자료구조 실습 보고서

[제 7주] : 재귀(성적 처리)

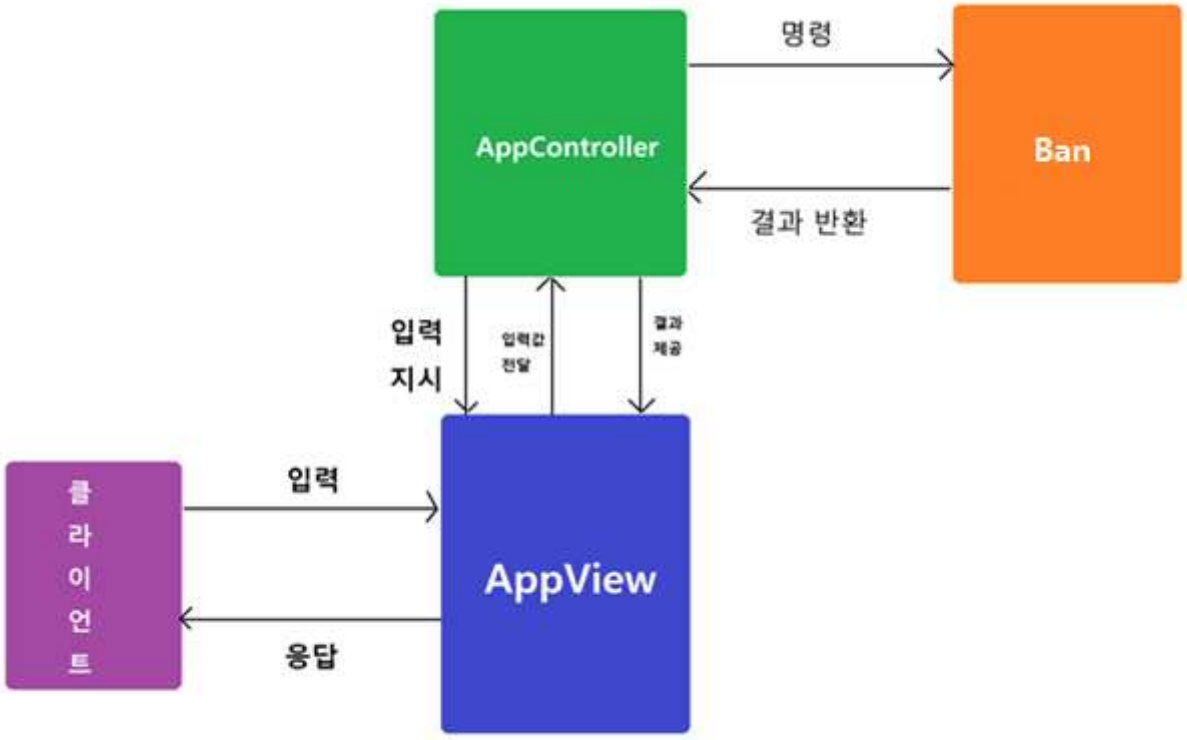
제출일: 2022-04-23(토)

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 출력 정보를 제공하며, 출력을 요청한다.
Model에게 명령을 지시하며 결과를 제공받아 AppView에게 전달한다.

AppView

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

Ban

UnsortedArrayList를 상속받아 학생들을 성적을 기준으로 정렬하고, 성적의 평균값 등을 구한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

2-1. 사용된 자료구조

Ban은 UnsortedArrayList를 상속받았으며, UnsortedArrayList는 내부적으로 배열을 사용하여 구현하였다.
배열은 메모리를 연속적인 공간에 할당하여, 컴파일 시 배열의 타입과 크기가 고정되어 있다는 단점이 있으나, 임의의 인덱스에 대한 접근에는 매우 빠른 속도로 연산이 가능하다는 장점이 있다.
또한 점수를 기준으로 정렬을 할 때 재귀적으로 Quick Sort 알고리즘을 사용하였다.
Quick Sort는 divide and conquer(분할 정복)방법으로 정렬을 수행하며, 리스트 안의 한 요소를 정해 이를 pivot으로 정한 후, 피봇을 기준으로 작은 요소들은 왼쪽, 큰 요소들은 오른쪽으로 옮긴 이후, 각각의 partition을 기준으로 재귀적으로 다시 피봇을 잡고 분할하며 정렬을 수행하는 알고리즘이다.

