자료구조 실습 보고서

[제 3주] : 동전 가방

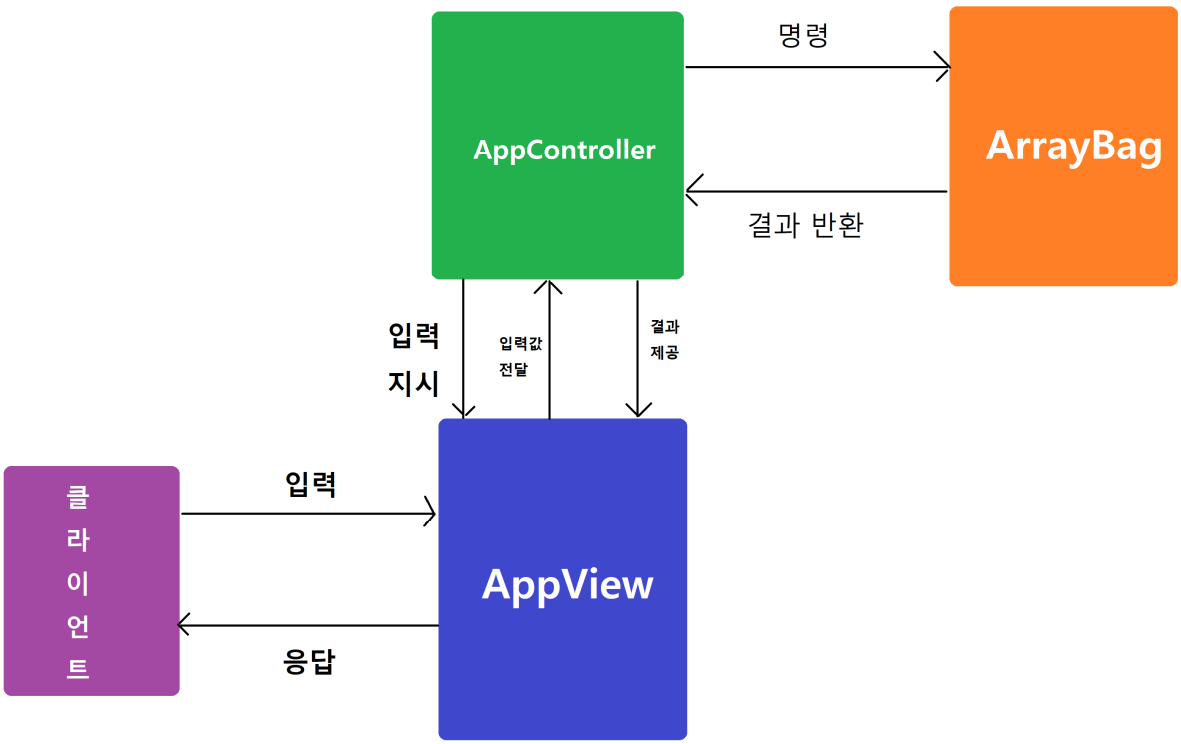
제출일: 2022-03-24(목)

201902708 신동훈

프로그램 설명

1. 프로그램의 전체 설계 구조

간단하게 표현하면 다음과 같다.



MVC(Model - View - Controller)구조를 따르며, 각각의 역할은 다음과 같다.

AppController

AppView에게 입력 요청을 보내며, 그에 대한 응답으로 반환된 입력값을 받아온다.
ArrayBag에게 Coin 객체를 전달하며 보관, 삭제 등의 계산을 요청하고, 결과를 받아온다.
AppView에게 출력 정보를 제공하며, 출력을 요청한다.

AppView

화면(콘솔)에 메시지를 출력하는 역할과, 화면으로부터 입력값을 받아와 다른 객체에게 전달하는 역할을 담당한다.

ArrayBag

Coin 객체를 받아서 저장하거나 삭제하는 등의 로직을 수행한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

2-1. 사용된 자료구조

ArrayBag은 내부적으로 1차원 배열을 사용한다.
배열이기 때문에, 생성할 당시 지정해준 용량을 초과해서 값을 담을 수 없으며, 담기 위해서는 크기가 더 큰 배열을 새로 만들어준 후 기존의 값을 이동시켜서 구현하는 방식으로 늘릴 수 있겠지만, 이번 ArrayBag에서는 그렇게 하지 않았다.
또한 제네릭을 사용하여 동일한 코드를 통해 여러 타입의 객체들을 보관할 수 있도록 만들었다.

