# 자료구조 실습 보고서

[제 8주] : 스택 기본기능

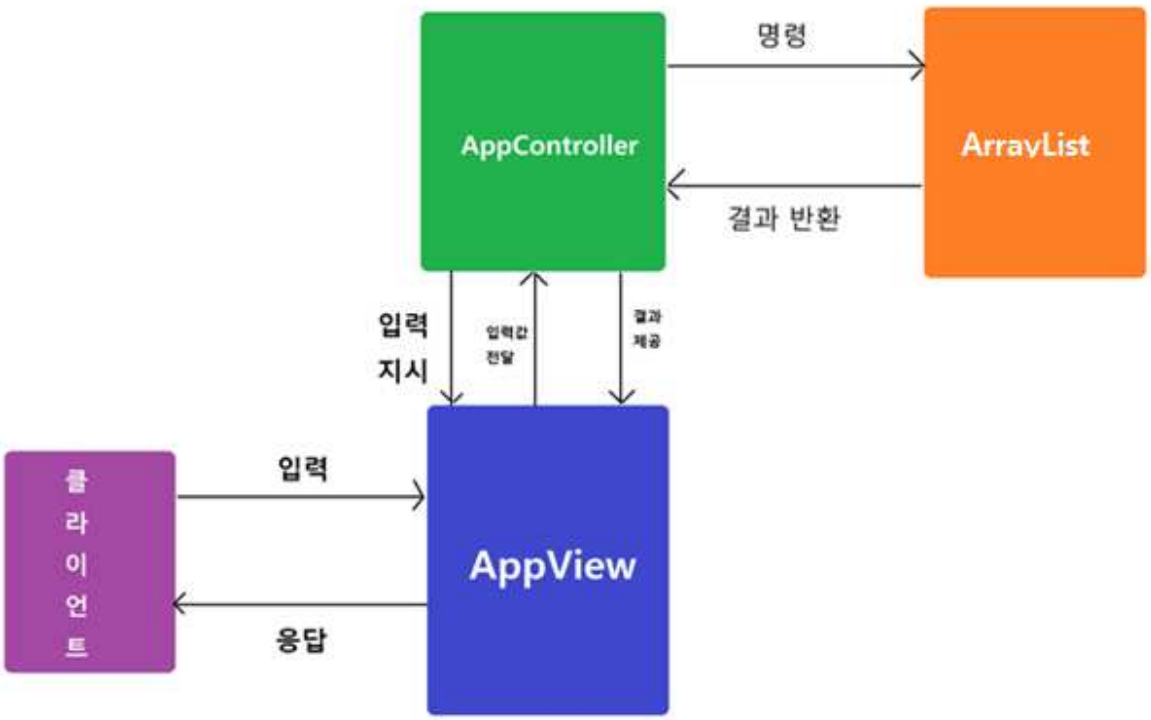
제출일: 2022-05-01(일)

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.

### AppView

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

### ArrayList

Stack 인터페이스를 구현하였으며 ArrayList와 Stack의 기능들을 수행한다.

## 2. 함수 설명( 주요 알고리즘/ 자료구조의 설명 포함 )

### 2-1. 사용된 자료구조

Stack 은 LIFO(last in first out) 자료구조로써, 가장 마지막에 들어온 원소를 가장 먼저 출력하는 자료구조이다,
Stack 은 push, pop, peek, clear 메서드를 지원하는데, 각각의 기능들은 다음과 같다.
push는 스택의 가장 마지막에 원소를 하나 추가한다.
pop은 가장 마지막에 들어온 원소를 반환하며 삭제한다.
peek는 가장 마지막에 들어온 원소를 반환만 한다.
clear은 스택에 존재하는 모든 원소를 삭제한다.
그리고 Stack 인터페이스는 ArrayList를 사용하여 구현하였다.
ArrayList는 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.

