자료구조: 2022년 1학기 [강의]

스택 (Stack)

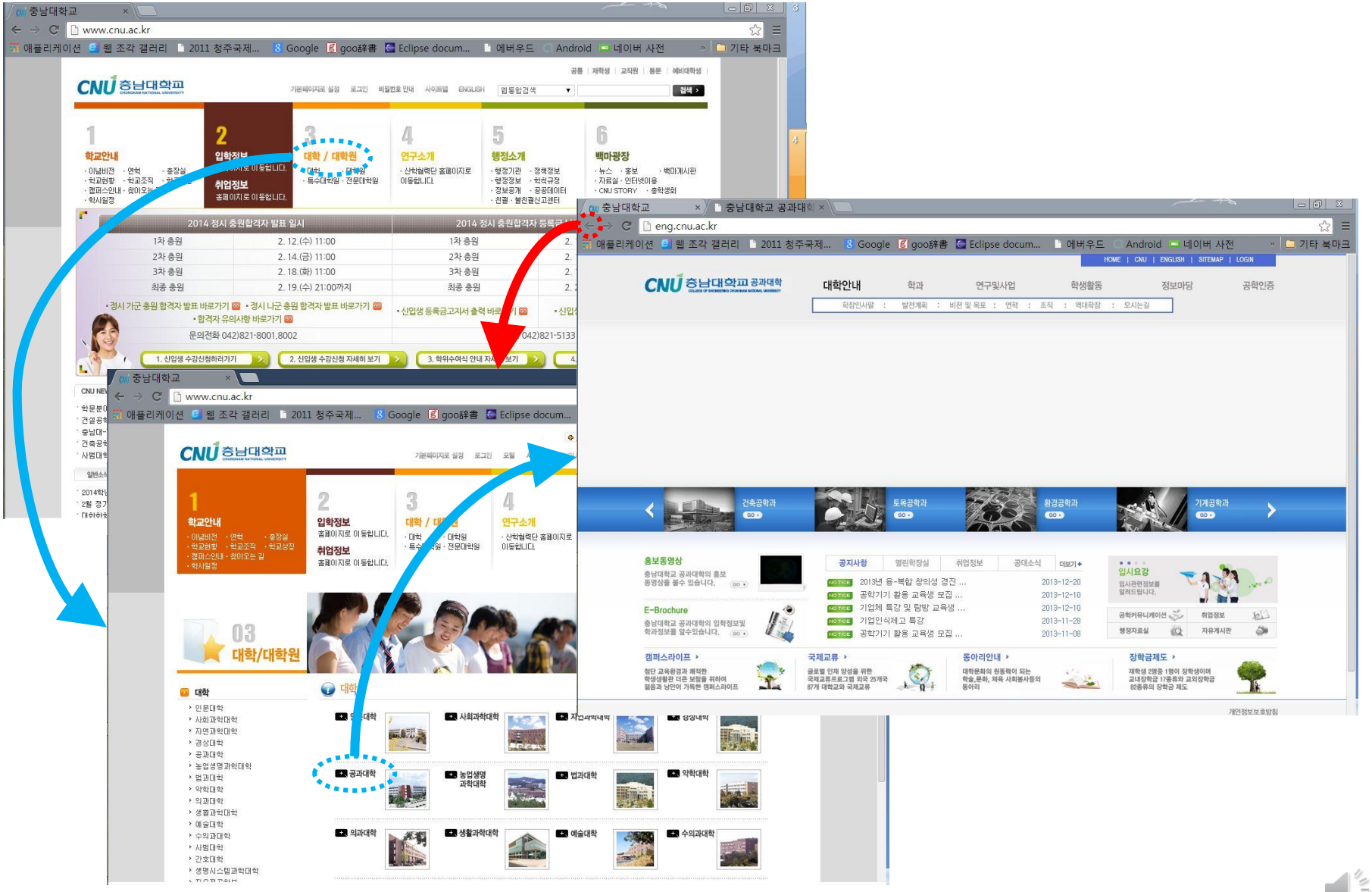
강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

스택(Stack) 이란?

스택 (Stack)

- 더미(stack): 쌓여 있는 것들
- 쌓인 순서가 있다
- 새로운 것을 쌓기 편한 곳은?
- 스택에서 하나를 빼 내기 편한 곳은?

웹 브라우징에서의 Navigation History 관리



쓸모 [1]

- 웹 브라우저: 방문 History 관리
  - 링크 클릭: 현재의 주소를 "방문 History 스택"에 쌓는다.
  - Back 단추 클릭: 스택에서 직전에 쌓은 주소를 꺼낸다.
- 수식 계산기
- 컴파일러(Compiler) / 인터프리터(Interpreter)
  - 고급 언어를 기계어로 번역 (compile) / 실행 (interpret)
  - 구분 분석 (Parsing)
    - 컴파일러는 수식을 포함하는, 보다 일반화된 복잡한 고급언어 표현을 분석하고 번역한다.

쓸모 [2]

재귀 함수의 처리

- Activation record: 현재 실행 중인 함수에서 다른 새로운 함수로의 call 이 발생하는 시점에, 스택에 쌓아서 보관하려는 새로운 함수의 실행 정보:
  - 새로운 함수로의 매개변수 전달 값 (Parameter Value)
  - 새로운 함수의 지역 변수들 (Local Data)
  - 새로운 함수가 종료 후에 return 즉 돌아가야 할, call 하는 함수 내의 위치 (return address)
- call: 새롭게 call 하는 함수의 activation record 를 스택에 쌓는다.
- return: 현재의 activation record 를 스택에서 꺼내어, 이전 call 했던 함수의 상태로 돌아간다.
- 스택의 top 에는 언제나, 현재 실행 중인 함수의 activation record 가 존재하게 된다.

스택 (Stack)

- 원소의 삽입과 삭제가 순서 리스트의 한쪽 끝에서만 발생
  - 스택의 꼭대기(top): 삽입과 삭제가 발생하는 한쪽 끝
  - 사용자가 얻을 수 있는 유일한 원소는 가장 최근에 삽입된 원소