2-2. 주요 함수

간단한 설명

ArrayBag의 함수

public ArrayBag() : 기본 용량(100)의 ArrayBag 객체를 생성한다.
public ArrayBag(int givenCapacity) : **givenCapacity를 용량으로 가지는**ArrayBag 객체를 생성한다.
public int size() : 현재 가방의 사이즈를 반환한다.
public boolean isEmpty() : 가방이 현재 비어있는지를 확인한다.
public boolean isFull() : 가방이 현재 가득찬 상태인지를 반환한다.
public boolean doesContain(E anElement) : 주어진 원소와 동일한 원소가 가방 속에 존재하는지 확인한다.
public int frequencyOf(E anElement) : 가방 속에 주어진 원소가 몇 개 있는지를 확인하여 반환한다.
public boolean add(E anElement) : 주어진 원소를 가방에 추가한다.
public boolean remove(E anElement) : 주어진 원소를 가방에서 제거한다.
public void clear() : 가방을 비운다
public E elementAt(int order) : 주어진 위치에 있는 원소를 반환한다

자세한 구현

```
/**
 * _capacity 를 DEFAULT_CAPACITY 로 초기화
 * _elements 를 _capacity 만큼 생성
 * _size 를 0으로 초기화
 */
@SuppressWarnings("unchecked")
public ArrayBag() {
    this.setCapacity(ArrayBag.DEFAULT_CAPACITY);
    this.setElements((E[]) new Object[_capacity]);
    this.setSize(0);
}


/**
 * _capacity 를 givenCapacity 로 초기화
 * _elements 를 _capacity 만큼 생성
 * _size 를 0으로 초기화
 */
@SuppressWarnings("unchecked")
public ArrayBag(int givenCapacity) {
    this.setCapacity(givenCapacity);
    this.setElements((E[]) new Object[_capacity]);
    this.setSize(0);
}
```

```
/**
 * 현재 가방의 사이즈를 반환한다
 * @return 현재 가방의 사이즈
 */
public int size() {
    return this._size;
}

/**
 * 가방이 현재 비어있는지를 확인한다.
 * @return 가방이 비었으면 true, 비어있지 않다면 false
 */
public boolean isEmpty() {
    return size()==0;
}

/**
 * 가방이 현재 가득찬 상태인지를 반환한다.
 * @return 가방이 가득 차있다면 true, 공간이 남아있다면 false
 */
public boolean isFull() {
    return size() ==capacity();
}

/**
 * 주어진 원소와 동일한 원소가 가방 속에 존재하는지 확인한다
 * @param anElement 포함되어있는지 확인할 원소
 * @return 포함되어 있다면 true, 포함되어있지 않다면 false
 */
public boolean doesContain(E anElement) {
    boolean found = false;//원소를 찾으면 true 로 변경
    for (int i = 0; i < size() && !found; i++) {
   //_elements[] 의 값을 모두 확인하였거나 found 가 true 이면 찾는것을 종료
        if(anElement.equals(this.elements()[i])){
            found = true;
        }
    }
    return found; // found 값을 반환
}
```

```
/**
 * 가방 안에 주어진 원소가 몇 개 있는지를 돌려준다.
 * @param anElement 빈도수를 확인할 원소
 * @return 가방 속에 포함된 주어진 원소의 개수
 */
public int frequencyOf(E anElement) {
    int frequencyCount = 0;
    for (int i = 0; i < size(); i++) {
        if(anElement.equals(elements()[i])){
            frequencyCount++;
        }
    }
    return frequencyCount;
}



/**
 * 주어진 원소를 가방에 추가한디.
 * @param anElement 가방에 추가할 원소
 * @return 주어진 가방이 가득 차 있는 경우 false 를 반환한다.
 */
public boolean add(E anElement){
    if(isFull()) return false; //가방이 가득 차 있는 경우 false 를 반환

    this.elements()[this.size()] = anElement;
    this.setSize(this.size()+1);
    return true;
}
```

```
    /**
     * 주어진 원소를 가방에서 제거한다.
     * @param anElement 가방에서 제거할 원소.
     * @return 가방이 비어있거나, 주어진 원소를 가방에서 찾지 못한다면 false 를 반환한다.
     */
    public boolean remove(E anElement){
        if(isEmpty()) return false;
        int foundIndex = -1;//제거할 원소가 존재하는 위치(인덱스)

        //주어진 원소의 의치를 찾는다
        for (int i = 0; i < size() && foundIndex < 0; i++) {
            if(this.elements()[i].equals(anElement)){//만약 제거할 원소를 찾았다면 해당 원소가
존재하는 인덱스를 저장해준다
                foundIndex = i;
            }
        }

        //제거할 원소를 찾지 못했다면 false 를 반환한다.
        if(foundIndex < 0){
            return false;
        }

        //제거할 원소 이후의 모든 원소들을 앞쪽으로 한 칸씩 이동시킨다
        for (int i = foundIndex; i < size()-1; i++) {
            this.elements()[i] = this.elements()[i+1];
        }

        this.elements()[this.size()-1] = null;//다 쓴 참조를 해제
        this.setSize(this.size()-1); //사이즈를 1 줄인다.
        return true;
    }
```

```
/**
 * 가방을 비운다
 */
public void clear(){
    for (int i = 0; i < size(); i++) {
        this.elements()[i] = null;//참조를 모두 해제한다
    }
    this.setSize(0);//size를 0으로 설정한다
}



/**
 * 주어진 순서에 있는 원소를 돌려준다
 * @param order 찾아낼 순서
 * @return order 위치에 존재하는 요소
 */
public E elementAt(int order){
    if( 0<= order && order< this.size()){
        return this.elements()[order];
    }
    else {
        return null;
    }
}
```