## 2-2. 주요 함수

### Stack 인터페이스

```
        /**
         * 주어진 원소를 Stack 의 맨 마지막에 추가한다.
         * @param anElement 추가할 원소
         * @return 성공하면 true
         */
        public boolean push(E anElement);


        /**
         * Stack 에 가장 마지막으로 들어온 원소를 삭제한다.
         * @return 삭제한 원소
         */
        public E pop();

        /**
         * Stack 에 가장 마지막으로 들어온 원소를 보여준다
         * @return 가장 마지막으로 삽입된 원소
         */
        public E peek();

        /**
         * Stack 을 비운다
         */
        public void clear();
```

### ArrayList<E extends Comparable<E>> implements Stack<E>

```
    /**
     * 리스트가 비었는지 확인한다.
     * @return 비었으면 true
     */
    @Override
    public boolean isEmpty() {
          return (this.size() == 0);
    }

    /**
     * 리스트가 가득 찼는지 확인한다.
     * @return 가득 찼으면 true
     */
    @Override
    public boolean isFull() {
        return (this.size() == this._capacity);
     }

    /**
     * 주어진 원소가 존재하는지 확인한다.
     * @param anElement 존재여부를 확인할 원소
     * @return 존재한다면 true
     */
    public boolean doesContain(E anElement) {
     return (this.orderOf(anElement) >= 0);
}
```

```
        /**
         * 주어진 원소가 들어있는 순서를 받환한다.
         * @param anElement 순서를 찾을 원소
         * @return 없다면 -1
         */
    public int orderOf(E anElement) {
        return IntStream.range(0, this.size())
                                    .filter(i-> this.elements()[i].equals(anElement))
                                    .findFirst()
                                    .orElse(-1);
    }
        /**
         * 주어진 인덱스의 원소를 반환한다.
         * @param anOrder 원소의 인덱스
         * @return 인덱스에 해당하는 원서
         */
        public E elementAt(int anOrder) {
      return (anOrder < 0 || anOrder >= this.size())
                                        ? null
                                        : this.elements()[anOrder];
}

        /**
         * 주어진 순서에 원소를 배치한다.
         * @param anOrder 순서
         * @param anElement 원소
         */
  protected void setElementAt(int anOrder, E anElement) {
      if (anOrder >= 0 && anOrder < this.size()) {
          this.elements()[anOrder] = anElement;
      }
}


        /**
         * 맨 처음 위치에 원소를 삽입한다.
         * @param anElement 삽입할 원소
         * @return 성공하면 true
         */
public boolean addToFirst(E anElement) {
      if (this.isFull()) {
           return false;
      }

          this.makeRoomAt(0);
          this.elements()[0] = anElement;
          this.setSize(this.size() + 1);
          return true;
}
```

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

        /**
         * 임의의 위치(여기서는 마지막)에 원소를 삽입한다.
         * @param anElement 임의의 위치
         * @return 성공하면 true
         */
public boolean add(E anElement) { // 마지막 자리 삽입
    return this.addToLast(anElement);
}

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

```
        /**
         * 임의의(여기서는 맨 마지막) 원소를 삭제한다
         * @return 삭제한 원소
         */
public E removeAny() {
    return this.removeLast();
}
        /**
         * 주어진 원소를 삭제한다.
         * @param anElement 삭제할 원소
         * @return 성공하면 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;
    }
}
@Override
public boolean push(E anElement) {
    return this.addToLast(anElement);
}

@Override
public E pop() {
    return this.removeLast();
}

@Override
public E peek() {
        if (this.isEmpty()) {
                return null;
        }
        else {
                return this.elementAt(this.size() - 1);
        }
}
@Override
public void clear() {

        final Object[] es = this.elements();
        int size = this.size();
        for (int to = size, i = size = 0; i < to; i++){
                es[i] = null;
        }

}
```