2-2. 주요 함수

AppView

```
/**
     * Constructor
     * AppView 는 객체를 사용하여 만들 것이 아니기 때문에
     * 생성자를 private 으로 정의하여 객체를 생성하여 사용하는 것을 방지한다.
     */
    private AppView() {}


    /**
     * 줄바꿈 없이 메세지를 출력한다.
     * @param message 출력할 메세지
     */
    public static void output(String message) {
        System.out.print(message);
    }

    /**
     * 메세지를 출력후 줄을 바꾼다.
     * @param message 출력할 메세지
     */
    public static void outputLine(String message) {
        System.out.println(message);
    }

    /**
     * 학급 학생 수를 출력한다.
     * @param aNumberOfStudents 학급 학생 수
     */
    public static void outputNumberOfStudents (int aNumberOfStudents) {
        AppView.outputLine("학급 학생 수:" + aNumberOfStudents);
    }

    /**
     * 학급 내 최고 점수를 출력한다.
     * @param aScore 학급 내 최고 점수
     */
    public static void outputHighestScore (int aScore) {
        AppView.outputLine("학급 최고 점수: " + aScore);
    }
    /**
     * 학급 내 최저 점수를 출력한다.
     * @param aScore 학급 내 최고저 점수
     */
    public static void outputLowestScore (int aScore) {
        AppView.outputLine("학급 최저 점수: " + aScore);
    }

    /**
     * 학급 평균 점수를 출력한다
     * @param anAverageScore 학급 평균 점수
     */
    public static void outputAverageScore (double anAverageScore) {
        AppView.outputLine(String.format("학급 평균: %.1f", anAverageScore));
    }
```

```
/**
     * 평균 이상인 학생 수를 출력한다.
     * @param aNumberOfStudents 평균 이상인 학생 수
     */
    public static void outputNumberOfStudentsAboveAverage (int aNumberOfStudents) {
        AppView.outputLine("평균 이상인 학생 수: " + aNumberOfStudents);
    }

    /**
     * 각 학점에 대한 학생 수 출력
     * @param aGrade
     * @param aNumberOfStudents
     */
    public static void outputNumberOfStudentsForGrade (char aGrade, int aNumberOfStudents) {
        AppView.outputLine(aGrade + " 학점의 학생 수는 " + aNumberOfStudents + " 입니다.");
    }

    /**
     * 학생들의 점수 출력
     * @param aScore
     */
    public static void outputScore (int aScore) {
        AppView.outputLine("점수:    " + aScore);
    }


    /**
     * 정수를 입력받는다.
     * @return 입력받은 정수
     * @throws NumberFormatException 정수 평테가 아닌 다른 형식으로 입력받을시 발생
     */
    public static int inputInt () throws NumberFormatException  {
        return Integer.parseInt(AppView.scanner.nextLine());
    }

    /**
     * 점수를 입력받는다.
     * @return 입력받은 점수
     */
    public static int inputScore() {
        while(true) {
            try {
                AppView.output("- 점수를 입력하시오 (0..100): ");
                int score = AppView.inputInt();
                return score;
            } catch ( NumberFormatException e) {
                AppView.outputLine("(오류) 정수가 입력되지 않았습니다.");
            }
        }
    }

    /**
     * 계속해서 점수를 입력받을지에 대한 여부를 반환한다.
     * @return 계속 입력받으면 true
     */
    public static boolean doesContinueToInputStudent() {
        AppView.output("성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: ");
        String line = null;
        do {
            line = AppView.scanner.nextLine();
        }while(line.equals("") );

        char answer = line.charAt(0);
        return ( (answer == 'Y') || (answer == 'y') );
    }
```

