

클래스, 객체

추상클래스, 추상메소드, 인터페이스, 내포클래스,
익명내부클래스, ActionListener

클래스 작성: 추상클래스, 추상메소드



실체 클래스 vs. 추상클래스

- 실체클래스(Concrete class)
 - ◆ 객체화 가능한 클래스
- 추상클래스(Abstract class)
 - ◆ Abstract 키워드가 부착된 클래스로 객체화될 수 없는 클래스
 - ◆ 추상메소드를 포함할 수 있음
 - 추상메소드는 그 구현 부분이 누락된 메소드 (아래 setColor() 참조)
 - 메소드 구현이 기술되는 { ... } 부분이 없으며 마지막에 세미콜론으로 종료
 - ◆ 자식 클래스로의 상속은 가능

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

```
public class Test {  
    public static void main(String[] args) {  
        Person person=new Person("S123", "Obama");  
    }  
}  
new 연산자를 통해 Person 객체 생성 가능
```

```
public abstract class Colorable {  
    abstract void setColor(int color);  
}
```

추상클래스
(abstract class)

추상메소드
(abstract method)

```
public class Test {  
    public static void main(String[] args) {  
        Colorable colorable=new Colorable(); // 오류 !  
    }  
}  
new 연산자를 통해 Colorable 객체 생성 불가  
setColor() 메소드 구현 부분이 존재하지 않음
```

클래스 작성: 추상클래스, 인터페이스, final



추상클래스 vs. 인터페이스

- 추상클래스와 인터페이스 모두 new 연산자를 통해 객체화될 수 없음
- 추상클래스
 - ◆ abstract 키워드가 부착된 것과 추상메소드를 포함할 수 있다는 것을 제외하면 일반 실체클래스와 크게 다르지 않음
 - 즉 속성 필드 및 실체메소드 등도 정의 가능
 - ◆ 클래스에서와 마찬가지로 추상클래스도 다중 상속이 허용되지 않음
- 인터페이스 (참조: <https://docs.oracle.com/javase/tutorial/java/landl/interfaceDef.html>)
 - ◆ class 키워드 대신 interface 키워드로 정의됨
 - ◆ 추상메소드, default 메소드, static 메소드 및 상수 필드 포함 가능
 - ◆ 모든 메소드는 public임(생략 가능), 모든 상수필드는 public static final임(생략 가능)
 - ◆ 다중 상속이 허용됨

```
public abstract class Colorable {  
    abstract void setColor(int color);  
}
```

```
public class Car extends Colorable {  
    int color;  
    @Override  
    void setColor(int color) {  
        this.color=color;  
    }  
}
```

```
public interface Colorable {  
    public static final int RED=1;  
    public static final int GREEN=2;  
    public static final int BLUE=3;  
    void setColor(int color);  
}
```

```
public class Car implements Colorable {  
    int color;  
    @Override  
    public void setColor(int color) {  
        this.color=color;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Car car=new Car();  
        car.setColor(1);  
    }  
}
```

클래스 작성: 인터페이스의 다중 상속

인터페이스는 다중 상속 가능

- 다중 상속된 인터페이스 내 동일 명칭 필드는 static이므로 객체와 무관 (예: Colorable.RED, Paintable.RED)

클래스는 단일 상속만 허용됨

- 다중 상속이 허용된다면 서로 다른 부모 클래스 내 동일 명칭 필드 접근 시 문제 발생

```
public interface Colorable {  
    static final int RED=1;  
    static final int GREEN=2;  
    static final int BLUE=3;  
    void setColor(int color);  
}
```

```
public interface Movable {  
    void moveTo(int x, int y);  
    void moveBy(int dx, int dy);  
}
```

```
public class Car implements Colorable, Movable {  
    int x, y, color;  
    @Override  
    public void moveTo(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
    @Override  
    public void moveBy(int dx, int dy) {  
        this.x+=dx;  
        this.y+=dy;  
    }  
    @Override  
    public void setColor(int color) {  
        this.color=color;  
    }  
    @Override  
    public String toString() {  
        return "Color: "+color+", Position="+x+", "+y+"";  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Car car=new Car();  
        car.setColor(Colorable.RED);  
        car.moveTo(100, 100);  
        car.moveBy(50, 70);  
        System.out.println(car);  
    }  
}
```