## AppController

```
	/**
	 * 스택의 모든 원소를 Bottom 부터 출력한다.
	 */
	private void showAllFromBottom() {
		//스택의 모든 원소를 bottom 부터 Top 까지 출력한다.
		AppView.output("[Stack] <Bottom> ");
		for(int order = 0; order < this.stack().size(); order++) {
			AppView.output(this.stack().elementAt(order).toString()+ " ");
		}
		AppView.outputLine(" <Top>");
	}

	/**
	 * 스택의 모든 원소를 Top 부터 출력한다.
	 */
	private void showAllFromTop() {
		//스택의 모든 원소를 Top 부터 Bottom 까지 출력한다.
		AppView.output("[Stack] <Top> ");
		for(int order = this.stack().size()-1; order > -1; order--) {
			AppView.output(this.stack().elementAt(order).toString()+ " ");
		}
		AppView.outputLine(" <Bottom>");

	}


	/**
	 * Top 원소를 출력한다.
	 */
	private void showTopElement() {

		if(this.stack().isEmpty()) {
			AppView.outputLine("[Top.Empty] 스택이 비어서 Top 원소가 존재하지 않습니다.");
			return;
		}

		Character peekChar = this.stack().peek();
		if(peekChar == null) {
			AppView.outputLine("(오류) 스택에서 해당하는 위치의 원소가 존재하지 않습니다.");
			return;
		}

		AppView.outputLine("[Top] 스택의 Top 원소는 '" + peekChar + "' 입니다.");
	}
```

```
/**
 * Stack 의 size 를 출력한다.
 */
private void showStackSize() {
		AppView.outputLine("[Size] 스택에는 현재 %d 개의 원소가 있습니다.".formatted(this.stack().size()) );

}


/**
 * Stack 을 사용한 통계 기록을 출력한다.
 */
private void showStatistics() {
		AppView.outputLine("");
		AppView.outputLine(" <스택 사용 통계>");
		AppView.outputLine("- 입력된 문자는 " + this.inputChars() + " 개 입니다.");
		AppView.outputLine
		("- 정상 처리된 문자는 " + (this.inputChars()-this.ignoredChars()) + "개 입니다." );
		AppView.outputLine("- 무시된 문자는 " + this.ignoredChars() + " 개 입니다.");
		AppView.outputLine("- 삽입된 문자는 " + this.pushedChars() + " 개 입니다.");
}


//== 스텍 수행 관련 ==//
/**
 * Stack 의 Push 를 구현한다
 * @param aCharForAdd Push할 원소
 */
private void pushToStack (char aCharForAdd) {
		if(this.stack().isFull()) {
				AppView.outputLine
						("(오류) 스택이 꽉차서, 더 이상 넣을 수 없습니다.");
		}
		else {
				Character charObjectForAdd = Character.valueOf(aCharForAdd);
				if(this.stack().push(charObjectForAdd)) {
						AppView.outputLine
								("[Push] 삽입된 원소는 '" + aCharForAdd + "' 입니다.");
				}
				else {
						AppView.outputLine("(오류) 스택에 넣는 공안에 오류가 발생하였습니다.");
				}
		}
}

/**
 * Stack 의 Pop 을 구현한다.
 */
private void popOne () {
		if(this.stack().isEmpty()) {
				AppView.outputLine("[Pop.Empty] 스택에 삭제할 원소가 없습니다.");
		}
		else {
				Character poppedChar = this.stack().pop();
				if(poppedChar == null) {
						AppView.outputLine("(오류) 스택에서 삭제하는 동안에 오류가 발생하였습니다.");
				}
				else {
						AppView.outputLine("[Pop] 삭제된 원소는 '" + poppedChar + "' 입니다.");
				}
		}
}
```

```
/**
 * 입력받은 수 만큼의 원소를 Stack 에서 Pop 한다
 * @param numberOfCharsToBePopped 개수
 */
private void popN (int numberOfCharsToBePopped) {
		if(numberOfCharsToBePopped == 0) {
			AppView.outputLine("[Pop] 삭제할 원소의 개수가 0 개 입니다.");
		}
		else {
			int count = 0;
			while(count < numberOfCharsToBePopped && (!this.stack().isEmpty())) {
				Character poppedChar = this.stack().pop();
				if(poppedChar == null) {
					AppView.outputLine("(오류) 스택에서 삭제하는 동안에 오류가 발생하였습니다.");
				}
				else {
					AppView.outputLine("[Pops] 삭제된 원소는 '" + poppedChar + "' 입니다.");
				}
				count++;
			}
			if(count < numberOfCharsToBePopped) {
				//model.Stack has become empty before we remove N elements
				AppView.outputLine("[Pops.Empty] 스택에 더이상 삭제할 원소가 없습니다.");
			}
		}
}

/**
 * 스택을 비우고 사용을 종료한다.
 */
private void quitStackProcessing() {
		AppView.outputLine("");
		AppView.outputLine(" <스택을 비우고 사용을 종료합니다.>");
		this.showAllFromBottom();
		this.popN(this.stack().size());
}


/**
 * 문자를 입력받는다.
 * @return 입력받은 문자
 */
private char inputChar() {
		AppView.output("? 문자를 입력하시오: ");
		return AppView.inputChar();
}
```

```
/**
 * 프로그램을 실행한다.
 */
public void run() {
		AppView.outputLine("<<< 스택 기능 확인 프로그램을 시작합니다 >>>");
		AppView.outputLine("");

		char input = this.inputChar();
		while(input != '!') {
			this.countInputChar();
			if(Character.isAlphabetic(input)) {
				this.pushToStack(input);
				this.countPushedChar();
			}
			else if(Character.isDigit(input)) {
				this.popN(Character.getNumericValue(input));
			}
			else if(input == '-') {
				this.popOne();
			}
			else if(input == '#') {
				this.showStackSize();
			}
			else if(input == '/') {
				this.showAllFromBottom();
			}
			else if(input == '\\') {
				this.showAllFromTop();
			}
			else if(input == '^') {
				this.showTopElement();
			}
			else {
				AppView.outputLine("[Ignore] 의미 없는 문자가 입력되었습니다.");
				this.countIgnoredChar();
			}
			input = this.inputChar();
		}
		this.quitStackProcessing();

		this.showStatistics();
		AppView.outputLine("");
		AppView.outputLine("<<< 스택 기능 확인 프로그램을 종료합니다 >>>");

}
```

