

1. 컬렉션 프레임워크(Collections Framework)

컬렉션 프레임워크이란, '데이터 군(群)을 저장하는 클래스들을 표준화한 설계'를 뜻한다. 컬렉션(Collection)은 다수(多數)의 데이터, 즉 데이터 그룹을, 프레임워크는 표준화된 프로그래밍 방식을 의미한다.

| 참고 | Java API 문서에서는 컬렉션 프레임워크를 '데이터 군(群, group)을 다루고 표현하기 위한 단일화된 구조(architecture)'라고 정의하고 있다.

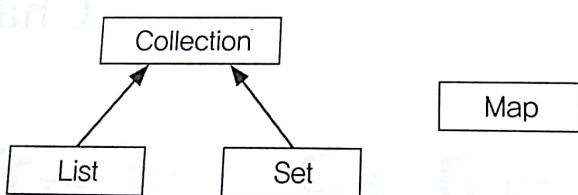
JDK1.2 이전까지는 Vector, Hashtable, Properties와 같은 컬렉션 클래스, 다수의 데이터를 저장할 수 있는 클래스들을 서로 다른 각자의 방식으로 처리해야 했으나 JDK1.2부터 컬렉션 프레임워크가 등장하면서 다양한 종류의 컬렉션 클래스가 추가되고 모든 컬렉션 클래스를 표준화된 방식으로 다룰 수 있도록 체계화되었다.

| 참고 | 앞으로 Vector와 같이 다수의 데이터를 저장할 수 있는 클래스를 '컬렉션 클래스'라고 하겠다.

컬렉션 프레임워크는 컬렉션, 다수의 데이터, 을 다루는 데 필요한 다양하고 풍부한 클래스들을 제공하기 때문에 프로그래머의 짐을 상당히 덜어 주고 있으며, 또한 인터페이스와 형성을 이용한 객체지향적 설계를 통해 표준화되어 있기 때문에 사용법을 익히기에도 편리하고 재사용성이 높은 코드를 작성할 수 있다는 장점이 있다.

1.1 컬렉션 프레임워크의 핵심 인터페이스

컬렉션 프레임워크에서는 컬렉션 데이터 그룹을 크게 3가지 타입이 존재한다고 인식하고 각 컬렉션을 다루는데 필요한 기능을 가진 3개의 인터페이스를 정의하였다. 그리고 인터페이스 List와 Set의 공통된 부분을 다시 뽑아서 새로운 인터페이스인 Collection을 추가로 정의하였다.



▲ 그림 11-1 컬렉션 프레임워크의 핵심 인터페이스간의 상속계층도

인터페이스 List와 Set을 구현한 컬렉션 클래스들은 서로 많은 공통부분이 있어서, 공통된 부분을 다시 뽑아 Collection 인터페이스를 정의할 수 있었지만 Map 인터페이스는 이들과는 전혀 다른 형태로 컬렉션을 다루기 때문에 같은 상속계층도에 포함되지 못했다.

이러한 설계는 객체지향언어의 장점을 극명히 보여주는 것으로 객체지향개념을 학습하는 사람들에게 많은 것을 느끼게 한다. 후에 프로그래밍 실력을 어느 정도 갖추게 되었을 때 컬렉션 프레임워크의 실제 소스를 분석해보면 객체지향적인 설계능력을 향상시키는데 많은 도움이 될 것이다.

| 참고 | JDK1.5부터 Iterable 인터페이스가 추가되고 이를 Collection 인터페이스가 상속받도록 변경되었으나 이것은 단지 인터페이스들의 공통적인 메서드인 iterator()를 뽑아서 중복을 제거하기 위한 것에 불과하므로 상속계층도에서 별 의미가 없다.