AppController

```
    /**
     * 입력한 점수가 범위를 벗어나지 않는지 확인
     * @param aSocre 입력받은 점수
     * @return 범위를 벗어났으면 false
     */
    private static boolean scoreIsValid(int aSocre) {
        return ( aSocre >= AppController.VALID_MIN_SCORE &&
                aSocre <= AppController.VALID_MAX_SCORE);
    }

    /**
     * 점수를 입력받는다
     * @return 입력받은 점수를 가진 Student 객체
     */
    private static Student inputStudent() {
        int score = AppView.inputScore();
        while(! AppController.scoreIsValid(score)) {
            AppView.outputLine("[오류] " +
                    AppController.VALID_MIN_SCORE + " 보다 작거나 " +
                    AppController.VALID_MAX_SCORE + " 보다 커서, 정상적인 점수가 아닙니다.");
            score = AppView.inputScore();
        }
        Student student = new Student();
        student.setScore(score);
        return student;
    }

/**
     * 점수를 ban 객체에 저장한다.
     */
    private void inputAndStoreStudent() {
        AppView.outputLine("");
        boolean storingAStudentWasSuccessful = true;
        while(storingAStudentWasSuccessful && AppView.doesContinueToInputStudent()) {
            Student student = AppController.inputStudent();
            if(! this.ban().add(student)) {
                AppView.outputLine("(경고) 입력에 오류가 있습니다. 학급에 더이상 학생을 넣을 공간이 없습니다.");
                storingAStudentWasSuccessful = false;
            }
        }
        AppView.outputLine("! 성적 입력을 마칩니다.");
    }

    /**
     * 성적 통계를 출력한다.
     * 학생수 , 최고 점수, 최저 점수, 평균 점수. 평균이상인 학생수
     */
    private void showStatistics() {
        AppView.outputLine("");
        AppView.outputLine("[학급 성적 통계]");

        AppView.outputNumberOfStudents(this.ban().size());
        AppView.outputHighestScore(this.ban().highest().score());
        AppView.outputLowestScore(this.ban().lowest().score());
        AppView.outputAverageScore(this.ban().average());
        AppView.outputNumberOfStudentsAboveAverage(this.ban().numberOfStudentsAboveAverage());
    }
```

```
/**
 * 학점별 학생수 출력
 */
private void showGradeCounts() {
    AppView.outputLine("");
    AppView.outputLine("[학점별 학생수]");

    this.setGradeCounter(this.ban().countGrades());
    AppView.outputNumberOfStudentsForGrade('A', this.gradeCounter().numberOfA());
    AppView.outputNumberOfStudentsForGrade('B', this.gradeCounter().numberOfB());
    AppView.outputNumberOfStudentsForGrade('C', this.gradeCounter().numberOfC());
    AppView.outputNumberOfStudentsForGrade('D', this.gradeCounter().numberOfD());
    AppView.outputNumberOfStudentsForGrade('F', this.gradeCounter().numberOfF());
}


/**
 * 성적순 목록 출력
 */
private void showStudentsSortedByScore() {
    AppView.outputLine("");
    AppView.outputLine("[학생들의 성적순 목록]");

    this.ban().sortByScore();

    Iterator<Student> iterator = this.ban().iterator();
    Student student = null;
    while (iterator.hasNext()) {
        student = iterator.next();
        AppView.outputScore(student.score());
    }
}



/**
 * 프로그램을 동작시킨다.
 */
public void run() {
    AppView.outputLine("");
    AppView.outputLine("<<<학급 성적 처리를 시작합니다 >>>");

    this.setBan(new Ban(AppController.BAN_CAPACITY));
    this.inputAndStoreStudent();
    if(this.ban().isEmpty()) {
        AppView.outputLine("");
        AppView.outputLine("(경고) 입력된 성적이 없습니다.");
    }
    else {
        this.showStatistics();
        this.showGradeCounts();
        this.showStudentsSortedByScore();
    }
    AppView.outputLine("");
    AppView.outputLine("<<< 학급 성적 처리를 종료합니다. >>>");

}
```