## 3. 종합 설명

해당 프로그램은 Stack에 원소를 저장하고, 이에 대한 작업을 수행하는 프로그램이다..
앱을 실행하기 위해서는 '_DS08_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 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.

.

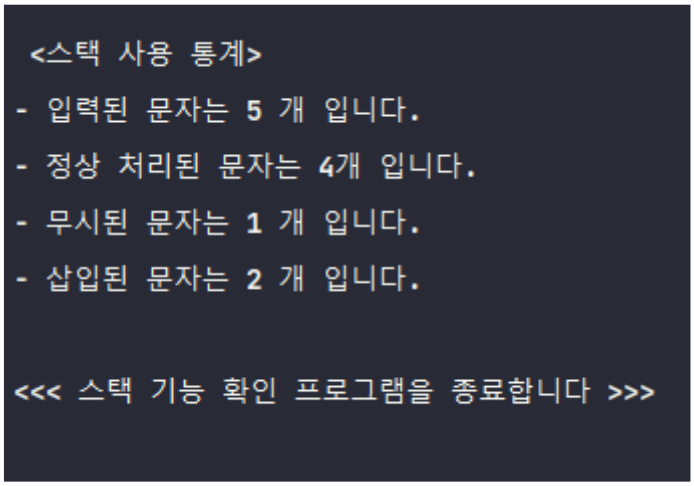
# 실행 결과 분석

## 1. 입력과 출력

```
? 문자를 입력하시오: A
[Push] 삽입된 원소는 'A' 입니다.
? 문자를 입력하시오: x
[Push] 삽입된 원소는 'x' 입니다.
? 문자를 입력하시오: h
[Push] 삽입된 원소는 'h' 입니다.
? 문자를 입력하시오: /
[Stack] <Bottom> A x h  <Top>
? 문자를 입력하시오: #
[Size] 스택에는 현재 3 개의 원소가 있습니다.
? 문자를 입력하시오: W
[Push] 삽입된 원소는 'W' 입니다.
? 문자를 입력하시오: z
[Push] 삽입된 원소는 'z' 입니다.
? 문자를 입력하시오: p
(오류) 스택이 꽉차서, 더 이상 넣을 수 없습니다.
? 문자를 입력하시오: -
[Pop] 삭제된 원소는 'z' 입니다.
? 문자를 입력하시오: \
[Stack] <Top> W h x A  <Bottom>
? 문자를 입력하시오: ^
[Top] 스택의 Top 원소는 'W' 입니다.
? 문자를 입력하시오: 5
[Pops] 삭제된 원소는 'W' 입니다.
[Pops] 삭제된 원소는 'h' 입니다.
[Pops] 삭제된 원소는 'x' 입니다.
[Pops] 삭제된 원소는 'A' 입니다.
[Pops.Empty] 스택에 더이상 삭제할 원소가 없습니다.
```

```
? 문자를 입력하시오: ^
[Top.Empty] 스택이 비어서 Top 원소가 존재하지 않
? 문자를 입력하시오: B
[Push] 삽입된 원소는 'B' 입니다.
? 문자를 입력하시오: e
[Push] 삽입된 원소는 'e' 입니다.
? 문자를 입력하시오: !

 <스택을 비우고 사용을 종료합니다.>
[Stack] <Bottom> B e  <Top>
[Pops] 삭제된 원소는 'e' 입니다.
[Pops] 삭제된 원소는 'B' 입니다.

 <스택 사용 통계>
- 입력된 문자는 15 개 입니다.
- 정상 처리된 문자는 15개 입니다.
- 무시된 문자는 0 개 입니다.
- 삽입된 문자는 8 개 입니다.

<<< 스택 기능 확인 프로그램을 종료합니다 >>>
```

```
<<< 스택 기능 확인 프로그램을 시작합니다 >>>

? 문자를 입력하시오: A
[Push] 삽입된 원소는 'A' 입니다.
? 문자를 입력하시오: x
[Push] 삽입된 원소는 'x' 입니다.
? 문자를 입력하시오: &
[Ignore] 의미 없는 문자가 입력되었습니다.
? 문자를 입력하시오: \
[Stack] <Top> x A  <Bottom>
? 문자를 입력하시오: -
[Pop] 삭제된 원소는 'x' 입니다.
? 문자를 입력하시오: !
```