Coin의 함수

public Coin(): Coin 객체를 생성한다. 코인의 값은 기본 값(0)이다.

public Coin(int givenValue):전달받은 값을 갖는 새로운Coin 객체를 생성한다.

public int value(): 코인의 금액을 얻는다.

public void setValue(int newValue): 전달받은 newValue를 현재 코인의 금액으로 설정한다.

public boolean equals(Object otherCoin) ;현재 코인의 금액이 otherCoin 의 금액과 같은지 확인한다

자세한 구현

```
/**
 * DEFAULT_VALUE 값을 가진 Coin 객체를 생성한다.
 */
public Coin() {
    this._value = DEFAULT_VALUE;
}

/**
 * givenValue 값을 가진 Coin 객체를 생성한다.
 */
public Coin(int givenValue) {
    this._value = givenValue;
}

/**
 * Coin 의 값을 반환한다
 * @return 코인의 value
 */
public int value() {
    return _value;
}

/**
 * Coin 의 값을 설정한다
 * @param newValue 설정할 값
 */
public void setValue(int newValue) {
    this._value = newValue;
}
```

```
/**
 * 현재 코인의 금액이 otherCoin과 같은지 확인한다
 * @param otherCoin 비교할 대상 Coin
 * @return 같으면 true, 다르다면 false
 */
@Override
public boolean equals(Object otherCoin) {
    if (otherCoin.getClass() != Coin.class){
        return false;
    }else {
        return (this.value() == ((Coin)otherCoin)._value);
    }
}
```

AppController의 함수

public void run(): 프로그램을 실행한다. 사용자로부터 값을 입력받아 작업을 처리한다
private void addCoin(): 동전을 가방에 넣는 작업을 처리한다
private void removeCoin(): 동전을 가방에서 제거하는 작업을 처리한다
private void searchCoin(): 동전을 가방에서 찾는 작업을 처리한다
private void frequencyCoin(): 동전의 빈도수를 구하는 작업을 처리한다.
private void undefinedMenuNumber(int menuNumber): 사용자가 잘못된 값을 입력하였을때 이를 처리한다.
private void showStatistics(): 현재 가방안에 있는 동전의 상태(동전의 개수, 가장 큰 동전의 값, 동전의 합)를 보여준다.
private int sumOfCoinValue(): 동전의 합을 구한다
private int maxCoinValue(): 가방 속 가장 값이 큰 동전을 구한다.