클래스 작성: 추상클래스, 인터페이스의 상속

- 추상클래스나 인터페이스를 상속 받는 자식 클래스를 실제 클래스로 정의할 때
 - 추상클래스나 인터페이스를 상속 받는 자식 클래스에서는 추상 클래스나 인터페이스에 정의된 모든 추상 메소드를 구현해야 한다
 - 구현할 내용이 없다면 메소드 본체를 빈 블록으로 남겨두면 됨 (참조: 아래 예의 moveBy() 참조)
 - 만약 아래 예에서 moveBy()의 구현 내용이 없다고 이 메소드를 삭제한다면 오류 발생

```
public interface Colorable {  
    static final int RED=1;  
    static final int GREEN=2;  
    static final int BLUE=3;  
    void setColor(int color);  
}
```

```
public interface Movable {  
    void moveTo(int x, int y);  
    void moveBy(int dx, int dy);  
}
```

```
public class Car implements Colorable, Movable {  
    int x, y, color;  
    @Override  
    public void moveTo(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
    @Override  
    public void moveBy(int dx, int dy) {  
    }  
    @Override  
    public void setColor(int color) {  
        this.color=color;  
    }  
    @Override  
    public String toString() {  
        return "Color: "+color+", Position=("+x+", "+y+")";  
    }  
}
```

클래스 상속, 인터페이스 구현

✚ class B extends A { ... }

- 기존 클래스 A를 확장하는(extends) 새로운 클래스 B를 정의

✚ class C implements D { ... }

- 기존 인터페이스 D의 추상 메소드를 구현(implements)하는 새로운 클래스 C를 정의

✚ class E extends F implements G, H { ... }

- 기존 클래스 F를 확장하면서, 동시에 기존 두 인터페이스 G, H의 추상메소드들을 구현하는 새로운 클래스 E를 정의

✚ interface P extends Q, R { ... }

- 기존 인터페이스 Q, R을 확장하면서 새로운 인터페이스 P를 정의

클래스 작성: 내포 클래스(nested class)

```
public class Employee {
```

```
    class Date {  
        int year, month, day;  
    }
```

```
    String id;  
    Date startDate; // 근무시작일
```

```
    public Employee(String id) {  
        this.id=id;  
        this.startDate=new Date();  
    }  
    public void setStartDate(int year, int month, int day) {  
        this.startDate.year=year;  
        this.startDate.month=month;  
        this.startDate.day=day;  
    }  
}
```

내포클래스 (nested class)

- 다른 클래스 내부에 정의된 클래스
- Static nested class vs. non-static nested class

내부클래스 (inner class)

- 다른 클래스 내부에 정의된 non-static 클래스

로컬클래스 (local class)

- 내부클래스 중 블록(예: 메소드) 내부에 정의된 클래스

익명클래스 (anonymous class)

- 내부클래스 중 블록(예: 메소드) 내부에 정의된 이름 없는 클래스

```
public class Test {  
    public static void main(String[] args) {  
        Employee e=new Employee("E-001");  
        e.setStartDate(2019,4,10);  
        System.out.println("근무시작년도: "+e.startDate.year);  
    }  
}
```

익명내부클래스를 활용한 코딩

익명내부클래스(Anonymous Inner Class)

- 자식 클래스 정의와 그 자식 클래스의 객체 생성을 동시에 작성하는 코드에 사용된 그 자식 클래스

```
public class Animal {  
    String speak(){  
        return "";  
    }  
}
```

```
public class Dog extends Animal {  
    @Override  
    String speak() {  
        return "멍멍";  
    }  
}
```

자식클래스 정의

```
public class Test {  
    public static void main(String[] args) {  
        Dog dog=new Dog(); // 객체 생성  
        System.out.println(dog.speak());  
    }  
}
```

실습

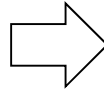
- 아래 코드는 야옹을 출력한다. 익명내부클래스를 이용하여 이 코드를 완성하시오

```
public class Test {  
    public static void main(String[] args) {  
        Animal cat=  
        System.out.println(cat.speak());  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Animal dog=new Animal(){  
            @Override  
            String speak() {  
                return "멍멍";  
            }  
        }; // 자식클래스 정의와 객체생성을 한번에  
        System.out.println(dog.speak());  
    }  
}
```