UnsortedArrayList<E extends Comparable<E>>

```
// Constructor
    @SuppressWarnings("unchecked")
    public UnsortedArrayList(int givenCapacity) {
        this.setCapacity(givenCapacity);
        this.setElements((E[]) new Comparable[this.capacity()]);
    }

    public UnsortedArrayList() {
        this(UnsortedArrayList.DEFAULT_CAPACITY);
    }

    /**
     * 특정 인덱스에 값을 집어넣기 위해 빈 공간을 만든다.
     * @param aPosition 공간을 만들 자리
     */
    private void makeRoomAt(int aPosition) {
        for (int i = this.size(); i > aPosition; i--) {
            this.elements()[i] = this.elements()[i - 1];
        }
    }

    /**
     * 특정 인덱스의 원소를 제거할 경우, 그 이후 위치의 원소들을 한칸씩 땡겨줄 때 사용한다.
     * @param aPosition 채울 인덱스
     */
    private void removeGapAt(int aPosition) {
        for (int i = aPosition + 1; i < this.size(); i++) {
            this.elements()[i - 1] = this.elements()[i];
        }
        this.elements()[this.size() - 1] = null;
    }


    public boolean isEmpty() {
        return (this.size() == 0);
    }

    public boolean isFull() {
        return (this.size() == this._capacity);
    }
```

```
/**
 * 원소의 존재를 확인한다.
 * @param anElement 존재를 확인할 원소
 * @return 있으면 true
 */
public boolean doesContain (E anElement) {
    return (this.orderOf(anElement) >= 0 ) ;
}

/**
 * 원소의 순서를 반환한다.
 * @param anElement 순서를 찾을 원소
 * @return 원소의 순서, 없으면 -1
 */
public int orderOf (E anElement) {

    int order = -1;// 가방이 비어있다면 return -1;

    for( int index = 0; index < this.size() && order <0; index++) {
        if(this.elements()[index].equals(anElement)) {
            order = index;
        }
    }
    return order;
}
/**
 * 주어진 순서에 있는 원소를 반환한다.
 * @param anOrder 순서
 * @return 순서에 존재하는 원소, 없으면 null
 */
public E elementAt(int anOrder) {
    if(anOrder < 0 || anOrder >= this.size()) {
        return null;
    }
    else {
        return this.elements()[anOrder];
    }
}

/**
 * 특정 인덱스에 원소를 삽입한다.
 * @param anOrder 삽입할 인덱스
 * @param anElement 삽입할 원소
 */
protected void setElementAt (int anOrder, E anElement) {
    if(anOrder < 0 || anOrder >= this.size()) {
        return ;
    }
    else {
        this.elements()[anOrder] = anElement;
    }
}

/**
 * 원소를 맨 앞에 삽입한다.
 * @param anElement 삽입할 원소
 * @return 성공시 true
 */
public boolean addToFirst(E anElement) {
    if(this.isFull()) {
        return false;
    }
    else {
        this.makeRoomAt(0);
        this.elements()[0] = anElement;
        this.setSize(this.size() +1);
        return true;
    }
}
```

```
/**
 * 원소를 가장 마지막에 삽입한다.
 * @param anelement 삽입할 원소
 * @return 성공하면 true
 */
public boolean addToLast (E anelement) {
    if(this.isFull()) {
        return false;
    }
    else {
        this.elements()[this.size()] = anelement;
        this.setSize(this.size() +1);
        return true;
    }
}

/**
 * 원소를 임의의 순서(ArrayList의 경우 가장 마지막)에 삽입한다.
 * @param anElement 삽입할 원소
 * @return 성공시 true
 */
public boolean add(E anElement) {
    return this.addToLast(anElement);
}

/**
 * 가장 첫 원소를 삭제한다.
 * @return 삭제한 원소
 */
public E removeFirst() {
    if(this.isEmpty()) {
        return null;
    }
    else {
        E removedElement = this.elements()[0];
        this.removeGapAt(0);
        this.setSize(this.size() -1);
        return removedElement;
    }
}

/**
 * 맨 마지막 원소를 삭제한다.
 * @return 삭제한 원소
 */
public E removeLast() {
    if (this.isEmpty()) {
        return null;
    }
    else {
        E removedElement = this.elements()[this.size()];
        this.setSize(this.size() -1);
        return removedElement;
    }
}
```

```
/**
 * 임의의 순서(ArrayList의 경우 가장 마지막)의 원소를 삭제한다.
 * @return 삭제한 원소
 */
public E removeAny() {
    return this.removeLast();
}

/**
 * 주어진 원소를 삭제한다.
 * @param anelement 삭제할 원소
 * @return 성공시 true
 */
public boolean remove (E anelement) {
    //위치 찾기
    int foundIndex = this.orderOf(anelement);
    //존재하면 삭제
    if(foundIndex < 0) {
        return false; //not found.
    }
    else {
        //삭제 후 배열 정렬
        this.removeGapAt(foundIndex);
        this.setSize(this.size() -1);
        this.elements()[this.size()] = null;
        return true;
    }
}
/**
 * 반복자를 반환한다.
 * @return 반복자 객체
 */
public Iterator<E> iterator() {
    return (new ListIterator());
}

//inner class Iterator
private class ListIterator implements Iterator<E> {
    //private value
    private int _nextPosition;

    //Getter / Setter
    private int nextPosition() {
        return this._nextPosition;
    }

    private void setNextPosition(int newNextPosition) {
        this._nextPosition = newNextPosition;
    }

    private ListIterator() {
        this.setNextPosition(0);
    }

    @Override
    public boolean hasNext() {
        return (this.nextPosition() < UnsortedArrayList.this.size());
    }

    public E next() {
        E nextElement = null;
        if (this.hasNext()) {
            nextElement = UnsortedArrayList.this.elements()[this.nextPosition()];
            this.setNextPosition(this.nextPosition() + 1);
        }
        return nextElement;
    }
}
```