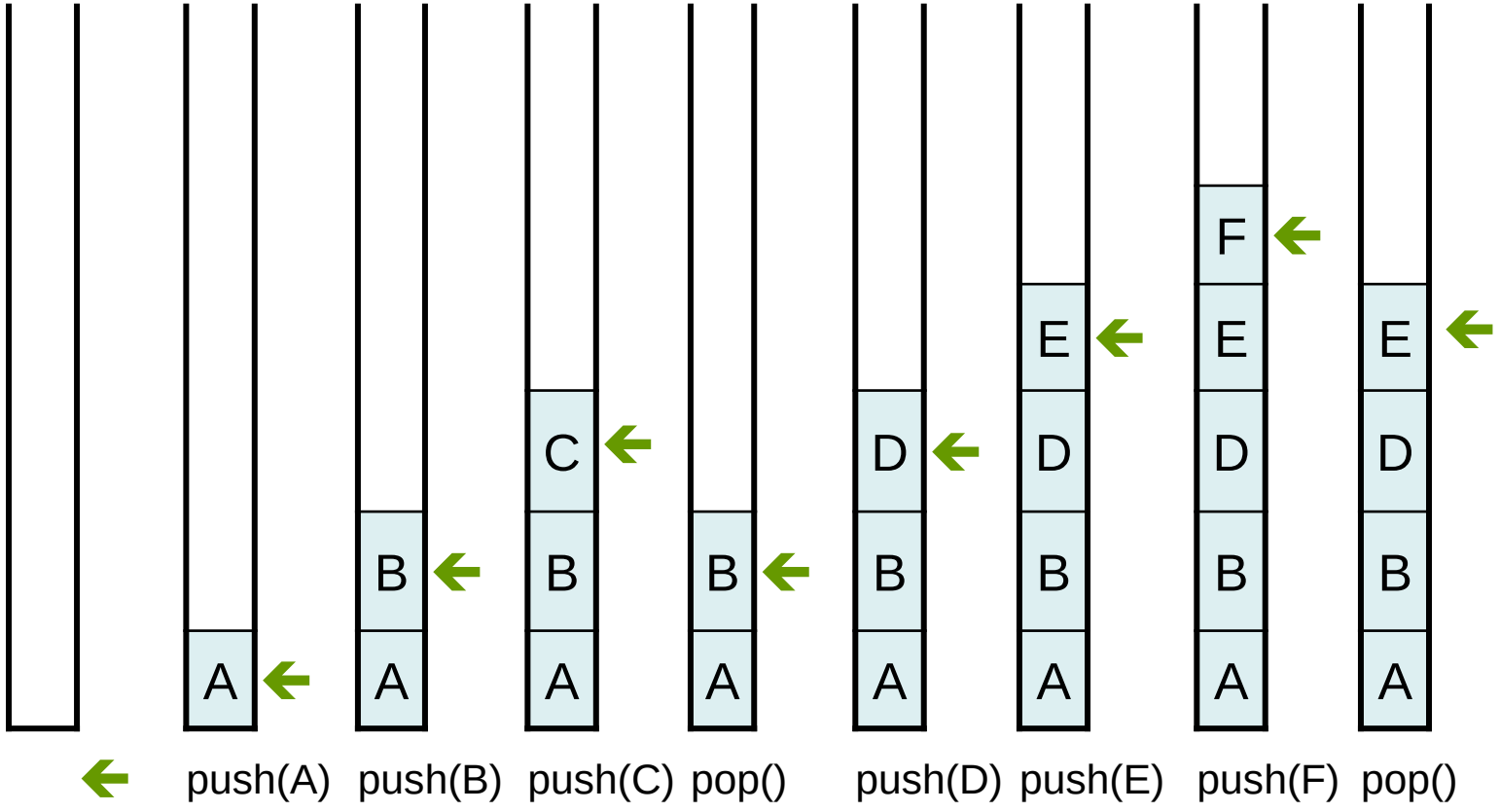
$$S = (e_0, \cdots, e_{n-1})$$

bottom 원소 top 원소

- 후입선출 (後入先出) 리스트
  - LIFO (last-in-first-out) List

삽입(push)과 삭제(pop)

"←" : 스택의 꼭대기(top) 원소를 가리킨다



스택의 사용법 (공개함수)

```
public class  Stack<E>
{
    public              Stack() { } // 생성자

    public boolean      isEmpty() { }
    public boolean      isFull() { }
    public int          size() { }

    public boolean      push (E anElement) { }
    public E            pop() { }
    public E            peek() { }
    public void         clear() { }
}
```

- isEmpty() : 스택이 비어있는지를 알려 준다.
- isFull() : 스택의 꽉 차서 더 이상 삽입할 수 없는 상태인지를 알려준다.
- size() : 스택에 있는 원소의 수를 얻는다.
- push() : 주어진 원소를 스택의 맨 위에 올려 놓는다.
- pop() : 비어 있는 스택이 아니면, 가장 꼭대기의 원소를 빼내어 얻는다.
- peek() : 비어 있는 스택이 아니면, 가장 꼭대기의 원소를 얻는다. 스택은 변하지 않는다.
- clear() : 스택을 비운다.

Class "ArrayStack<E>"

Class ArrayStack<E> 의 공개함수

- Stack 객체 사용법을 Java 로 구체적으로 표현
  - public ArrayStack () { }
  - public ArrayStack (int GivenCapacity) { }
  - public boolean isEmpty() { }
  - public boolean isFull() { }
  - public int size() { }
  - public boolean push (E anElement) { }
  - public E pop() { }
  - public E peek() { }
  - public void clear() { }

Class "ArrayStack<E>" 의 구현

Class "ArrayStack"의 초기 형태는 이렇게!

```
public class  ArrayStack<E>
{
        public  ArrayStack ( )
        {
                // 수정해야 함
        }
        public  ArrayStack (int givenCapacity)
        {
                // 수정해야 함
        }

        public int    size ()
        {
                return  0 ; // 수정해야 함
        }
        public boolean  isEmpty ()
        {
                return  true ; // 수정해야 함
        }
        public boolean  isFull ()
        {
                return  true ; // 수정해야 함
        }
        public boolean  push (E anElement)
        {
                return  true ; // 수정해야 함
        }
        public E  pop()
        {
                return  null ; // 수정해야 함
        }
        public void  clear()
        {
                return ; // 수정해야 함
        }

} // End of Class "ArrayStack<E>"
```

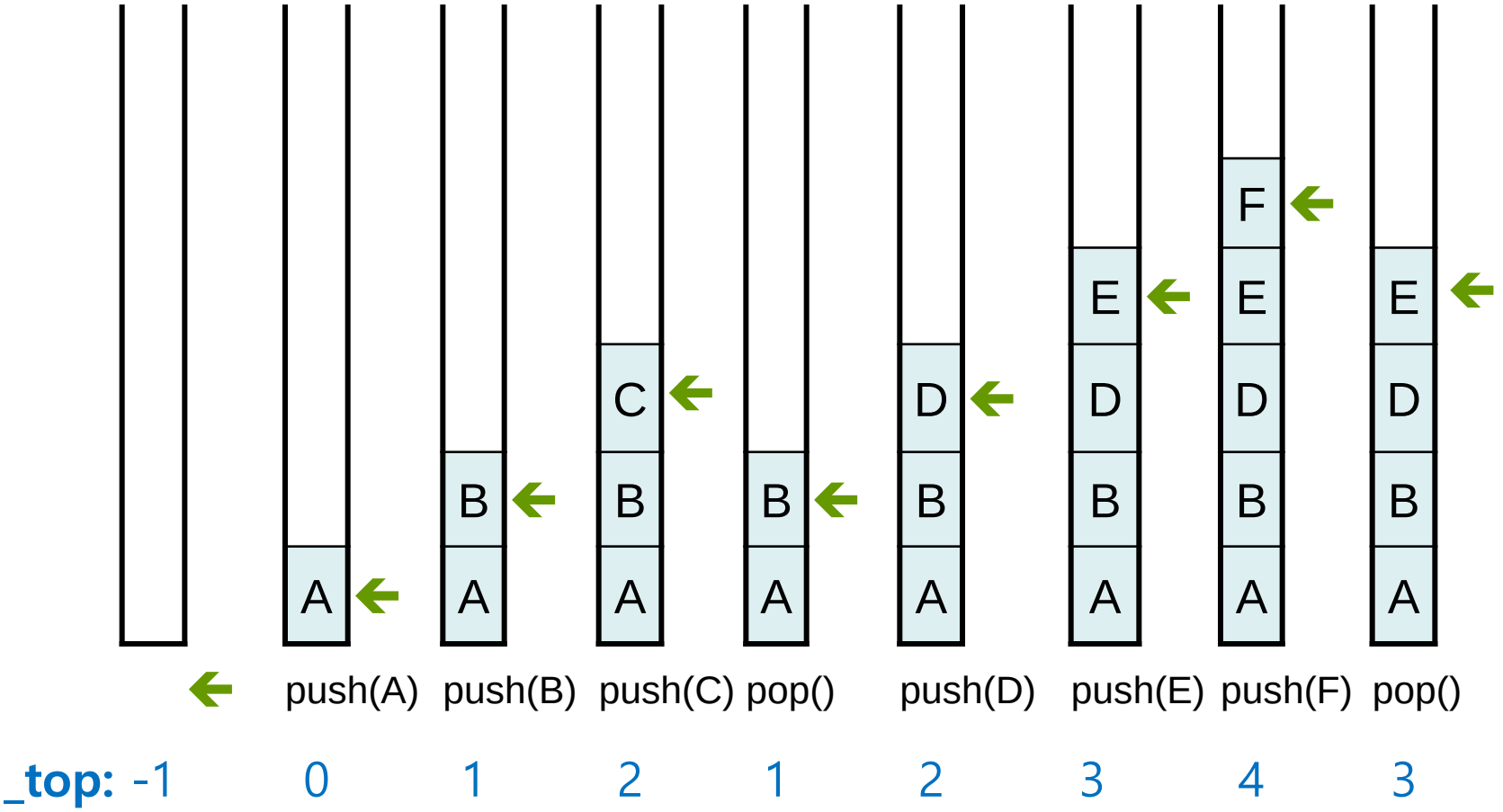
이렇게만 정의해 두어도
사용하는 곳에서 프로그래밍 하는 데는 전혀 지장이 없다.
즉 컴파일 오류가 발생하지 않는다.