인터페이스	특징
List	순서가 있는 데이터의 집합. 데이터의 중복을 허용한다. 예) 대기자 명단 구현클래스 : ArrayList, LinkedList, Stack, Vector 등
Set	순서를 유지하지 않는 데이터의 집합. 데이터의 중복을 허용하지 않는다. 예) 암의 정수집합, 소수의 집합 구현클래스 : HashSet, TreeSet 등
Map	키(key)와 값(value)의 쌍(pair)으로 이루어진 데이터의 집합 순서는 유지되지 않으며, 키는 중복을 허용하지 않고, 값은 중복을 허용한다. 예) 우편번호, 지역번호(전화번호) 구현클래스 : HashMap, TreeMap, Hashtable, Properties 등

▲ 표 11-1 컬렉션 프레임워크의 핵심 인터페이스와 특징

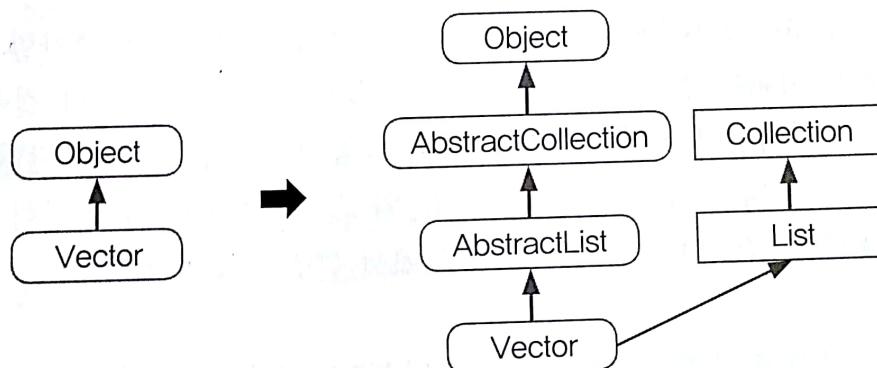
| 참고 | 키(Key)란, 데이터 집합 중에서 어떤 값(value)을 찾는데 열쇠(key)가 된다는 의미에서 붙여진 이름이다. 그래서 키(Key)는 중복을 허용하지 않는다.

실제 개발 시에는 다루고자 하는 컬렉션의 특징을 파악하고 어떤 인터페이스를 구현한 컬렉션 클래스를 사용해야하는지 결정해야하므로 위의 표11-1에 적힌 각 인터페이스의 특징과 차이를 잘 이해하고 있어야 한다.

컬렉션 프레임워크의 모든 컬렉션 클래스들은 List, Set, Map 중의 하나를 구현하고 있으며, 구현한 인터페이스의 이름이 클래스의 이름에 포함되어있어서 이름만으로도 클래스의 특징을 쉽게 알 수 있도록 되어있다.

그러나 Vector, Stack, Hashtable, Properties와 같은 클래스들은 컬렉션 프레임워크 만들어지기 이전부터 존재하던 것이기 때문에 컬렉션 프레임워크의 명명법을 따르지 않는다.

Vector나 Hashtable과 같은 기존의 컬렉션 클래스들은 호환을 위해, 설계를 변경해서 남겨두었지만 가능하면 사용하지 않는 것이 좋다. 그 대신 새로 추가된 ArrayList와 HashMap을 사용하자.



▲ 그림 11-2 Vector 클래스의 상속 계층도 변화 – 왼쪽이 JDK1.2 이전, 오른쪽이 이후

Collection 인터페이스

List와 Set의 조상인 Collection 인터페이스에는 다음과 같은 메서드들이 정의되어 있다.