자세한 구현

```
public void run() {
    AppView.outputLine("<<< 동전 가방 프로그램을 시작합니다 >>>");
    AppView.outputLine("");

    int coinBagSize = AppView.inputCapacityOfCoinBag();
    this.setCoinBag(new ArrayBag<Coin>(coinBagSize));


    int menuNumber = AppView.inputMenuNumber();
    while (menuNumber != MENU_END_OF_RUN){
        switch ( menuNumber ){
            case MENU_ADD:
                this.addCoin();
                break;
            case MENU_REMOVE:
                this.removeCoin();
                break;
            case MENU_SEARCH:
                this.searchCoin();
                break;
            case MENU_FREQUENCY:
                this.frequencyCoin();
                break;
            default:
                this.undefinedMenuNumber(menuNumber);
        }
        menuNumber = AppView.inputMenuNumber();
    }

    this.showStatistics();
    AppView.outputLine("<<< 동전 가방 프로그램을 종료합니다 >>>");

}
```

```
/**
 * 동전을 가방에 넣는 작업을 제어한다.
 */
private void addCoin() {
    if( this.coinBag().isFull() ){
        AppView.outputLine("- 동전 가방이 꽉 차서 동전을 넣을 수 없습니다");
    }
    else {
        int codeValue = AppView.inputCoinValue();
        if(this.coinBag().add(new Coin(codeValue))){
            AppView.outputLine("- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.");
        }else {
            AppView.outputLine("- 주어진 값을 갖는 동전을 가방에 넣는데 실패하였습니다.");
        }
    }
}


/**
 * 동전을 가방에서 제거하는 작업을 제어한다.
 */
private void removeCoin() {
    int coinValue = AppView.inputCoinValue();
    if(! this.coinBag().remove(new Coin(coinValue))){
        AppView.outputLine("- 주어진 값을 갖는 동전은 가방 안에 존재하지 않습니다.");
    }else{
        AppView.outputLine("- 주어진 값을 갖는 동전 하나가 가방에서 정상적으로 삭제되었습니다.");
    }
}

/**
 * 동전을 가방에서 찾는 작업을 제어한다.
 */
private void searchCoin() {
    int coinValue = AppView.inputCoinValue();
    if (this.coinBag().doesContain(new Coin(coinValue))){
        AppView.outputLine("- 주어진 값을 갖는 동전이 가방 안에 존재합니다.");
    }else {
        AppView.outputLine("- 주어진 값을 갖는 동전은 가방 안에 존재하지 않습니다.");
    }
}
```

```
/**
 * 동전의 빈도수를 구하는 작업을 제어한다
 */
private void frequencyCoin() {
    int coinValue = AppView.inputCoinValue();
    int frequency = this.coinBag().frequencyOf(new Coin(coinValue));
    AppView.outputLine("- 주어진 값을 갖는 동전의 개수는 " + frequency + "개 입니다.");
}

/**
 * 잘못된 번호를 입력받았을 경우에 대한 처리를 담당한다.
 * @param menuNumber 입력받은 번호
 */
private void undefinedMenuNumber(int menuNumber) {
    AppView.outputLine("- 선택된 메뉴 번호 " + menuNumber+ " 는 잘못된 번호입니다.");
}


/**
 * 가방 속 상태를 보여준다
 */
private void showStatistics() {
    AppView.outputLine("가방에 들어 있는 동전의 개수: " + this.coinBag().size());
    AppView.outputLine("동전 중 가장 큰 값: " + this.maxCoinValue());
    AppView.outputLine("모든 동전 값의 합: " + this.sumOfCoinValue());
    AppView.outputLine("");
}
```

```
/**
 * 가방속 동전의 합을 구한다
 * @return 동전의 총 합
 */
private int sumOfCoinValue() {
    int sum = 0;
    for (int i = 0; i < this.coinBag().size(); i++) {
        sum += this.coinBag().elementAt(i).value();
    }
    return sum;
}

/**
 * 가방속 가장 값이 큰 동전의 값을 구한다
 * @return 가장 큰 동전의 값
 */
private int maxCoinValue() {
    int maxValue = 0;
    for (int i = 0; i < this.coinBag().size(); i++) {
        if(maxValue < this.coinBag().elementAt(i).value()){
            maxValue = this.coinBag().elementAt(i).value();
        }
    }

    return maxValue;
}
```

3. 종합 설명

해당 프로그램은 동전을 가방에 넣고 빼는 동작과, 해당 과정들이 끝날 때 가방의 상태를 보여주는 프로그램이다.
앱을 실행하기 위해서는 '_DS03_Main_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

처음은 항상 동전을 저장할 가방의 용량을 정해주는 것으로 시작된다. 사용자가 입력한 용량을 가지고 AppController는 새로운 ArrayBag 객체를 생성한다. 이후 가방에 대하여 수행할 수 있는 행위를 선택지를 제공한다. 가방에 동전을 추가(add)하거나, 가방에서 동전을 제거(remove)할 수 있으며, 입력받은 값의 동전이 가방속에 존재하는지 검색(search)할 수 있고, 동전의 빈도수(frequancy)를 확인할 수 있다.