Class "ArrayStack"의 구현: 멤버변수

```
public class  ArrayStack<E>
{
    // 비공개 상수
    private static final int  DEFAULT_CAPACITY = 50 ;

    // 비공개 멤버 변수
    private    int     _capacity ;
    private    int     _top ;
    private    E[]     _elements ;
```

- DEFAULT_CAPACITY: 사용자가 지정하지 않을 경우에 사용될 초기 스택 크기
- _capacity: 현재 스택의 최대 크기
- _top : 현재 스택의 top 원소의 배열에서의 위치
- _elements : 스택의 원소들이 저장되는 배열

배열로 구현된 스택에서의 삽입/삭제



일반화된 (generic) 클래스

```
public class  ArrayStack<E>
{
    .....
    // 비공개 멤버 변수
    private   int     _capacity ;
    private   int     _top ;
    private   E[]     _elements ;
```

- 스택의 선언에서 원소의 자료형(클래스)을 매개변수처럼 사용
  - E 는 구체화 되지 않은 상태의 object class 이다.
- ArrayStack<E>는 일반화된 클래스로 선언되었다.
  - 사용자가 필요한 스택 객체를 선언할 때, E 가 구체적으로 주어지게 된다.
- 목적: 클래스를 정의하는 코드의 재활용
  - 동일한 기능을 하는 스택을, 원소가 달라질 때마다 따로 정의할 필요는 없다.
  - 편의성, 효율성

일반화된 클래스의 사용법

```
public class  ArrayStack<E>
{
    .....
    // 비공개 멤버 변수
    private   int     _capacity ;
    private   int     _top ;
    private   E[]     _elements ;
```

- 일반화된 클래스 변수의 선언과 사용

```
ArrayStack<String> wordStack = new ArrayStack<String>() ;
    // 원소의 자료형이 String 인 ArrayStack 객체를 생성
    // 객체 변수 wordStack 이 생성된 스택 객체를 소유
wordStack.push ("Hello") ;
wordStack.push ("abc") ;
String  poppedString = wordStack.pop() ;
```

ArrayStack": Getter/Setter

```
public class  ArrayStack<E>
{
      // 비공개 상수
      private static final int  DEFAULT_CAPACITY = 50 ;

      // 비공개 멤버 변수
      private    int          _capacity ;
      private    int          _top ;
      private    E[]          _elements ;

      // Getter/Setter
      private int  capacity() { // Class 내부에서만 사용
            return  this._capacity ;
      }
      private void  setCapacity (int newCapacity) { // Class 내부에서만 사용
            this._capacity = newCapacity ;
      }

      private int  top() {
            return  this._top ;
      }
      private void  setTop (int newTop) { // Class 내부에서만 사용
            this._top = newTop ;
      }

      private E[]  elements () { // Class 내부에서만 사용
            return  this._elements ;
      }
      private void  setElements (E[] newElements) { // Class 내부에서만 사용
            this._elements = newElements ;
      }
```

ArrayStack: 생성자

```
public class  ArrayStack<E> {
    // 비공개 멤버 변수
    ......

    // 생성자
    @SuppressWarnings ("unchecked")
    public  ArrayStack ( )
    {
        this.setCapacity (ArrayStack.DEFAULT_CAPACITY) ;
        this.setElements ( (E[]) new Object[this.capacity()] ) ;
        this.setTop (-1) ;
    }

    @SuppressWarnings ("unchecked")
    public  ArrayStack (int givenCapacity)
    {
        this.setCapacity (givenCapacity) ;
        this.setElements ( (E[]) new Object[this.capacity()] ) ;
        this.setTop (-1) ;
    }
```

ArrayStack: 다른 생성자 활용

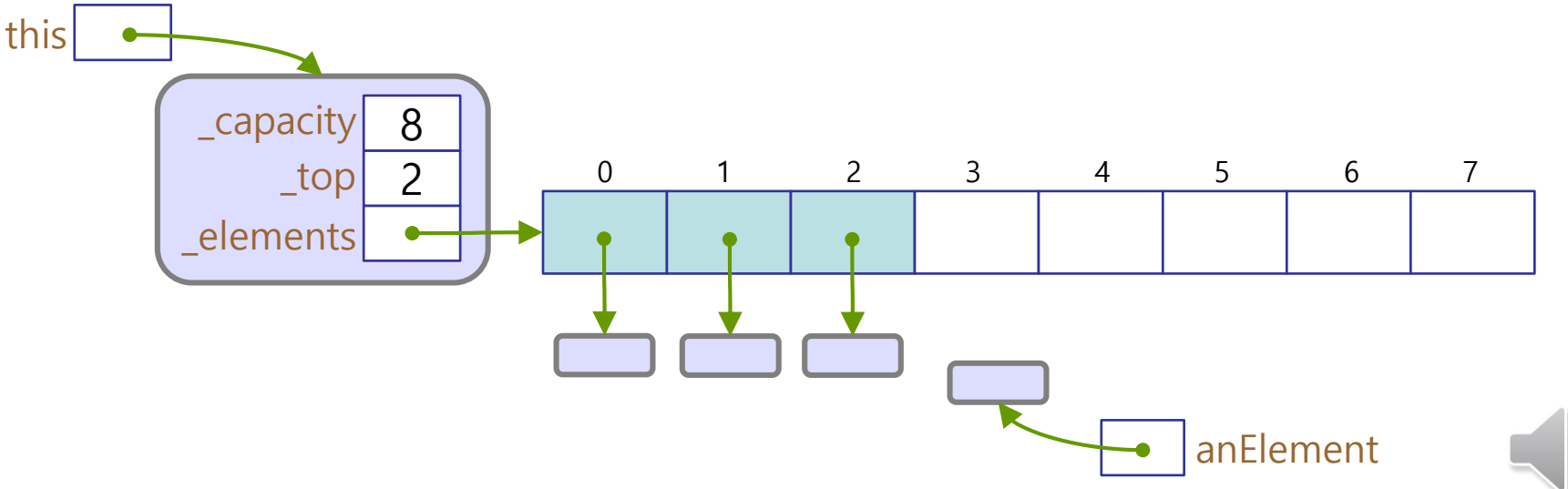
```
public class  ArrayStack<E> {
    // 비공개 멤버 변수
    ……

    // 생성자
    @SuppressWarnings ("unchecked")
    public  ArrayStack ( )
    {
        this (ArrayStack.DEFAULT_CAPACITY) ;


    }

    @SuppressWarnings ("unchecked")
    public  ArrayStack (int givenCapacity)
    {
        this.setCapacity (givenCapacity) ;
        this.setElements ( (E[]) new Object[this.capacity()] ) ;
        this.setTop (-1) ;
    }
```