메서드	설명
boolean add(Object o) boolean addAll(Collection c)	지정된 객체(o) 또는 Collection(c)의 객체들을 Collection에 추가한다.
void clear()	Collection의 모든 객체를 삭제한다.
boolean contains(Object o) boolean containsAll(Collection c)	지정된 객체(o) 또는 Collection의 객체들이 Collection에 포함되어 있는지 확인한다.
boolean equals(Object o)	동일한 Collection인지 비교한다.
int hashCode()	Collection의 hash code를 반환한다.
boolean isEmpty()	Collection이 비어있는지 확인한다.
Iterator iterator()	Collection의 Iterator를 얻어서 반환한다.
boolean remove(Object o)	지정된 객체를 삭제한다.
boolean removeAll(Collection c)	지정된 Collection에 포함된 객체들을 삭제한다.
boolean retainAll(Collection c)	지정된 Collection에 포함된 객체만을 남기고 다른 객체들은 Collection에서 삭제한다. 이 작업으로 인해 Collection에 변화가 있으면 true를 그렇지 않으면 false를 반환한다.
int size()	Collection에 저장된 객체의 개수를 반환한다.
Object[] toArray()	Collection에 저장된 객체를 객체 배열(Object[])로 반환한다.
Object[] toArray(Object[] a)	지정된 배열에 Collection의 객체를 저장해서 반환한다.

▲ 표 11-2 Collection 인터페이스에 정의된 메서드

| 참고 | JDK1.8부터 추가된 parallelStream, removeIf, stream, forEach 등은 14장 람다와 스트림에서 설명한다.

| 참고 | Iterator 인터페이스는 컬렉션에 포함된 객체들에 접근할 수 있는 방법을 제공한다. 곧 배울 것이다.

Collection 인터페이스는 컬렉션 클래스에 저장된 데이터를 읽고, 추가하고 삭제하는 등 컬렉션을 다루는데 가장 기본적인 메서드들을 정의하고 있다.

위의 표 11-2에서 반환 타입이 boolean인 메서드들은 작업을 성공하거나 사실이면 true를, 그렇지 않으면 false를 반환한다.

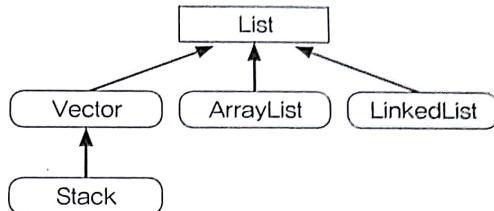
예를 들어 'boolean add(Object o)'를 사용해서 객체를 컬렉션에 추가할 때, 성공하면 true를, 실패하면 false를 반환한다. 'boolean isEmpty()'를 사용해서 컬렉션에 포함된 객체가 없으면, 즉 컬렉션이 비어있으면 true를, 그렇지 않으면 false를 반환한다.

이 외에도 JDK1.8부터 추가된 '람다(Lambda)와 스트림(Stream)'에 관련된 메서드들이 더 있는데, 이 메서드들은 '14장 람다와 스트림'에서 설명할 것이다.

그리고 Java API 문서를 보면, 표 11-2에 사용된 'Object'가 아닌 'E'로 표기되어 있는데, E는 특정 타입을 의미하는 것으로 JDK1.5부터 추가된 지네릭스(Generics)에 의한 표기이다. 아직 지네릭스를 배우지 않았기 때문에 이해를 돋기 위한 의도로 'E' 대신 'Object'로 표기했다. 'E' 외에도 'T'나 'K', 'V'를 사용하는 경우도 있는데 모두 Object 타입이라고 이해하자. 지네릭스는 12장에서 배울 것이다.

List인터페이스

List인터페이스는 중복을 허용하면서 저장순서가 유지되는 컬렉션을 구현하는데 사용된다.



▲ 그림 11-3 List의 상속계층도

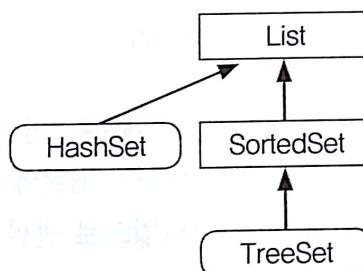
List인터페이스에 정의된 메서드는 다음과 같다. Collection인터페이스로부터 상속받은 것들은 제외하였다.