마지막으로 9를 입력하면 현재 가방에 들어있는 동전의 개수와, 동전 중 가장 큰 값, 모든 동전 값의 크기를 보여주며 프로그램은 종료된다.

프로그램 장단점/특이점

1. 프로그램의 장점

MVC 패턴을 사용하였기에 유지보수성과, 이후 요구사항의 변경 시 코드의 변경의 파급효과가 적을 뿐더러 ArrayBag 클래스는 제네릭을 사용하여 다양한 객체들을 가방에 담을 수 있음과 동시에 잘못된 객체를 담게되었을 때 런타임이 아닌 컴파일 시에 예외가 발생하여 실수를 방지할 수 있다는 장점이 있다. 현재 프로그램에서는 Coin 객체만 담았지만, Coin 뿐만이 아니라 어떤 객체도 담을 수 있기에 프로그램을 확장하는 데 있어서 간편하다는 장점을 가진다.

2 프로그램의 단점

제네릭을 사용하였긴 했지만, 가방에는 담을 수 없는 것들, 혹은 담아서는 안 되는 것들이 존재한다. 그러나 현재 ArrayBag은 어떠한 객체던 차별 없이 가방에 보관할 수 있다는 문제점을 갖는다.
이를 해결하기 위해서는 간단한 예시로 '담을 수 있는'이라는 속성을 인터페이스로 정의해둔 후, 제네릭의 extends 키워드를 사용하여 상한 제한을 걸 수 있다.

예를 들면 다음과 같다.
'public class ArrayBag<E extends Containable> {...}'

실행 결과 분석

1. 입력과 출력

```
C:\Users\user\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ
 깃허브\School-Lecture-2022\3주차\out\production\3주차 _DS03_Main_201902708_신동훈
<<< 동전 가방 프로그램을 시작합니다 >>>

? 동전 가방의 크기, 즉 가방에 들어갈 동전의 최대 개수를 입력하시오: 6

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
? 동전 값을 입력하시오: 5
- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
? 동전 값을 입력하시오: 10
- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
? 동전 값을 입력하시오: 20
- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
? 동전 값을 입력하시오: 5
- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
? 동전 값을 입력하시오: 6
- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
? 동전 값을 입력하시오: 1
- 주어진 값을 갖는 동전을 가방에 성공적으로 넣었습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 1
- 동전 가방이 꽉 차서 동전을 넣을 수 없습니다
```

```
? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 3
? 동전 값을 입력하시오: 20
- 주어진 값을 갖는 동전이 가방 안에 존재합니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 3
? 동전 값을 입력하시오: 70
- 주어진 값을 갖는 동전은 가방 안에 존재하지 않습니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 4
? 동전 값을 입력하시오: 5
- 주어진 값을 갖는 동전의 개수는 1개 입니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 4
? 동전 값을 입력하시오: 80
- 주어진 값을 갖는 동전의 개수는 0개 입니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 0
- 선택된 메뉴 번호 0 는 잘못된 번호입니다.

? 수행하고자 하는 메뉴 번호를 선택하시오 (add: 1, remove: 2, search: 3, frequency: 4, exit: 9) : 9
가방에 들어 있는 동전의 개수: 5
동전 중 가장 큰 값: 20
모든 동전 값의 합: 42

<<< 동전 가방 프로그램을 종료합니다 >>>
```

2. 결과 분석

프로그램을 시작하면 AppController는 AppView의 outputLine 메서드를 통해 프로그램을 시작한다는 문구를 출력시키며, 이후 inputCapacityOfCoin 메서드를 통해 코인 가방의 사이즈를 입력받도록 명령한다.
이후 이 값으로 새로운 ArrayBag 객체를 생성한 후, 사용자에게 번호를 받아와서 해당 번호에 해당하는 행동들을 수행한다.
처음 가방의 용량을 6으로 설정한 후, 동전 6개를 add한 뒤에 한번 더 add를 시도하였더니 가방이 가득 차서, 동전을 넣을 수 없다는 메시지를 출력하였다. 이때 coinBag의 ArrayBag의 isFull 메서드를 사용하여 가방이 가득 찼는지의 여부를 판한한다.
동전을 제거하거나, 동전이 존재하는지의 여부, 혹은 빈도수를 구할 때도 정삭적으로 작동한다.
그러나 맨 처음 동전 가방의 용량을 -1처럼 음숫값을 설정하게 된다면 new Object[음수]부분에서 예외가 발생하여 프로그램이 종료된다.