ArrayStack: 상태 알아보기

```
public class  ArrayStack<E>
{
    ……

    // 비공개 함수
    ……

    // size
    public int  size()
    {
        return  (this.top() + 1) ;
    }

    // Stack이 비어 있는지 확인
    public boolean  isEmpty ()
    {
        return  (this.top() < 0) ;
    }

    // Stack이 꽉차 있는지 확인
    public boolean  isFull ()
    {
        return  ( (this.top()+1) == this.capacity() ) ;
    }
```

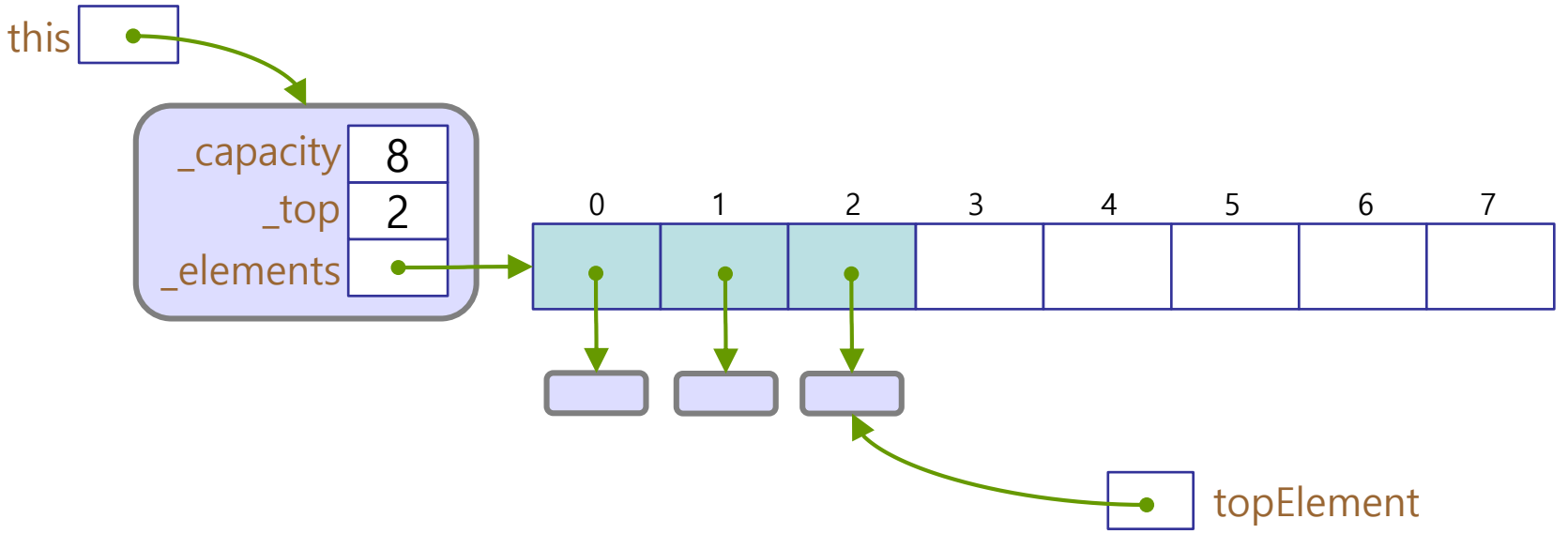
ArrayStack: push()

```
// Push
public boolean   push (E   anElement)
{
     if ( this.isFull() ) {
          return   false ;
     }
     else {
          this.setTop (this.top()+1) ;
          this.elements()[this.top()] = anElement ;
          return   true ;
     }
}
```



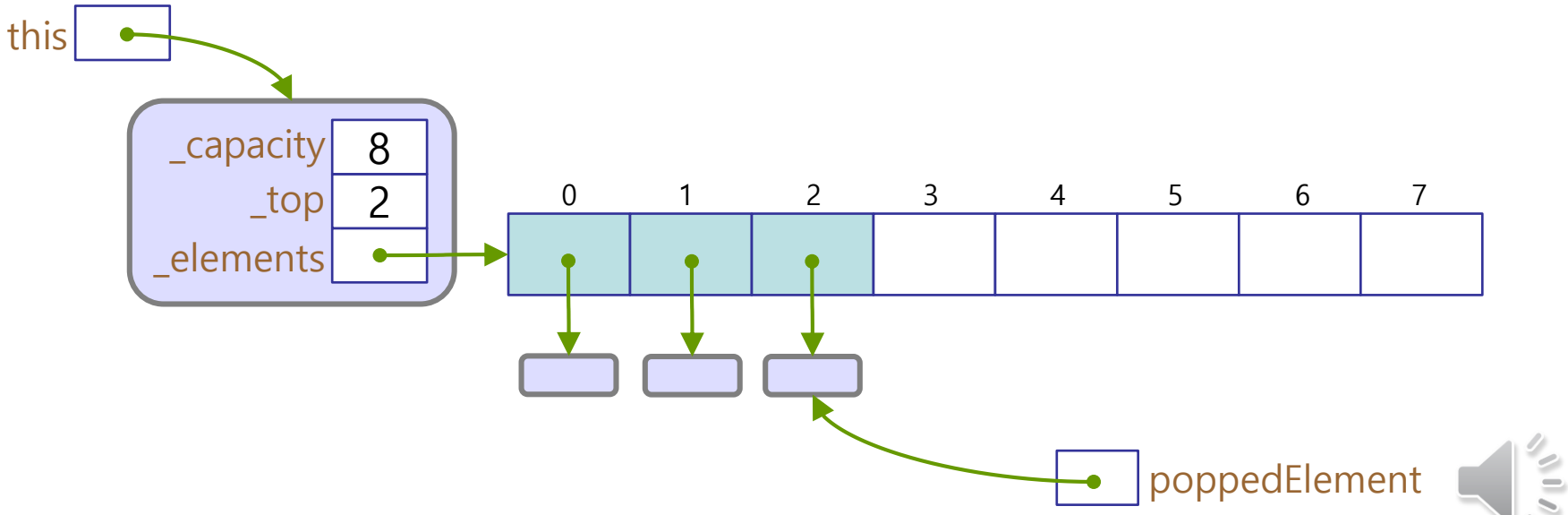
ArrayStack: peek()

```
// peek
public E   peek()
{
     E   topElement = null ;
     if ( ! this.isEmpty() ) {
          topElement = this.elements()[this.top()] ;
     }
     return   topElement ;
}
```