메서드	설명
void add(int index, Object element) boolean addAll(int index, Collection c)	지정된 위치(index)에 객체(element) 또는 컬렉션에 포함된 객체들을 추가한다.
Object get(int index)	지정된 위치(index)에 있는 객체를 반환한다.
int indexOf(Object o)	지정된 객체의 위치(index)를 반환한다. (List의 첫 번째 요소부터 순방향으로 찾는다.)
int lastIndexOf(Object o)	지정된 객체의 위치(index)를 반환한다. (List의 마지막 요소부터 역방향으로 찾는다.)
ListIterator listIterator() ListIterator listIterator(int index)	List의 객체에 접근할 수 있는 ListIterator를 반환한다.
Object remove(int index)	지정된 위치(index)에 있는 객체를 삭제하고 삭제된 객체를 반환한다.
Object set(int index, Object element)	지정된 위치(index)에 객체(element)를 저장한다
void sort(Comparator c)	지정된 비교자(comparator)로 List를 정렬한다.
List subList(int fromIndex, int toIndex)	지정된 범위(fromIndex부터 toIndex)에 있는 객체를 반환한다.

▲ 표 11-3 List인터페이스의 메서드

Set인터페이스

Set인터페이스는 중복을 허용하지 않고 저장순서가 유지되지 않는 컬렉션 클래스를 구현하는데 사용된다. Set인터페이스를 구현한 클래스로는 HashSet, TreeSet 등이 있다.

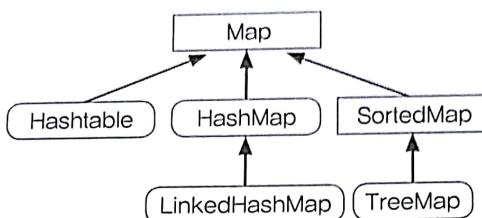


▲ 그림 11-4 Set의 상속계층도

Map인터페이스

Map인터페이스는 키(key)와 값(value)을 하나의 쌍으로 묶어서 저장하는 컬렉션 클래스를 구현하는 데 사용된다. 키는 중복될 수 있지만 값은 중복을 허용한다. 기존에 저장된 데이터와 중복된 키와 값을 저장하면 기존의 값은 없어지고 마지막에 저장된 값이 남게 된다. Map인터페이스를 구현한 클래스로는 Hashtable, HashMap, LinkedHashMap, SortedMap, TreeMap 등이 있다.

| 참고 | Map이란 개념은 어떤 두 값을 연결한다는 의미에서 붙어진 이름이다.



▲ 그림 11-5 Map의 상속계층도

메서드	설명
void clear()	Map의 모든 객체를 삭제한다.
boolean containsKey(Object key)	지정된 key객체와 일치하는 Map의 key객체가 있는지 확인한다.
boolean containsValue(Object value)	지정된 value객체와 일치하는 Map의 value객체가 있는지 확인한다.
Set entrySet()	Map에 저장되어 있는 key-value쌍을 Map.Entry타입의 객체로 저장한 Set으로 반환한다.
boolean equals(Object o)	동일한 Map인지 비교한다.
Object get(Object key)	지정한 key객체에 대응하는 value객체를 찾아서 반환한다.
int hashCode()	해시코드를 반환한다.
boolean isEmpty()	Map이 비어있는지 확인한다.
Set keySet()	Map에 저장된 모든 key객체를 반환한다.
Object put(Object key, Object value)	Map에 value객체를 key객체에 연결(mapping)하여 저장한다.
void putAll(Map t)	지정된 Map의 모든 key-value쌍을 추가한다.
Object remove(Object key)	지정한 key객체와 일치하는 key-value객체를 삭제한다.
int size()	Map에 저장된 key-value쌍의 개수를 반환한다.
Collection values()	Map에 저장된 모든 value객체를 반환한다.

▲ 표 11-4 Map인터페이스의 메서드

values()에서는 반환타입이 Collection이고, keySet()에서는 반환타입이 Set인 것에 주목하자. Map인터페이스에서 값(value)은 중복을 허용하기 때문에 Collection타입으로 반환하고, 키(key)는 중복을 허용하지 않기 때문에 Set타입으로 반환한다.