익명내부클래스를 활용한 코딩

```
public interface ActionListener extends EventListener {  
    public void actionPerformed(ActionEvent e);  
}
```



```
public class MyActionListener implements ActionListener {  
    @Override  
    public void actionPerformed(ActionEvent arg0) {  
        JOptionPane.showMessageDialog(null, "부모님, 사랑합니다.");  
    }  
}
```

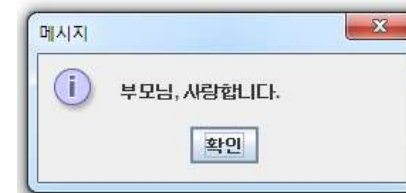
자식클래스 정의

```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        JButton button=new JButton("Click me");  
        ActionListener actionListener=new MyActionListener();  
        button.addActionListener(actionListener);  
        w.add(button);  
        w.setVisible(true);  
    }  
}
```

객체생성

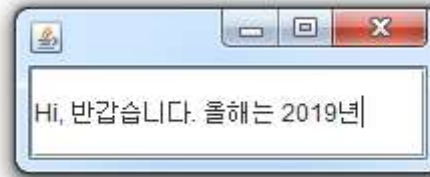
```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        JButton button=new JButton("Click me");  
        ActionListener actionListener=new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                JOptionPane.showMessageDialog(null, "부모님, 사랑합니다.");  
            }  
        };  
        button.addActionListener(actionListener);  
        w.add(button);  
        w.setVisible(true);  
    }  
}
```

인터페이스 ActionListener의 자식클래스를 명시적으로 정의하는 대신, 익명내부클래스를 활용하여 자식클래스정의와 객체생성을 한꺼번에 작성하였음

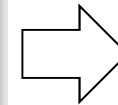


익명내부클래스를 활용한 코딩

```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        JTextField textField=new JTextField();  
        w.add(textField);  
        w.pack();  
        w.setVisible(true);  
    }  
}
```



JTextField 객체를 통해
한글, 영문, 숫자 등 입력 가능



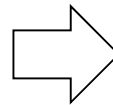
숫자만 입력 가능하도록
JTextField 변경

```
public class NumberTextField extends JTextField {  
    @Override  
    public void replaceSelection(String content) {  
        super.replaceSelection(content.replaceAll("[^0-9]", ""));  
    }  
}
```

자식클래스 정의

```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        NumberTextField textField=new NumberTextField();  
        w.add(textField);  
        w.pack();  
        w.setVisible(true);  
    }  
}
```

객체 생성



```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        JTextField textField=new JTextField() {  
            @Override  
            public void replaceSelection(String v) {  
                super.replaceSelection(v.replaceAll("[^0-9]", ""));  
            }  
        };  
        w.add(textField);  
        w.pack();  
        w.setVisible(true);  
    }  
}
```

익명내부클래스 객체 생성

인터페이스 활용 예: ActionListener

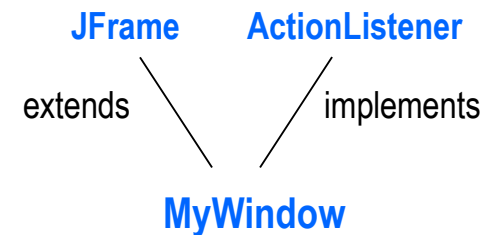
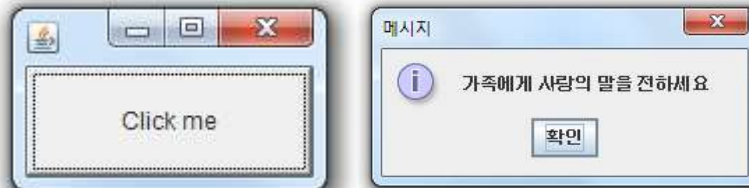
```
public class MyWindow extends JFrame implements ActionListener {
    public MyWindow() {
        JButton button=new JButton("Click me");
        add(button);
        button.addActionListener(this);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "가족에게 사랑의 말을 전하세요");
    }
    public static void main(String args[]){
        MyWindow w=new MyWindow();
        w.setVisible(true);
    }
}
```