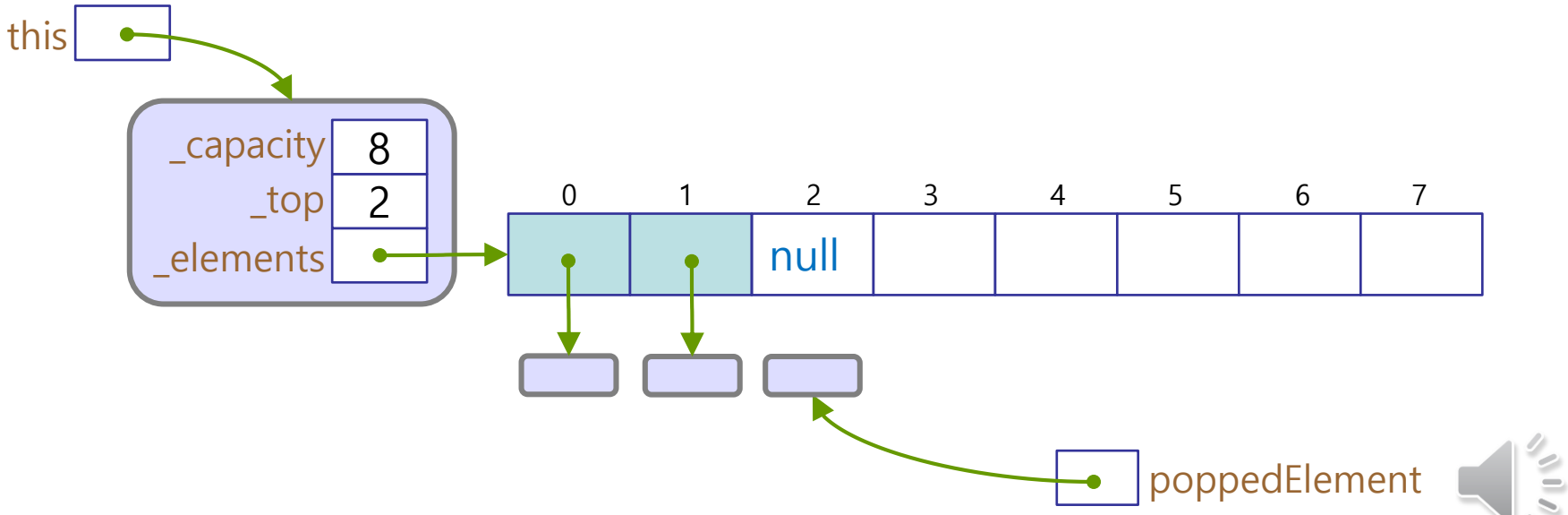
ArrayStack: pop() [1]

```
// pop
public E  pop()
{
    E  poppedElement = null ;
    if ( ! this.isEmpty() ) {
        poppedElement = this.elements()[this.top()] ;
        this.elements()[this.top()] = null ;
        this.setTop (this.top()-1) ;
    }
    return  poppedElement ;
}
```



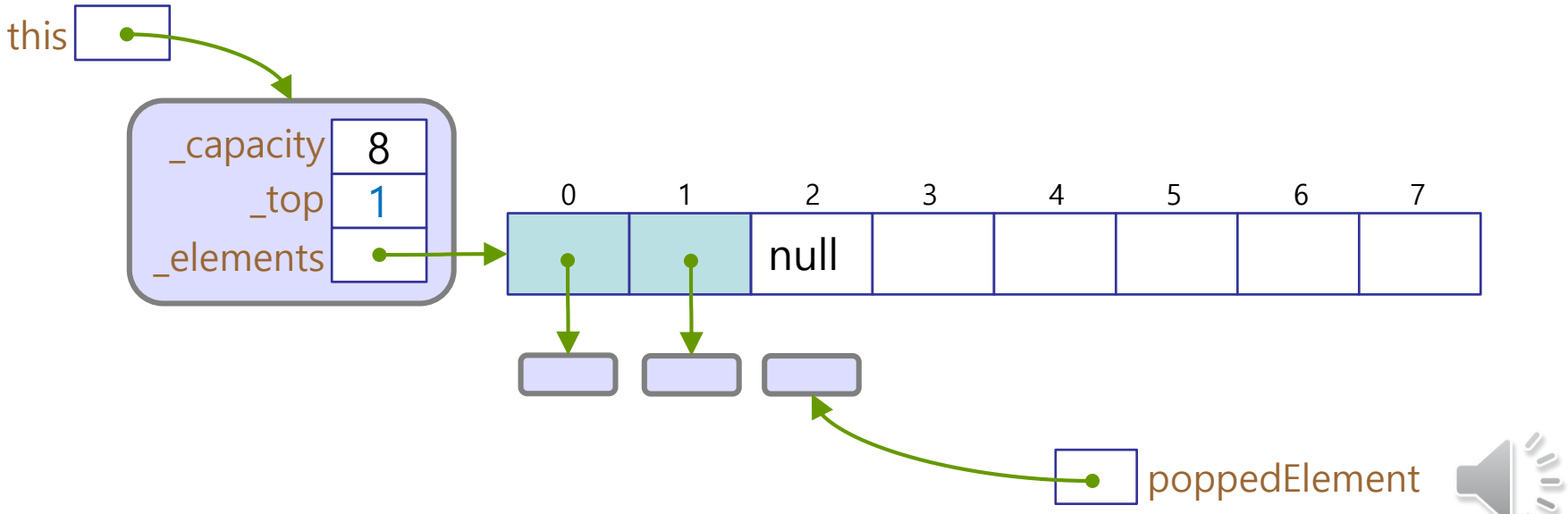
ArrayStack: pop() [2]

```
// pop
public E  pop()
{
    E  poppedElement = null ;
    if ( ! this.isEmpty() ) {
        poppedElement = this.elements()[this.top()] ;
        this.elements()[this.top()] = null ;
        this.setTop (this.top()-1) ;
    }
    return  poppedElement ;
}
```



ArrayStack: pop() [3]

```
// pop
public E  pop()
{
    E  poppedElement = null ;
    if ( ! this.isEmpty() ) {
        poppedElement = this.elements()[this.top()] ;
        this.elements()[this.top()] = null ;
        this.setTop (this.top()-1) ;
    }
    return  poppedElement ;
}
```



ArrayStack: clear()

```
// clear
public void   clear()
{
     while ( this.top() >= 0 ) {
          this.elements()[this.top()] = null ;
          this.setTop (this.top()-1) ;
     }
}
```

Full 처리 방법: resize()

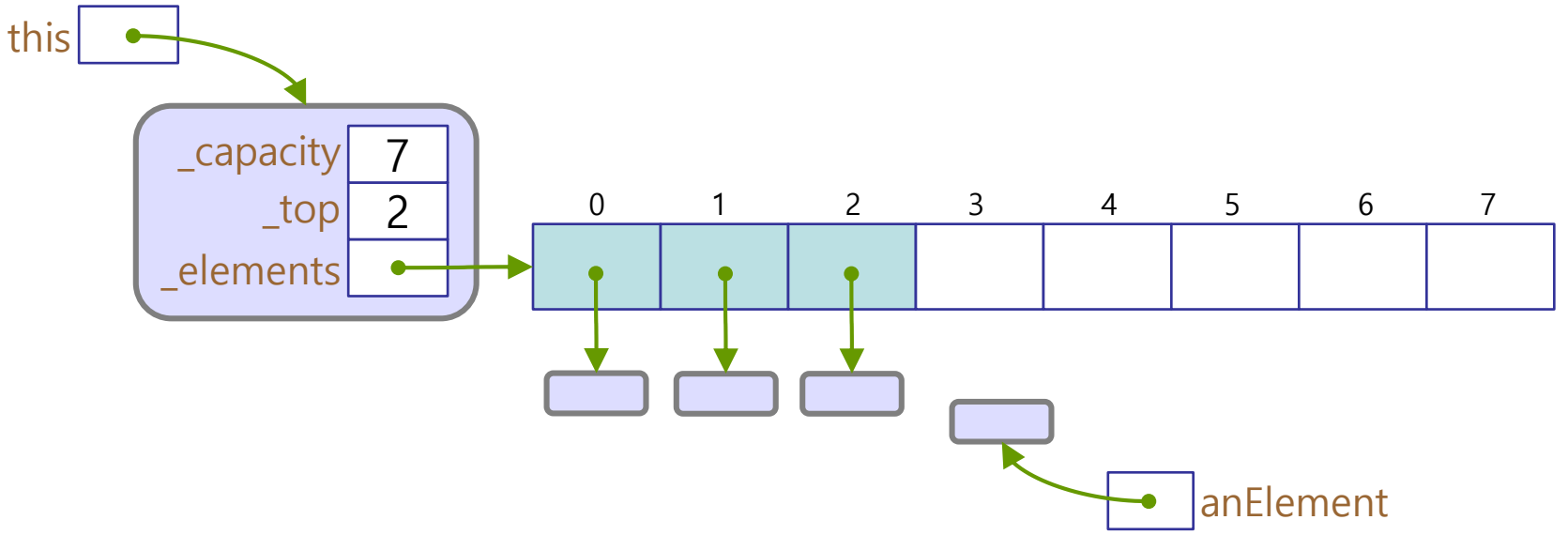
```
public class  ArrayStack<E>
{
    // 비공개 멤버 변수
    ......
    // 비공개 함수
    @SuppressWarnings ("unchecked")
    private void  resize()
    {
        // 배열의 크기를 2 배로 늘려준다.
        this.setCapacity (this.capacity() * 2) ;
        E[]  oldElements = this.elements() ;
        this.setElements ( (E []) new Object[this.capacity()] ) ;
        for ( int i = 0 ; i <= this.top() ; i++ ) {
            this.elements()[i] = oldElements[i] ;
            oldElements[i] = null ;
        }
        // this.setElements ( Arrays.copyOf(this.elements(), this.capacity()) ) ;
    }
```