Ban

```
/**
     * 점수를 학점으로 변환한다.
     * @param aScore 점수
     * @return 학점
     */
    private static char scoreToGrade(int aScore) {
        if (aScore >= 90) {
            return 'A';
        }
        else if(aScore >= 80) {
            return 'B';
        }
        else if(aScore >= 70) {
            return 'C';
        }
        else if(aScore >= 60) {
            return 'D';
        }
        else return 'F';
    }

    //Constructor
    public Ban() {
        super();
    }
    public Ban(int givenCapacity) {
        super(givenCapacity);
    }

    /**
     * 점수가 가장 낮은 학생을 찾아서 반환한다.
     * @return 점수가 가장 낮은 학생
     */
    public Student lowest() {
        if(this.isEmpty()) {
            return null;
        }
        else {
            return this.lowestRecursively(0, this.size()-1);
        }
    }

    /**
     * 점수가 가장 낮은 학생을 재귀적으로 찾는다.
     * @param left 가장 왼쪽 범위
     * @param right 가장 오른쪽 범위
     * @return 성적이 가장 낮은 학생
     */
    private Student lowestRecursively(int left, int right) {
        if(left == right) return this.elementAt(left);
        else {
            Student lowestFromRights = lowestRecursively(left +1, right);
            if(lowestFromRights.compareTo(this.elementAt(left)) <= 0) {
                return lowestFromRights;
            }
            else {
                return this.elementAt(left);
            }
        }
    }
```

```
/**
 * 점수가 가장 높은 학생을 찾아서 반환한다.
 * @return 점수가 가장 높은 학생
 */
public Student highest() {
    if(this.isEmpty()) {
        return null;
    }
    else {
        return this.highestRecursively(0, this.size()-1);
    }
}

/**
 * 점수가 가장 높은 학생을 재귀적으로 찾아서 반환한다.
 * @param left 가장 왼쪽 범위
 * @param right 가장 오른쪽 범위
 * @return 성적이 가장 낮은 학생
 */
private Student highestRecursively(int left, int right) {
    if(left == right) return this.elementAt(left);
    else {
        Student highestFromRights = highestRecursively(left +1, right);
        if(highestFromRights.compareTo(this.elementAt(left)) >= 0) {
            return highestFromRights;
        }
        else {
            return this.elementAt(left);
        }
    }
}

/**
 * 성적의 합계를 반환한다.
 * @return 성적의 합계
 */
public int sum() {
    if(this.isEmpty()) {
        return 0;
    }
    else {
        return this.sumOfScoresRecursively(0, this.size() -1);
    }
}


/**
 * 성적의 합계를 재귀적으로 반환한다.
 * @param left 성적의 합계를 구할 왼쪽 구간
 * @param right 성적의 합계를 구할 오른쪽 구간
 * @return 주어진 구간의 성적의 합계
 */
private int sumOfScoresRecursively(int left, int right) {
    int mid = (left + right) / 2;
    if(left == right) return this.elementAt(left).score();
    else {
        int leftSum = this.sumOfScoresRecursively(left, mid);