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

MyWindow

JFrame

ActionListener



MyWindow는 JFrame이다
MyWindow는 ActionListener이다

추상 클래스 실습 A

- + 추상클래스 Shape(도형)을 다음 조건에 따라 정의하시오
 - 추상메소드 double getArea(): 도형의 면적 반환
- + 클래스 Rectangle(직사각형)을 추상클래스 Shape을 상속받아 다음 조건에 따라 정의하시오
 - 필드: width(가로 길이,정수), height(세로 길이,정수)
 - width, height 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 직사각형의 면적을 실수로 반환하는 메소드 getArea()을 Shape의 getArea()를 오버라이드하여 작성
- + 클래스 Test의 main() 내에서 다음 절차를 코딩하시오
 - Rectangle 객체(가로 3, 세로 4) 생성 후 getArea() 호출하여 Rectangle 객체의 면적 출력

추상 클래스 실습 B

- ✚ 추상클래스 Shape(도형)을 다음 조건에 따라 정의하시오
 - 추상메소드 `double getArea()`: 도형의 면적 반환
- ✚ 클래스 Rectangle(직사각형)을 추상클래스 Shape을 상속받아 다음 조건에 따라 정의하시오
 - 필드: `width`(가로 길이, 정수), `height`(세로 길이, 정수)
 - `width`, `height` 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 직사각형의 면적을 실수로 반환하는 메소드 `getArea()`을 Shape의 `getArea()`를 오버라이드하여 작성
- ✚ 클래스 Circle(원)을 추상클래스 Shape을 상속받아 다음 조건에 따라 정의하시오
 - 필드: `radius`(반지름, 정수)
 - `radius` 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 원의 면적을 실수로 반환하는 메소드 `getArea()`을 Shape의 `getArea()`를 오버라이드하여 작성
- ✚ 클래스 Test의 `main()` 내에서 다음 절차를 코딩하시오
 - 다음 객체들을 **Shape 배열**에 저장
 - ◆ Rectangle 객체(가로 3, 세로 4), Circle 객체(반지름 5), Circle 객체(반지름 2)
 - 반복문을 통해 배열에 저장된 각 도형의 면적(`getArea()` 호출)을 출력

인터페이스 실습 A

- ✚ 인터페이스 ShapeAction을 다음 조건에 따라 정의하시오
 - 추상메소드 void moveTo(int x, int y): 도형을 (x,y) 좌표 위치로 이동
 - 추상메소드 void setColor(int color): 도형의 색을 color로 설정
- ✚ 다음 조건에 따라 클래스 Rectangle(직사각형)을 인터페이스 ShapeAction을 구현하여 정의하시오
 - 필드: x(왼쪽상단꼭지점 x좌표,정수), y(왼쪽상단꼭지점 y좌표,정수), width(가로 길이,정수), height(세로 길이,정수)
 - x, y, width, height 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 직사각형의 왼쪽상단꼭지점 위치를 변경하는 메소드 moveTo()를 ShapeAction의 moveTo()를 오버라이드하여 작성
 - x, y, width, height 값을 문자열로 반환하는 toString()
- ✚ 클래스 Test의 main() 내에서 다음 절차를 코딩하시오
 - 다음 객체를 생성하고 moveTo()를 호출하여 위치를 (120,130)으로 이동 후 객체 출력
 - ◆ Rectangle 객체(왼쪽상단꼭지점 (50,70), 가로 3, 세로 4)

인터페이스 실습 B

- ✚ 인터페이스 ShapeAction을 다음 조건에 따라 정의하시오
 - 추상메소드 `double getArea()`: 도형의 면적 반환
 - 원주율 값 3.14159를 갖는 상수 변수 `PI` 정의
- ✚ 다음 조건에 따라 클래스 Circle(원)을 인터페이스 ShapeAction을 구현하여 정의하시오
 - 필드: `radius`(반지름, 정수)
 - `radius` 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 원의 면적을 실수로 반환하는 메소드 `getArea()`을 ShapeAction의 `getArea()`를 오버라이드하여 작성
 - ◆ ShapeAction의 `PI`를 이용하여 원 면적 계산
- ✚ 클래스 Test의 `main()` 내에서 다음 절차를 코딩하시오
 - Circle 객체(반지름 5) 생성 후 `getArea()` 호출하여 면적 출력