Map.Entry인터페이스

Map.Entry인터페이스는 Map인터페이스의 내부 인터페이스이다. 내부 클래스와 같이 인터페이스도 인터페이스 안에 인터페이스를 정의하는 내부 인터페이스(inner interface)를 정의하는 것이 가능하다.

Map에 저장되는 key-value쌍을 다루기 위해 내부적으로 Entry인터페이스를 정의해 놓았다. 이것은 보다 객체지향적으로 설계하도록 유도하기 위한 것으로 Map인터페이스를 구현하는 클래스에서는 Map.Entry인터페이스도 함께 구현해야한다.

다음은 Map인터페이스의 소스코드의 일부이다.

```
public interface Map {
    ...
    interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
        boolean equals(Object o);
        int hashCode();
        ...
    }
}
```

메서드	설명
boolean equals(Object o)	동일한 Entry인지 비교한다.
Object getKey()	Entry의 key객체를 반환한다.
Object getValue()	Entry의 value객체를 반환한다.
int hashCode()	Entry의 해시코드를 반환한다.
Object setValue(Object value)	Entry의 value객체를 지정된 객체로 바꾼다.

▲ 표11-5 Map.Entry인터페이스의 메서드

이제 본론으로 들어가 컬렉션 프레임워크의 구성요소들을 하나씩 살펴보도록 하자.

1.2 ArrayList

ArrayList는 컬렉션 프레임워크에서 가장 많이 사용되는 컬렉션 클래스일 것이다. 이 ArrayList는 List 인터페이스를 구현하기 때문에 데이터의 저장순서가 유지되고 중복을 허용한다는 특징을 갖는다.

ArrayList는 기존의 Vector를 개선한 것으로 Vector와 구현원리가 가능한적인 측면에서 동일하다고 할 수 있다. 앞에서 얘기했던 것과 같이 Vector는 기존에 작성된 소스와의 호환성을 위해서 계속 남겨 두고 있을 뿐이기 때문에 가능하면 Vector보다는 ArrayList를 사용하자.

ArrayList는 Object 배열을 이용해서 데이터를 순차적으로 저장한다. 예를 들면, 첫 번째로 저장한 객체는 Object 배열의 0번째 위치에 저장되고 그 다음에 저장하는 객체는 1번 째 위치에 저장된다. 이런 식으로 계속 배열에 순서대로 저장되며, 배열에 더 이상 저장할 공간이 없으면 보다 큰 새로운 배열을 생성해서 기존의 배열에 저장된 내용을 새로운 배열로 복사한 다음에 저장된다.

```
public class ArrayList extends AbstractList
    implements List, RandomAccess, Cloneable, java.io.Serializable {
    ...
    transient Object[] elementData; // Object 배열
    ...
}
```

| 참고 | transient는 직렬화(serialization)와 관련된 제어자이다. 직렬화에 대해서는 15장에서 다룬다.

위의 코드는 ArrayList의 소스코드 일부인데 ArrayList는 elementData라는 이름의 Object 배열을 멤버 변수로 선언하고 있다는 것을 알 수 있다. 선언된 배열의 타입이 모든 객체의 최고조상인 Object이기 때문에 모든 종류의 객체를 담을 수 있다.