        int rightSum = this.sumOfScoresRecursively(mid +1, right);
        return (leftSum + rightSum);
    }
}
```

```
/**
 * 점수의 평균을 반환한다.
 * @return 평균 점수
 */
public double average() {
    if(this.isEmpty()) return 0;
    else return (((double) this.sum()) / ((double)this.size()));
}

/**
 * 점수가 평균 이상인 학생의 수를 출력한다.
 * @return 평균 이상의 점수를 가진 학생 수
 */
public int numberOfStudentsAboveAverage() {
    double average = this.average();
    int numberOfStudentsAboveAverage = 0;
    Iterator<Student> iterator = this.iterator();
    while (iterator.hasNext()) {
        Student student = iterator.next();
        if(student.score() >= average) {
            numberOfStudentsAboveAverage++;
        }
    }
    return numberOfStudentsAboveAverage;
}


/**
 * 두 순서의 값을 바꾼다
 * @param p 첫번째 순서
 * @param q 두번째 순서
 */
private void swap(int p, int q) {
    Student temp = this.elementAt(p);
    this.setElementAt(p, this.elementAt(q));
    this.setElementAt(q, temp);
}


/**
 * Quick Sort 를 위해 pivot 을 기준으로 구간을 분할한다.
 * @param left
 * @param right
 * @return pivot의 위치
 */
private int partition(int left, int right) {
    int pivot = left;
    int toRight = left;
    int toLeft = right+1;
    do {
        do {toRight++;} while(this.elementAt(toRight).score() < this.elementAt(pivot).score());
        do {toLeft--;} while(this.elementAt(toLeft).score() > this.elementAt(pivot).score());
        if(toRight < toLeft) {
            this.swap(toRight, toLeft);
        }
    }while(toRight < toLeft);
    this.swap(left, toLeft);
    return toLeft;
}
```

```
/**
 * 퀵 정렬을 수행한다.
 * @param left 왼쪽 시작 구간
 * @param right 오른쪽 끝 구간
 */
private void quicksortRecursively(int left, int right) {
    if(left < right) {
        int mid = this.partition(left, right);
        this.quicksortRecursively(left, mid-1);
        this.quicksortRecursively(mid+1, right);
    }
}

/**
 * 성적 순으로 정렬한다.
 */
public void sortByScore() {
    if(this.size() > 1) {
        int maxLoc = 0;
        for(int i = 1; i < this.size(); i++) {
            if(this.elementAt(i).score() > this.elementAt(maxLoc).score()) {
                maxLoc = i;
            }
        }
        this.swap(maxLoc, this.size()-1);
        this.quicksortRecursively(0, this.size()-2);
    }

}

/**
 * 각 학점에 해당하는 학생들을 센다.
 * @return 학생들의 수를 담고있는 GradeCounter 객체
 */
public GradeCounter countGrades() {
    GradeCounter gradeCounter = new GradeCounter();
    Iterator<Student> iterator = this.iterator();
    while( iterator.hasNext()){
        Student student = iterator.next();
        char grade = Ban.scoreToGrade(student.score());
        gradeCounter.count(grade);
    }
    return gradeCounter;
}
```