인터페이스 실습 C

- ✚ 인터페이스 ShapeAction을 다음 조건에 따라 정의하시오
 - 실수를 반환하는 추상메소드 getArea()
- ✚ 다음 조건에 따라 클래스 Rectangle(사각형)을 인터페이스 ShapeAction을 구현하여 정의하시오
 - 필드: width(가로 길이, 정수), height(세로 길이, 정수)
 - width, height 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 사각형의 면적을 실수로 반환하는 메소드 getArea()을 ShapeAction의 getArea()를 오버라이드하여 작성
- ✚ 다음 조건에 따라 클래스 Circle(원)을 인터페이스 ShapeAction을 구현하여 정의하시오
 - 필드: radius(반지름, 정수)
 - radius 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - 원의 면적을 실수로 반환하는 메소드 getArea()을 ShapeAction의 getArea()를 오버라이드하여 작성
- ✚ 클래스 Test의 main() 내에서 다음 절차를 코딩하시오
 - 다음 객체들을 **ShapeAction** 배열에 저장
 - ◆ Rectangle 객체(가로 3, 세로 4), Circle 객체(반지름 5), Circle 객체(반지름 2)
 - 반복문을 통해 배열에 저장된 각 도형의 면적(getArea() 호출)을 출력

인터페이스 실습 D

✚ 은행업무인터페이스 BankingAction을 다음 조건에 따라 작성하시오

- 입금 업무 추상메소드 void deposit(double money)
- 출금 업무 추상메소드 void withdraw(double money)
- 잔고 확인 추상메소드 double getBalance()

✚ 클래스 Test의 main() 내에서 다음 절차를 코딩하시오

- BankingAction을 구현하는 익명내부클래스의 객체를 다음 조건에 따라 작성하시오
 - ◆ 필드: balance(잔고, double)
 - ◆ deposit(double money) 구현: 입금액 money을 balance에 누적
 - ◆ getBalance() 구현: balance 값을 반환
- 위 객체에 대해 deposit()을 호출하여 1000원 입금 후, getBalance() 호출하여 현재 잔고 값 출력

인터페이스 실습 E

- ✚ 은행업무인터페이스 BankingAction을 다음 조건에 따라 작성하시오
 - 입금 업무 추상메소드 void deposit(double money)
 - 출금 업무 추상메소드 void withdraw(double money)
- ✚ 은행계좌클래스 BankAccount를 다음 조건에 따라 작성하시오
 - 필드: id(아이디,String), balance(잔고,double), interestRate(이자율,double)
 - id, interestRate 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - id, interestRate, balance를 반환하는 toString()
- ✚ 온라인은행계좌클래스 OnlineBankAccount를 다음 조건에 따라 작성하시오.
 - BankAccount를 상속하고 BankingAction을 구현하여 작성
 - id, interestRate 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - Deposit(int money) 구현: 입금액(money)을 잔고에 누적 후, 입금액(money)의 일정비율(interestRate)을 잔고에 추가 누적
 - Withdraw(int money) 구현: 출금액(money)을 잔고에서 삭감
- ✚ 오프라인은행계좌클래스 OfflineBankAccount를 다음 조건에 따라 작성하시오.
 - BankAccount를 상속하고 BankingAction을 구현하여 작성
 - id, interestRate 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
 - deposit(int money) 구현: 입금액(money)을 잔고에 누적 후, 잔고의 일정비율(interestRate)을 잔고에 추가 누적
 - withdraw(int money) 구현: 출금액(money)을 잔고에서 삭감
- ✚ 클래스 Test의 main() 내에서 다음 절차를 코딩하시오
 - OnlineBankAccount 객체(id=O-001, interestRate=0.1)를 생성하고 5000원, 1000원을 차례로 입금한 후 해당 객체를 출력
 - OfflineBankAccount 객체(id=F-001, interestRate=0.1)를 생성하고 5000원, 1000원을 차례로 입금한 후 해당 객체를 출력