메서드	설명
ArrayList()	크기가 10인 ArrayList를 생성
ArrayList(Collection c)	주어진 컬렉션이 저장된 ArrayList를 생성
ArrayList(int initialCapacity)	지정된 초기용량을 갖는 ArrayList를 생성
boolean add(Object o)	ArrayList의 마지막에 객체를 추가. 성공하면 true
void add(int index, Object element)	지정된 위치(index)에 객체를 저장
boolean addAll(Collection c)	주어진 컬렉션의 모든 객체를 저장한다.
boolean addAll(int index, Collection c)	지정된 위치부터 주어진 컬렉션의 모든 객체를 저장한다.
void clear()	ArrayList를 완전히 비운다.
Object clone()	ArrayList를 복제한다.
boolean contains(Object o)	지정된 객체(o)가 ArrayList에 포함되어 있는지 확인
void ensureCapacity(int minCapacity)	ArrayList의 용량이 최소한 minCapacity가 되도록 한다.
Object get(int index)	지정된 위치(index)에 저장된 객체를 반환한다.

```
int indexOf()
boolean isEmpty()
Iterator iterator()
int lastIndexOf()
ListIterator listIterator()
Object remove()
boolean remove()
boolean remove()
Object set(int index, Object element)
int size()
void sort(Comparator<T> comparator)
List subList(int fromIndex, int toIndex)
Object[] toArray()
Object[] toArray()
void trimToSize()
```

▼ 예제 11-1/

```
import java.util.ArrayList;
class ArrayList {
    public ArrayList() {
        Arr
        lis
        lis
        lis
        lis
        lis
        lis
        lis
        lis
        lis
        Arr
        pri
        Co
        Co
        pri
        Sys
        lis
        lis
```

int indexOf(Object o)	지정된 객체가 저장된 위치를 찾아 반환한다.
boolean isEmpty()	ArrayList가 비어있는지 확인한다.
Iterator iterator()	ArrayList의 Iterator 객체를 반환
int lastIndexOf(Object o)	객체(o)가 저장된 위치를 끝부터 역방향으로 검색해서 반환
ListIterator listIterator()	ArrayList의 ListIterator를 반환
ListIterator listIterator(int index)	ArrayList의 지정된 위치부터 시작하는 ListIterator를 반환
Object remove(int index)	지정된 위치(index)에 있는 객체를 제거한다.
boolean remove(Object o)	지정한 객체를 제거한다.(성공하면 true, 실패하면 false)
boolean removeAll(Collection c)	지정한 컬렉션에 저장된 것과 동일한 객체들을 ArrayList에서 제거한다.
boolean retainAll(Collection c)	ArrayList에 저장된 객체 중에서 주어진 컬렉션과 공통된 것들만을 남기고 나머지는 삭제한다.
Object set(int index, Object element)	주어진 객체(element)를 지정된 위치(index)에 저장한다.
int size()	ArrayList에 저장된 객체의 개수를 반환한다.
void sort(Comparator c)	지정된 정렬기준(c)으로 ArrayList를 정렬
List subList(int fromIndex, int toIndex)	fromIndex부터 toIndex사이에 저장된 객체를 반환한다.
Object[] toArray()	ArrayList에 저장된 모든 객체들을 객체배열로 반환한다.
Object[] toArray(Object[] a)	ArrayList에 저장된 모든 객체들을 객체배열 a에 담아 반환한다.
void trimToSize()	용량을 크기에 맞게 줄인다.(빈 공간을 없앤다.)

▲ 표11-6 ArrayList의 생성자와 메서드

▼ 예제 11-1/ch11/ArrayListEx1.java

```

import java.util.*;

class ArrayListEx1{
    public static void main(String[] args) {
        ArrayList list1 = new ArrayList(10);
        list1.add(new Integer(5));
        list1.add(new Integer(4));
        list1.add(new Integer(2));
        list1.add(new Integer(0));
        list1.add(new Integer(1));
        list1.add(new Integer(3));
        ArrayList list2 = new ArrayList(list1.subList(1,4));
        print(list1, list2);
        Collections.sort(list1);      // list1과 list2를 정렬한다.
        Collections.sort(list2);      // Collections.sort(List l)
        print(list1, list2);
        System.out.println("list1.containsAll(list2):"
                           + list1.containsAll(list2));
        list2.add("B");
        list2.add("C");
    }
}