생각해 볼 점에 대한 의견

1. Class의 인스턴스 변수의 getter/setter 가, “private" 함에도 불구하고 만들어 사용하는 이유는?

추상화를 더욱 강하게 함으로써 여러 이점을 취하기 위함이다.
getter와 setter를 만들어서 사용하지 않는다면, 개발하는 개발자의 입장에서는 넣어서는 안 될 값을 실수로 넣을 가능성이 있으며, 이는 이후 오류가 발생했을 때 원인을 분석하는데 어려움을 줄 수 있다.
내부에서도 인스턴스 변수를 바로 사용하는 것이 아니라, getter/setter를 사용한다면 변수를 사용하는 시점을 완벽하게 파악할 수 있어서, 이후 디버깅에도 용이하다.

게다가 추가적인 로직을 적용할 수도 있는데, 예를 들어 이번 프로그램에서 가방의 용량은 음수가 될 수 없다. 이런 경우 setter 속에 capacity가 0보다 작은 값이 들어온다면 예외를 발생시키도록 구현할 수도 있다.
즉 클래스 내부에서마저도 캡슐화를 적용하여 구현을 숨기는 것이다.

추가로 대부분의 개발자들이 인스턴스 필드를 private로 설정하기에, 필드의 접근 제어자만 보고는 외부에서 수정해도 되는 필드와 그렇지 않는 필드를 구분하는 것에 어려움을 겪을 가능성이 있다.
getter/setter가 private이라면 이는 외부에 노출해서는 안되는 필드라는 것을 암시하기 때문에 이후 기능을 확장할 때 클래스를 사용하는 것을 더욱 간편하게 만들어준다.

위와 같은 이점들 덕분에 getter, setter가 private 함에도 불구하고 만들어 사용하는 것이다.

그러나 당연하게도 단점도 존재한다.

우선 개발자가 항상 getter와 setter를 만들어주어야 한다는 것부터가 번거롭다.
또한 자바빈 프로터피 규약에 따라 getter와 setter는 인스턴스 변수의 이름을 사용하여 생성하는데, 만약 클래스에 변경이 생겨 인스턴스 변수의 이름을 변경해야 할 경우, getter와 setter를 변경해주어야 한다. 그렇지 않으면 이후 코드를 사용하여 기능을 추가하거나 변경할 때, getter/setter와 인스턴스 변수의 이름이 달라 혼동이 생길 수 있기 때문이다.

2. Scanner는 왜 class의 “AppController”가 아닌 class "AppView"에서 선언하는가?

웹 사이트에서 화면을 클릭하거나 , 스마트폰으로 화면을 클릭, 파일의 글을 입력으로 사용하는 경우 등 입력에는 다양한 종류의 입력이 존재한다.
따라서 입력받는 환경에 따라서 입력받는 방법을 다르게 구현해 주어야 한다.

Model - View - Controller에서 View는 이름 그대로 화면과 관련된 역할을 수행하며, 출력 뿐만 아니라 입력 또한 View의 역할이다.
Controller는 이름 그대로 시스템을 ‘제어’하는 역할을 하는 것이기에, 입력값을 받아 이를 적절한 책임을 가진 객체에게 전달하는 역할을 담당하는 것이지, 입력을 어떻게 받아오는가는 Controller에서 신경 쓸 부분이 아니다.

입출력은 하드웨어에 의존적일 수밖에 없으며, View는 입출력을 담당하여 Controller나 Model은 입출력의 방식에 관계없이, 즉 하드웨어와 독립적으로 개발할 수 있도록 하드웨어를 ‘캡슐화’하는 역할을 수행한다.

이렇게 Model - View - Controller는 각각의 관심사가 있고, 이러한 관심사의 분리를 통해 프로그램의 유지보수성을 증가시키며, 변경에 대처하기 쉽도록 만들어주는 것이다.