## 2. 결과 분석

모든 기능이 문제 없이 잘 작동한다.

# 생각해 볼 점에 대한 의견

## 1.Abstractclass 와interface 의 유사점과 차이점은?

### 특별히, Abstractclass 를 상속 받는 것과 interface를 구현하는 것의 유사점과 차이점은?

Abstract class 와 interface 모두 추상 메서드를 가지며, 인스턴스로 생성할 수 없다는 공통점이 있다.
자바 8 버전 이전에는 인터페이스에서는 메서드를 구현할 수 없었고, 추상 클래스에서는 구현할 수 있었으나,
자바 8 이후부터는 인터페이스에서도 default 메서드를 지원하여 메서드를 구현할 수 있게 되었다.

추상 클래스를 사용하는 목적은 추상 클래스를 상속 받아서 기능을 이용하고, 거기에 추가하는 것이며,
인터페이스를 사용하는 목적은 인터페이스의 구현 객체가 같은 동작을 하도록 보장하는 것이다.

즉 인터페이스는 Has-A 관계이고, 추상 클래스는 Is-A 관계일 때 사용한다.

기본적으로 인터페이스와 abstract class는 비슷한 역할을 하지만, 큰 차이점이 존재한다.
결론부터 말하자면 abstract class 보다는 인터페이스를 사용하는 것을 권장한다.

인터페이스는 기존에 작성된 클래스에도 손쉽게 새로운 인터페이스를 구현하도록 추가할 수 있다.
기존에 작성한 클래스가 다른 클래스를 상속받고 있다면 자바는 단일 상속만을 지원하기 때문에 추상 클래스는 상속받을 수 없다.
그러나 인터페이스는 다중 구현을 지원한다.
따라서 기존에 작성한 클래스가 어떠한 클래스를 상속하거나 구현하였는지에 관계없이 새로운 인터페이스를 구현하도록 할 수 있다.

또한 인터페이스와, 인터페이스를 구현한 추상 골격 구현(skeletal implementation) 클래스를 함께 제공함으로써, 인터페이스와 추상 클래스의 장점을 모두 취할 수 있다.
인터페이스로는 타입을 정의하고, 필요하다면 디폴트 메서드를 구현해 놓을 수 있다.
그리고 추상 골격 구현 클래스는 인터페이스의 나머지 매서드들을 구현한다. (이때 모든 메서드를 구현할 필요는 없다.)
이렇게 한다면 골격 구현 클래스를 상속받아 확장하는 것만으로도 해당 인터페이스를 구현하는데 필요한 일이 대부분 완료된다.
이러한 방법을 템플릿  메서드 패턴(GOF)이라 부른다.
이러한 추상 골격 구현 클래스의 대표적인 예시로는 java.util의 List<E>를 구현한 AbstractCollection<E>과 AbstractList<E>이다

결론은 다음과 같다.
추상 클래스보다는 인터페이스를 사용하는 것을 우선시 하는것이 좋다.
다중 구현을 지원함은 물론 재사용성 측면이나, 변경에 대처하는 유연성, 그리고 객체지향에서 가장 중요하다 생각하는 다형성 측면에서도 인터페이스를 우선하는 것이 적절하다.