Arrays.copyOf() 를 사용한 resize()

```
public class  ArrayStack<E>
{
    // 비공개 멤버 변수
    ......
    // 비공개 함수
    @SuppressWarnings ("unchecked")
    private void  resize()
    {
        // 배열의 크기를 2 배로 늘려준다.
        this.setCapacity (this.capacity() * 2) ;
        // E[]  oldElements = this.elements() ;
        // this.setElements ( (E []) new Object[this.capacity()] ) ;
        // for ( int i = 0 ; i <= this.top() ; i++ ) {
        //     this.elements()[i] = oldElements[i] ;
        //     oldElements[i] = null ;
        // }
        this.setElements ( Arrays.copyOf(this.elements(), this.capacity()) ) ;
    }
```

ArrayStack: push() // resize() 사용

```
// Push with resize()
public boolean   push(E anElement)
{
     if ( this.isFull() ) {
          this.resize() ;
     }
     this.setTop (this.top()+1) ;
     this.elements()[this.top()] = anElement ;
     return   true ;
}
```



Interface stack<E>

기존 class 의 특정 기능 확장은?

- Class “List<E>”에서 스택과 관련된 공개 함수
  - public boolean isEmpty () { }
  - public boolean isFull () { }
  - public boolean size () { }

  - public E elementAt (int order) { }
  - public E last () { }

  - public boolean addToLast (E anElement) { }
  - public E removeLast () { }
  - public void clear() { }

기존 class 의 특정 기능 확장은?

- Class "Stack<E>"의 공개 함수
  - public boolean isEmpty () { }
  - public boolean isFull () { }
  - public boolean size () { }

  - public E elementAt (int order) { }
  - public E last () { }

  - public boolean addToLast (E anElement) { }
  - public E removeLast () { }
  - public void clear() { }

  - public boolean push (E anElement) { }
  - public E pop () { }
  - public E peek () { }

기존 class 의 특정 기능 확장은?

- Stack 은 List 의 특수한 경우이다:
  - Stack 고유의 기능을 하도록 하는 공개함수가 추가되었다.
    - push() / pop() / peek()

- 이런 경우에 Stack 을 별도의 class 로 정의하는 대신, Java Interface 로 선언할 수 있다.
- 즉, Stack 고유의 공개함수를 모아서 Interface 로 선언한다.
- 그리고, class "List" 는 interface "Stack" 을 구현 (implements) 한다.

Java interface 로서의 스택

```
public interface  Stack<E>
{
    public abstract boolean   push (E anElement) ;
    public abstract E         pop() ;
    public abstract E         peek() ;
}
```

- Interface 는 단지 함수 header 만 정의한다.
  - 코드 구현은 없다.
  - 따라서, “abstract” 이다.
    그런데, "abstract" 가 당연하므로, 생략 가능하다.

Interface Stack<E>: Full Version

```
public interface  Stack<E>
{
    public abstract boolean   isEmpty() ;
    public abstract boolean   isFull() ;
    public abstract int       size() ;
    public abstract E         elementAt (int order) ;

    public abstract boolean   push (E anElement) ;
    public abstract E         pop() ;
    public abstract E         peek() ;

    public abstract void      clear() ;
}
```

Java interface 사용

```
public class  ArrayList<E> implements Stack<E>
{
    public boolean      isEmpty () {…}
    public boolean      isFull () {…}
    public boolean      size () {…}

    public E            elementAt (int order) {…}
    public E            last () {…}

    public boolean      addToLast (E anElement) {…}
    public E            removeLast () {…}
    public void         clear() {…}

    // 구현은 이곳 ArrayList 안에서
    public boolean      push (E anElement) {…}
    public E            pop () {…}
    public E            peek () {…}
}
```

Java interface 구현 [1]

```
public class  ArrayList<E> implements Stack<E>
{
        ……

        // 비공개 인스턴스 변수
        private int       _capacity ;
        private int       _size ;
        private E[]       _elements ;

        // Getter/Setter
        ……

        // 공개 함수
        @Override
        public boolean  isEmpty () {…}
        @Override
        public boolean  isFull () {…}
        @Override
        public boolean  size () {…}

        @Override
        public E    elementAt (int order) {…}
        public E    last () {…}

        public boolean  addToLast (E anElement) {…}
        public E        removeLast () {…}
        @Override
        public void     clear() {…}

        // 구현은 이곳 ArrayList 안에서
        @Override
        public boolean  push (E anElement) {
            if ( this.isFull() ) {
                return  false ;
            }
            else {
                this.elements()[this.size()] = anElement ;
                this.setSize (this.size()+1) ;
                return  true ;
            }
        }

        @Override
        public E  pop () {
            E  poppedElement = null ;
            if ( ! this.isEmpty() ) {
                this.setSize (this.size()-1);
                poppedElement = this.elements()[this.size()] ;
                this.elements()[this.size()] = null ;
            }
            return  poppedElement ;
        }

        @Override
        public E  peek () {
            E  topElement = null ;
            if ( ! this.isEmpty() ) {
                topElement = this.elements()[this.size()-1] ;
            }
            return  topElement ;
        }

} // End of class "ArrayList"
```

Java interface 구현 [2]: 기존의 함수를 활용

```
public class  ArrayList<E> implements Stack<E>
{
      ......

      // 비공개 인스턴스 변수
      private int       _capacity ;
      private int       _size ;
      private E[]       _elements ;

      // Getter/Setter
      ......

      // 공개 함수
      @Override
      public boolean  isEmpty () {...}
      @Override
      public boolean  isFull () {...}
      @Override
      public boolean  size () {...}

      @Override
      public E    elementAt (int order) {...}
      public E    last () {...}

      public boolean  addToLast (E anElement) {...}
      public E        removeLast () {...}
      @Override
      public void     clear() {...}
```

```
// 구현은 이곳 ArrayList 안에서
@Override
public boolean  push (E anElement) {
      return  this.addToLast(anElement) ;
}

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

@Override
public E  peek () {
     return  this.elementAt (this.size()-1) ;
}
```

```
} // End of class "ArrayList"
```