인터페이스 실습 E

```
public interface BankingAction {  
    void deposit(double money);  
    void withdraw(double money);  
}
```

```
public class BankAccount {  
    String id;  
    double balance;  
    double interestRate;  
    public BankAccount(String id, double interestRate) {  
        this.id=id;  
        this.interestRate=interestRate;  
    }  
    @Override  
    public String toString() {  
        return id+" "+interestRate+" "+balance;  
    }  
}
```

```
public class OfflineBankAccount extends BankAccount implements BankingAction {  
    public OfflineBankAccount(String id, double interestRate) {  
        super(id, interestRate);  
    }  
    @Override  
    public void deposit(double money) {  
        balance+=money;  
        balance+=money*interestRate;  
    }  
    @Override  
    public void withdraw(double money) {  
        balance-=money;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        OnlineBankAccount ba1=new OnlineBankAccount("O-001", 0.1);  
        ba1.deposit(5000);  
        ba1.deposit(1000);  
        System.out.println(ba1);  
  
        OfflineBankAccount ba2=new OfflineBankAccount("F-001", 0.1);  
        ba2.deposit(5000);  
        ba2.deposit(1000);  
        System.out.println(ba2);  
    }  
}
```

```
public class OnlineBankAccount extends BankAccount implements BankingAction {  
    public OnlineBankAccount(String id, double interestRate) {  
        super(id, interestRate);  
    }  
    @Override  
    public void deposit(double money) {  
        balance+=money;  
        balance+=balance*interestRate;  
    }  
    @Override  
    public void withdraw(double money) {  
        balance-=money;  
    }  
}
```

인터페이스 실습 F

✚ 인터페이스 VolumeAction을 다음 조건에 따라 작성하시오

- 추상메소드 void volumeUp()
- 추상메소드 void volumeDown()
- 추상메소드 int getVolume()

✚ 클래스 Test의 main() 내에서 다음 절차를 코딩하시오

- VolumeAction을 구현하는 익명내부클래스의 객체를 다음 조건에 따라 작성하시오
 - ◆ 필드: volumeLevel(볼륨레벨, int)
 - ◆ volumeUp() 구현: volumeLevel을 1씩 증가
 - ◆ volumeDown() 구현: volumeLevel을 1씩 감소
 - ◆ getVolume() 구현: volumeLevel 값을 반환
- 위 객체에 대해 volumeUp() 3회, volumeDown() 1회 호출 후, getVolume() 호출하여 현재 볼륨 값 출력

인터페이스 실습 G



인터페이스 VolumeAction을 다음 조건에 따라 작성하시오

- 추상메소드 void volumeUp();
- 추상메소드 void volumeDown();



클래스 Radio를 다음 조건에 따라 작성하시오

- 인터페이스 VolumeAction을 구현하여 작성
- 필드: volumeLevel(볼륨레벨, int)
- static 필드: MAX_VOLUME(최대볼륨값, int, 초기값 10)
- volumeUp() 구현
 - ◆ MAX_VOLUME 값을 초과하지 않는 범위 내에서 volumeLevel을 1씩 증가
- volumeDown() 구현
 - ◆ 0 미만의 값이 되지 않는 범위 내에서 volumeLevel을 1씩 감소
- volumeLevel을 반환하는 toString() 작성



클래스 Smartphone을 다음 조건에 따라 작성하시오

- 인터페이스 VolumeAction을 구현하여 작성
- 필드: volumeLevel(볼륨레벨, int)
- Static 필드: MAX_VOLUME(최대볼륨값, int, 초기값 100)
- volumeUp() 구현
 - ◆ MAX_VOLUME 값을 초과하지 않는 범위 내에서 volumeLevel을 5씩 증가
- volumeDown() 구현
 - ◆ 0 미만의 값이 되지 않는 범위 내에서 volumeLevel을 5씩 감소
- volumeLevel을 반환하는 toString() 작성




클래스 Test의 main() 내에서 다음 절차를 코딩하시오

- Radio 객체를 생성한 후 volumeUp() 3회, volumeDown() 1회 호출 후 해당 객체 출력
- Smartphone 객체를 생성한 후 volumeUp() 100회, volumeDown() 1회 호출 후 해당 객체 출력

References

 <http://docs.oracle.com/javase/7/docs/api/>

 <https://docs.oracle.com/javase/tutorial/java/>

 김윤명. (2008). 뇌를 자극하는 Java 프로그래밍. 한빛미디어.

 남궁성. 자바의 정석. 도우출판.

 황기태, 김효수 (2015). 명품 Java Programming. 생능출판사.