3. 종합 설명

해당 프로그램은 학생들의 성적을 입력받고, 프로그램 종료 시 학생들을 성적순으로 출력하며, 평균 점수와 최고, 최저점 등을 같이 출력해주는 프로그램이다.
앱을 실행하기 위해서는 '_DS07_Main_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

프로그램 장단점/특이점

1. 프로그램의 장점

MVC 패턴을 사용하였기에 유지보수성과, 이후 요구사항의 변경 시 코드의 변경의 파급효과가 적을 뿐더러 여러 List 클래스는 제네릭을 사용하여 다양한 객체들을 가방에 담을 수 있음과 동시에 잘못된 객체를 담게되었을 때 런타임이 아닌 컴파일 시예 예외가 발생하여 실수를 방지할 수 있다는 장점이 있다. 또한 제네릭의 상한 제한(extends)을 통해, Comparable을 구현한 객체만 사용할 수 있게끔 제한을 걸어 실수를 방지하는 것도 장점이라 볼 수 있다.

또한 Ban은 UnsortedArrayList를 상속받았기에 코드를 새롭게 작성하지 않고 재사용하였다는 장점이 있다.

실행 결과 분석

1. 입력과 출력

```
<<<학급 성적 처리를 시작합니다 >>>

성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 82
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 45
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 93
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 102
[오류] 0 보다 작거나 100 보다 커서, 정상적인 점수가 아닙니다.
- 점수를 입력하시오 (0..100): y
(오류) 정수가 입력되지 않았습니다.
- 점수를 입력하시오 (0..100): 66
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 87
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): -1
[오류] 0 보다 작거나 100 보다 커서, 정상적인 점수가 아닙니다.
- 점수를 입력하시오 (0..100): y
(오류) 정수가 입력되지 않았습니다.
- 점수를 입력하시오 (0..100): 70
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오:
```

```
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: k
! 성적 입력을 마칩니다.

[학급 성적 통계]
학급 학생 수:6
학급 최고 점수: 93
학급 최저 점수: 45
학급 평균: 73.8
평균 이상인 학생 수: 3

[학점별 학생수]
A 학점의 학생 수는 1 입니다.
B 학점의 학생 수는 2 입니다.
C 학점의 학생 수는 1 입니다.
D 학점의 학생 수는 1 입니다.
F 학점의 학생 수는 1 입니다.

[학생들의 성적순 목록]
점수:  45
점수:  66
점수:  70
점수:  82
점수:  87
점수:  93

<<< 학급 성적 처리를 종료합니다. >>>
```

```
- 점수를 입력하시오 (0..100): 1
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 1
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 1
성적을 입력하려면 'Y' 또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 점수를 입력하시오 (0..100): 1
(경고) 입력에 오류가 있습니다. 학급에 더이상 학생을 넣을 공간이 없습니다.
! 성적 입력을 마칩니다.

[학급 성적 통계]
학급 학생 수:10
학급 최고 점수: 1
학급 최저 점수: 1
```

2. 결과 분석

학생 수가 적어서 QuickSort를 통한 정렬이 순차 정렬에 비해 얼마나 빠른지 측정할 수는 없었으나, 기본적인 성적 관리 기능은 모두 정상적으로 작동한다.

생각해 볼 점에 대한 의견

1. Class “UnsortedArrayList”와 이를 상속받은 class “Ban” 의 관계를 이해하자. “Ban”이 “UnsortedArrayList”를 상속받도록 하는 장점은?

Ban이 UnsortedArrayList를 상속받게 만듦으로써, UnsortedArrayList에 정의된 ElementAt()등의 여러 메서드를 Ban에서 따로 정의하지 않고도 사용할 수 있다는 장점이 있다. 또한 따라서 Ban에서 추가로 구현해야 할 메서드들이 있는 경우에 기존 UnsortedArrayList의 메서드를 재활용함으로써, 개발 효율성등이 증가하는 장점이 있다.
또한 UnsortedArrayList를 상속받았다는 것 자체로도 Ban에서 List와 관련된 기능들을 하는 것을 쉽게 유추할 수 있다는 장점이 있다. 상속에 대한 여러 의견은 마지막 생각해 볼 점의 질문에 대한 답변에서 작성하도록 하겠다.

2.학번 등 학생 정보를 추가하여 더 입력 할 것들이 있을 경우, 프로그램에서 바뀌어야 하는 부분은?