LinkedList 도 스택으로

```
public class  LinkedList<E> implements Stack<E>
{
      @Override
      public boolean       isEmpty () {...}
      @Override
      public boolean       isFull () {...}
      @Override
      public boolean       size () {...}

      @Override
      public E    elementAt (int order) {...}
      public E    last () {...}

      public boolean  addToLast (E anElement) {...}
      public E           removeLast () {...}
      @Override
      public void       clear() {...}

      // 구현은 이곳 LinkedList 안에서
      @Override
      public boolean  push (E anElement) {...}
      @Override
      public E           pop () {...}
      @Override
      public E           peek () {...}

} // End of Class LinkedList<E>
```

Head Element 가 Order 0 인 경우 [1]

```
public class  LinkedList<E> implements Stack<E>
{
      // 비공개 인스턴스 변수
      private int                 _size ;
      private LinkedNode<E>   _head ;

      // 공개 함수
      public boolean  isEmpty () {...}
      public boolean  isFull () {...}
      public boolean  size () {...}

      @Override
      public E    elementAt (int order) {
            if ( order < 0 || order >= this.size() ) {
                  return  null ;
            }
            else {
                  LinkedNode<E> currentNode = this.head() ;
                  int  count = 0 ;
                  while ( count < order ) {
                        currentNode = currentNode.next() ;
                        count++ ;
                  }
                  return  currentNode.element() ;
            }
      }

      public E    last () {
            return  this.elementAt (this.size()-1) ;
      }
```

```
      public boolean  addToLast (E anElement) {
            if ( this.isFull() ) {
                  return  false ;
            }
            else {
                  LinkedNode<E> nodeForAdd =
                        new LinkedNode<E>()  ;
                  nodeForAdd.setElement (anElement) ;
                  nodeForAdd.setNext (null) ;
                  if ( this.head() == null ) {
                        this.setHead (nodeForAdd) ;
                  }
                  else {
                        LinkedNode<E> currentNode =
                              this.head() ;
                        while ( currentNode.next() != null ) {
                              currentNode = currentNode.next() ;
                        }
                        currentNode.setNext (nodeForAdd) ;
                  }
                  this.setSize (this.size()+1) ;
                  return  true ;
            }
      }
```

Head Element 가 Order 0 인 경우 [2]

```
public E  removeLast () {...}
        if ( this.isEmpty() ) {
                return  null ;
        }
        else {
                E  removedElement = null ;
                if  ( this.size() == 1 ) {
                        removedElement =
                                this._head.element() ;
                    this.setHead (null) ;
                }
                else {
                        LinkedNode<E> previousNode =
                                this.head() ;
                        LinkedNode<E> currentNode =
                                this.head().next() ;
                        while (  currentNode.next() != null) {
                                previousNode = currentNode ;
                                currentNode =
                                        currentNode.next() ;
                        }
                        removedElement =
                                currentNode.element() ;
                        previousNode.setNext (null) ;
                }
                this.setSize (this.size()-1) ;
                return  removedElement ;
        }
}

@Override
public void  clear() {...}
```

```
        // Stack의 구현
        @Override
        public boolean  push (E anElement) {
                return  this.addToLast (anElement) ;
        }
        @Override
        public E  pop () {
                return  this.removeLast() ;
        }
        @Override
        public E  peek () {
                return  this.last() ;
        }
} // End of class "LinkedList"
```

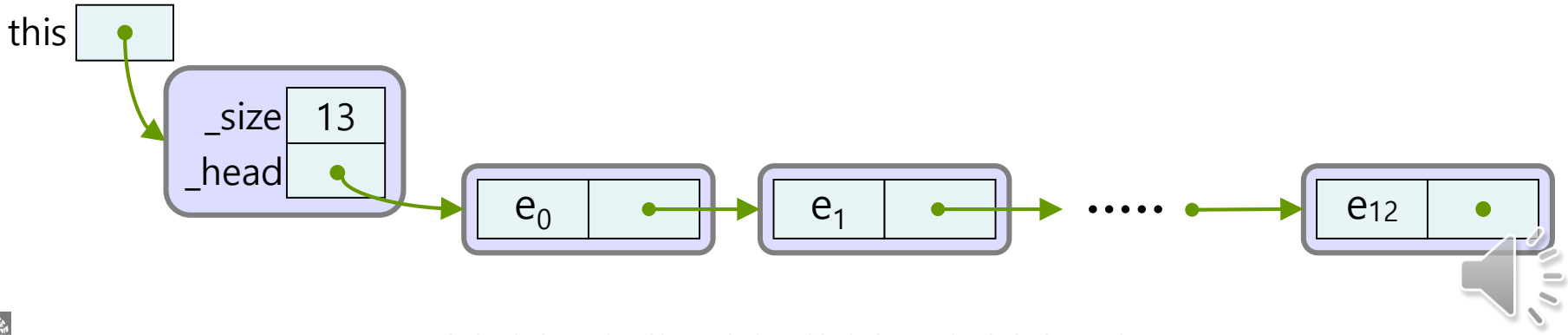
❑ Head Element 가 Last 인 경우

```
public class  LinkedList<E> implements Stack<E>
{
      // 비공개 인스턴스 변수
      private int                 _size ;
      private LinkedNode<E>   _head ;

      // 공개 함수
      public boolean  isEmpty () {...}
      public boolean  isFull () {...}
      public boolean  size () {...}

      @Override
      public E    elementAt (int order) {
            if ( order < 0 || order >= this.size() ) {
                  return  null ;
            }
            else {
                  LinkedNode<E> currentNode = this.head() ;
                  int  count = (this.size()-1) - order ;
                  while ( count > 0 ) {
                        currentNode = currentNode.next() ;
                        count-- ;
                  }
                  return  currentNode.element() ;
            }
      }

      public E    last () {
            return  this.elementAt (this.size()-1) ;
      }

      public boolean  addToLast (E anElement) {
            if ( this.isFull() ) {
                  return  false ;
            }
            else {
                  LinkedNode<E> nodeForAdd =
                        new LinkedNode<E>()  ;
                  nodeForAdd.setElement (anElement) ;
                  nodeForAdd.setNext (this.head()) ;
                  this.setHead (nodeForAdd) ;
                  this.setSize (this.size()+1) ;
            }
      }

      public E   removeLast () {...}
            if ( this.isEmpty() ) {
                  return  null ;
            }
            else {
                  E  lastElement = this.head().element() ;
                  this.setHead (this.head().next()) ;
                  this.setSize (this.size()-1) ;
                  return  lastElement ;
            }
      }

      @Override
      public void  clear() {...}


      // Stack의 구현
      @Override
      public boolean  push (E anElement) {
            return  this.addToLast (anElement) ;
      }
      @Override
      public E  pop () {
            return  this.removeLast() ;
      }
      @Override
      public E  peek () {
            return  this.last() ;
      }

} // End of class "LinkedList"
```

연결 체인을 이용한 스택의 구현에서 좀 더 생각해 볼 점

기존의 연결리스트를 그대로 사용?

