다중 상속): 하나의 클래스가 여러 개의 부모 클래스를 가지는 것
- 다이아몬드 문제
- 하나의 클래스를 상속하는 동시에 여러 인터페이스를 구현하면 다중 상속과 비슷한 효과를 가질 수 있다.

방법1

```
interface Drivable {
    public abstract void drive();
}

interface Flyable {
    public abstract void fly();
}

public class FlyingCar1 implements Drivable, Flyable {
    public void drive() {
        System.out.println("I'm driving");
    }

    public void fly() {
        System.out.println("I'm flying");
    }

    public static void main(String[] args) {
        FlyingCar1 obj = new FlyingCar1();

        obj.drive();
        obj.fly();
    }
}
```

방법2

```
interface Flyalbe {
    public abstract void fly();
}

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

public class FlyingCar2 extends Car implements Flyable{
    public void fly() {
        System.out.println("I'm flying");
    }
```

```
    public static void main(String[] args) {
        FlyingCar2 obj = new FlyingCar2();
        obj.setSpeed(100);
        obj.fly();
    }
}
```

상수 정의

- 인터페이스에서 정의된 변수는 자동으로 public, static, final이 되어 상수가 된다.

예제 7-3

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

interface Drawable {
    public abstract void draw(int x, int y);
}

class Circle1 extends Shape implements Drawable {
    int radius;
    public void draw(int x, int y) {
        System.out.println("Circle draw at (" + x + ", " + y + ")");
    }
}

public class TestInterface2 {
    public static void main(String[] args) {
        Circle1 circle = new Circle1();

        circle.draw(0, 0);
    }
}
```

7.4 디폴트 메소드와 정적 메소드

디폴트 메소드

- Default method: interface를 구현하는 class가 method의 몸체를 구현하지 않아도 되도록 하기 위해 Interface 개발자가 method의 default 구현하는 기능

```
interface MyInterface {
    public abstract void myMethod1();
    public default void myMethod2() {
        System.out.println("myMethod2");
    }
```

```
}

public class MyClass implements MyInterface {
    public void myMethod1() {
        System.out.println("myMethod1");
    }

    public static void main(String[] args) {
        MyClass myclass = new MyClass();
        myclass.myMethod1();
        myclass.myMethod2();
    }
}
```

예제 7-4

```
interface Drawable {
    public abstract void draw();
    public default void getSize() {
        System.out.println("1024x768 resolution");
    }
}

class Circle implements Drawable {
    public void draw() {
        System.out.println("Circle draw");
    }

    @Override
    public void getSize() {
        System.out.println("3000x2000 resolution");
    }
}

public class TestClass {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.draw();
        circle.getSize();
    }
}
```

정적 메소드

- JDK8부터 Interface에 static method를 추가할 수 있다.
- Factory method(팩토리 메소드): new를 호출하여 객체를 생성하는 코드를 부모 클래스에 위임하는 것. 팩토리 메소드를 사용하는 이유는 하나의 클래스가 변경되었을 경우에 다른 클래스의 변경을 최소화하기 위해서다.

예제 7-5

```
import javax.management.RuntimeErrorException;

interface Employable {
    public abstract String getName();

    static boolean isEmpty(String str) {
        if (str == null || str.trim().length() == 0) {
            return true;
        } else {
            return false;
        }
    }
}

class Employee implements Employable {
    private String name;

    public Employee(String name) {
        if (Employable.isEmpty(name) == true)
            throw new RuntimeException("이름을 반드시 입력하여야 함!");
        this.name = name;
    }

    @Override
    public String getName() {
        return this.name;
    }
}

public class StaticMethodTest2 {
    public static void main(String [] args) {
        Employee employee = new Employee("홍길동");
    }
}
```

7.5 중첩 클래스

중첩 클래스의 종류

내부 클래스

지역 클래스

중첩 클래스를 사용하는 이유

7.6 익명 클래스