당연하게도 Student 객체부터 바뀌어야 한다. 추가할 정보들을 Student 객체에 추가하고, compareTo도 목적에 따라서 다시 재정의 해주어야 한다. 또한 기존 프로그램에서는 작성하지 않았지만, equals와 hashCode에 대한 재정의도 추가되어야 한다.
Student 객체 뿐 아니라, AppController 등도 바뀌어야 한다. Student가 변화하였지만 그 영향이 AppController에도 영향을 끼치는 것이다.
MVC 구조를 따랐기에 변경의 영향이 매우 적은 범위에서 발생하는 것을 알 수 있으나, AppController에서 Student와 완전히 의존적으로 코드를 작성할 수 있는 방법이 없는지 알아보면 더 좋을 것이다.

3.재귀적이지 않은 문제 풀이에 비해, 재귀적 문제 풀이가 항상 좋은 성 능을 낸다고 할 수 있을까?

할 수 없다. 사실 기본적으로는 재귀함수 보다는 반복문을 이용하는 것이 빠르다. 함수를 호출하게 되면 함수에 대한 스택프레임(또는 Activation Records)이 계속 생성되며 쌓이기 때문에, 이를 생성하지 않는 반복문이 당연히 더 좋은 성능을 낼 것이다.
그러나 재귀함수는 사람이 이해하기 수월한 경우가 더 많다. 최댓값, 최솟값 찾기 정도야 단순 순회로 구현하는 경우 오히려 재귀함수가 더 이해하기 어려울 수 있으나, 퀵 정렬이나 피보나치 수 계산 등의 알고리즘을 구현할 때는 재귀함수를 이용하는 것이 더 사람이 이해하기 쉽고, 나중에 문제가 생겼을 때 쉽게 수정할 수 있다.

4. 상속 (Inheritance) 의 개념은? 객체의 설계도인 Class 의 재활용 면에서의 장점은?

상속이란 단어 그대로 부모의 필드와 메서드를 자식 클래스에서 물려받는 것이다. 그러나 private 접근 제어자를 가진 멤버변수는 상속이 불가능하다.
상속은 흔히 중복되는 코드를 줄이고, 작성한 코드를 재활용 할 수 있다는 장점이 있다고들 알려져 있다. 또한 상속을 통해 객체지향의 중요한 특성인 다형성을 구현할 수 있다.
이렇게만 보면 상속은 정말 좋을 기능 같지만 사실 그렇지 않다.
상속은 상위 클래스의 구현이 하위 클래스에 노출되기 때문에 캡슐화를 깨뜨린다.
캡슐화가 깨지기 때문에 상속은 부모 클래스와 자식 클래스가 강한 의존관계를 가지도록 하며, 이는 코드를 변경에 대처하기 어렵게 만든다. 부모 클래스의 변화는 이를 상속받은 모든 자식 클래스에 영향을 미치며, 심지어는 작동하지 않는 자식 클래스들이 생겨날 수 있다.
사실 자바를 조금만 더 공부해 보았다면 상속보다는 조합(Compoisition)을 사용하라는 말을 들어보았을 것이다.
이번에 내가 구현한 프로젝트를 예시로 들자면, Ban에서 UnsortedArraList를 상속받는 대신에, 이를 필드로 사용하는 것이다.
즉 조합이란 기존에 존재하던 클래스를 새로운 클래스의 구성요소로 사용하는 것인데, 이렇게 하면 다음과 같은 장점들이 있다.
1. 캡슐화를 깨뜨리지 않는다. 메서드를 사용하는 것이 아닌 호출하는 방식으로 동작하기에, 이는 캡슐화를 위반하지 않으며, 객체지향에서 중요시 생각하는 객체들과의 협력이 잘 이루어 진 것이라 볼 수 있다.
2. UnsortedArrayList의 내부 구현에 대한 변화를 통해 받는 영향이 적어진다. UnsortedArrayList의 내부 구현이 어떻게 바뀌던간에 퍼블릭 메서드의 변경이 아닌 이상, 이를 사용하는 객체들은 받는 영향이 적어진다.

따라서 상속보다는 컴포지션을 사용하는 것이 더욱 권장되며, 상속은 부모클래스와 자식 클래스가 순수한 is-a 관계일때만 사용해야 한다.