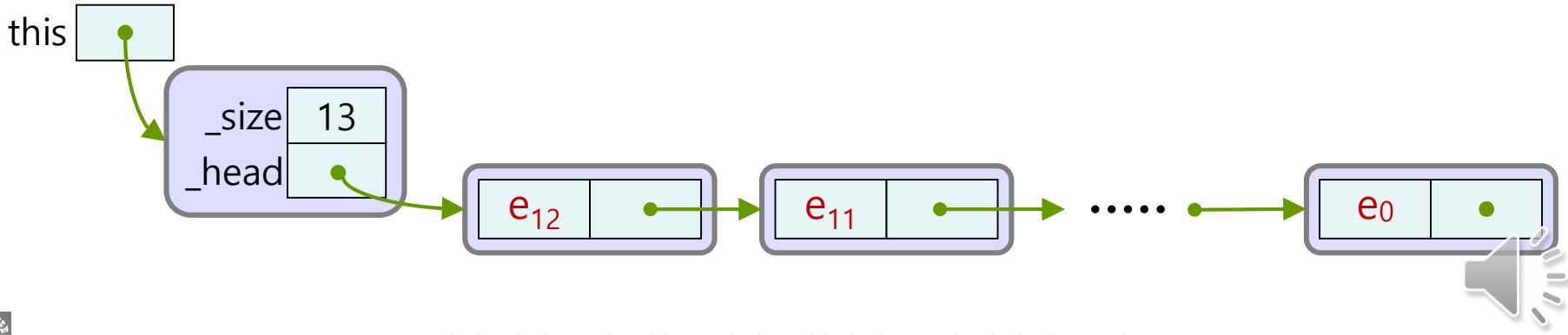
- 관찰:
  - 기존의 class LinkedList<E>는, Head node 가 order 0 의 원소를 저장하도록 구현
  - 그러나 스택으로 사용하려면 연결체인의 마지막 노드를 order 0 의 원소를 저장하는 것이 효율적
  - Class LinkedList<E>를 Stack 으로 사용하면 비효율적으로 작동하므로, 따라서 스택으로 사용하기에는 부적절
  - 기존의 연결 리스트의 구현 역시 필요하다면?



대안 1: 별도의 class 를 정의

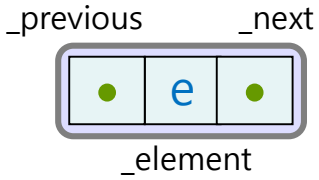
- 기존의 Class LinkedList<E> 는 스택으로 사용하지 않고 그대로 둔다.
  - Order 0 의 원소는 Head Node 에 저장

- 스택은 별도의 class, 이를 테면 class LinkedStack<E>을 따로 정의하고 구현하여 사용
  - Class LinkedStack<E> implements Stack<E> { …… }
  - 앞에서 보았던, head node 가 last order 의 원소를 저장하도록 하는 구현을 그대로 사용하면 된다.
    - Order 0 의 원소는 연결체인의 맨 마지막 노드에 저장

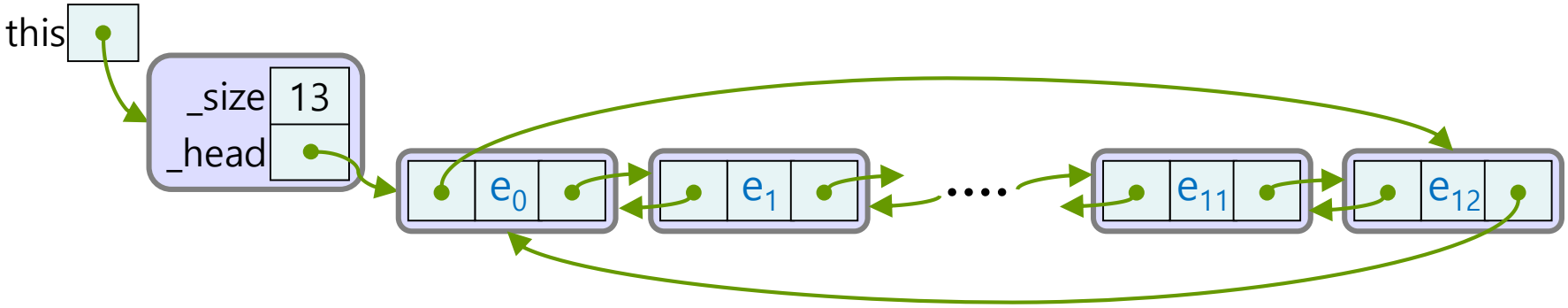


대안2: Doubly Linked List

- 이중 연결체인을 이용한 이중연결 리스트로 구현
  - Class DoublyLinkedNode<E>:



  - Class DoublyLinkedList<E>:

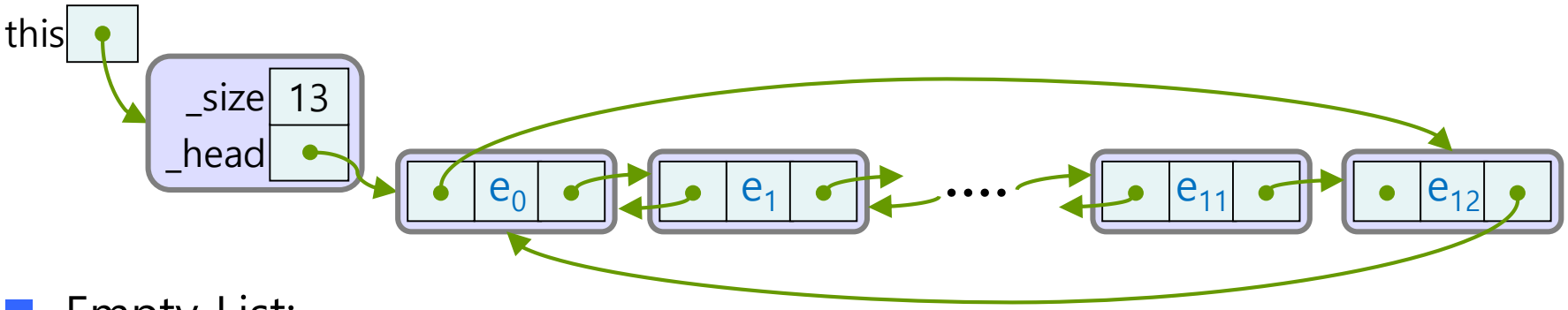


- 특징:
  - 기존의 Class LinkedList 에서의 원소의 order 와 동일한 순서로 저장
  - 환형으로 연결: 마지막 노드도 직접 접근 가능.
    - 마지막 위치로의 삽입, 마지막 원소의 삭제를 효율적으로 실행 가능
  - Linked List 로 사용하면서, 또한 Stack 으로 사용해도 효율적
  - 단점: 메모리를 더 사용

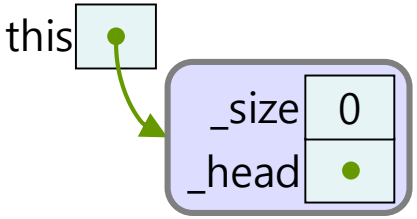
Doubly Linked List 의 구현

Class DoublyLinkedList<E>

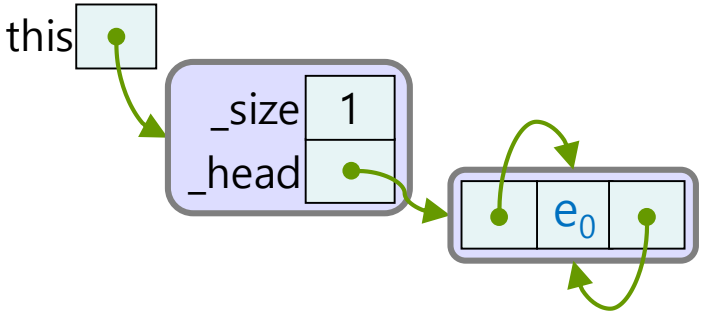
- 2 개 이상의 원소를 저장한 경우



- Empty List:



- 1 개의 원소를 저장한 경우



End of "Stack"