Controller에서 Scanner를 선언하여 사용한다고 가정해보면, 입력을 받는 방식이 콘솔이 아닌 모바일 기기의 화면 터치인 경우, Controller는 프로그램의 ‘제어’ 때문이 아닌 ‘입력 화면의 변경’이라는 이유로 변경해야 된다. 이는 관심사의 분리가 제대로 이루어지지 않은 것이고, 이후 코드를 유지보수 하는데 악영향을 끼치게 된다.

따라서 Scanner는 AppController가 아닌 AppView에서 선언하여 사용하는 것이 바람직하다.

3. Generic Type 이 무엇이고, 어떤 장점이 있는가

Generic Type은 자바 1.5 버전부터 등장하였으며 작성한 코드를 다양한 객체에 대하여 재사용하기 위해 등장한 개념이다.
제네릭을 사용하면 다양한 타입의 객체들을 다루는 메서드나, 컬렉션 클래스에서 컴파일 시에 타입을 체크해주어 객체의 타입 안정성을 높이고 형변환을 해야하는 번거로움을 줄여주는 기능을 제공한다.
제네릭을 사용하지 않았을 시 발생하는 문제점을 코드를 통해 예시를 들어보겠다.

```
List list = new ArrayList();
list.add(1);
int a = list.iterator().next();  //컴파일 에러 발생!
```

위의 코드에서 주석을 달아놓은 부분에서 컴파일 에러가 발생한다.
그 이유는 list.iterator().next()는 Object 타입을 반환하는데 이를 int에 대입하려 했기 때문이다.
따라서 해당 값을 사용하려면 다음과 같이 형변환을 해야 한다

```
int a = (int) list.iterator().next();
```

그런데 위의 코드에서 list가 반환하는 타입이 int인 것을 보장하는 조건이 없다. 그렇기에 아래 경우에는 컴파일 시에는 오류가 발생하지 않으나, 런타임 시 오류가 발생한다.

```
List list = new ArrayList();
list.add(1);
list.add("1");
list.add(new double[]{1.4, 115.4});
int a  = (int)list.get(0);
String b  = (String)list.get(2);//런타임 에러
```

이를 해결하기 위해 도입된 것이 바로 제네릭이다.
제네릭을 사용하면 위의 코드를 다음과 같이 바꿀 수 있다.

```
List<Integer> list = new ArrayList();
list.add(1);
list.add("1"); //컴파일 에러
list.add(new double[]{1.4, 115.4});//컴파일 에러
```

런타임시 코드는 동일하게 동작하지만, 런타임에 발생할 수 있는 문제를 **컴파일 시점에 방지**할 수 있다는 것이 제네릭의 장점이라 생각한다.
(pf: 런타임 시 코드가 동일하게 작동하는 것은, 타입 이레이저 프로세스에 의해 런타임 시 제네릭 타입에 대한 정보는 소거되기 때문이다.)

위와 같은 장점을 가진 제네릭을 사용할 때는 주의해야 할 점이 몇 가지 있는데, 우선 제네릭 메서드는 오버로딩이 불가능하다.
다음 예시를 보자.
public static void outBox(Box<? extends Toy> box) {...}
public static void outBox(Box<? extends Robot> box) {...}
위 두 메서드는 컴파일 시 다음과 같은 오류 메시지를 발생시킨다.
“outBox'(Box<? extends Toy>)' clashes with 'outBox(Box<? extends Robot>)'; both methods have same erasure“

위와 같은 오류가 발생하는 이유는 타입 이레이저에 의해 위 코드는 컴파일 이후 타입 정보가 사라지고 다음과 같이 변경되게 된다.
public static void outBox(Box box) {...}
public static void outBox(Box box) {...}
결국 메서드의 시그니처가 같아지기 때문에 오버로딩이 불가능한 것이다.

또한 제네릭은 무변성(invariant) 하기 때문에 A 클래스가 B 클래스의 상위 타입이어도 ‘제네릭 클래스<A>’ 는 ‘제네릭 클래스<B>’의 상위 타입이 아니며 이로 인해 발생하는 여러 문제점은 extends 키워드를 사용하여 공변(convariant)으로 만들어 주거나, super 키워드를 사용하여 반공변(contravariant)하게 만들어 주어서 해결하여야 한다.