```

```

list2.add(3, "A");
print(list1, list2);

list2.set(3, "AA");
print(list1, list2);

// list1에서 list2와 겹치는 부분만 남기고 나머지는 삭제한다.
// list1.retainAll(list2);
System.out.println("list1.retainAll(list2):" + list1.retainAll(list2));

print(list1, list2);

// list2에서 list1에 포함된 객체들을 삭제한다.
for(int i= list2.size()-1; i >= 0; i--) {
    if(list1.contains(list2.get(i)))
        list2.remove(i);
}

print(list1, list2);
} // main의 끝

static void print(ArrayList list1, ArrayList list2) {
    System.out.println("list1:" + list1);
    System.out.println("list2:" + list2);
    System.out.println();
}
} // class

```

▼ 실행결과

list1:[5, 4, 2, 0, 1, 3]
list2:[4, 2, 0]

list1:[0, 1, 2, 3, 4, 5] ← Collections.sort(List 1)를 이용해서 정렬하였다.
list2:[0, 2, 4]

list1.containsAll(list2) :true ← list1이 list2의 모든 요소를 포함하고 있을 때만 true
list1:[0, 1, 2, 3, 4, 5]
list2:[0, 2, 4, A, B, C] ← add(Object obj)를 이용해서 새로운 객체를 저장하였다.

list1:[0, 1, 2, 3, 4, 5]
list2:[0, 2, 4, AA, B, C] ← set(int index, Object obj)를 이용해서 다른 객체로 변경
list1.retainAll(list2) :true ← retainAll에 의해 list1에 변화가 있었으므로 true를 반환
list1:[0, 2, 4] ← list2와의 공통요소 이외에는 모두 삭제되었다(변화가 있었다).
list2:[0, 2, 4, AA, B, C]

list1:[0, 2, 4]
list2:[AA, B, C]

위의 예제는 ArrayList의 기본적인 메서드를 이용해서 객체를 다루는 방법을 보여 준다. ArrayList는 List인터페이스를 구현했기 때문에 저장된 순서를 유지한다는 것을 알 수 있다. 그리고 Collections클래스의 sort메서드를 이용해서 ArrayList에 저장된 객체들을 정렬하였는데 Collections클래스에 대한 내용과 정렬(sort)하는 방법에 대해서는 후에 자세히 다룰 것이므로 간단한 정렬방법이 있다는 정도만 이해하고 넘어가자.
주의! Collection은 인터페이스이고, Collections는 클래스임에 주의!

```

for(int i = list2.size()-1; i >= 0; i--) {
    if(list1.contains(list2.get(i)))
        list2.remove(i);
}

```

이 예제에서 한 가지 더 설명할 것은 위의 코드인데, list2에서 list1과 공통되는 요소들을 찾아서 삭제하는 부분이다.

여기서는 list2의 각 요소를 접근하기 위해 get(int index)메서드와 for문을 사용하였는데, for문의 변수 i를 0부터 증가시킨 것이 아니라, list2.size()-1부터 감소시켜면서 거꾸로 반복시켰다.

만일 변수 i를 증가시켜가면서 삭제하면, 한 요소가 삭제될 때마다 빈 공간을 채우기 위해 나머지 요소들이 자리이동을 하기 때문에 올바른 결과를 얻을 수 없다. 그래서 제어변수를 감소시켜가면서 삭제를 해야 자리이동이 발생해도 영향을 받지 않고 작업이 가능하다.

여기에 대해서는 나중에 다시 자세히 설명할 것이지만 잠시 고민해보고 넘어가면 좋을 것이다.

| 참고 | 예제11-1의 for문을 변경해서 i의 값을 증가시켜가면서 삭제해보면 쉽게 이해할 수 있을 것이다.

▼ 예제 11-2/ch11/ArrayListEx2.java

```

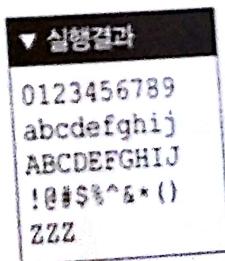
import java.util.*;

class ArrayListEx2 {
    public static void main(String[] args) {
        final int LIMIT = 10;           // 자르고자 하는 글자의 개수를 지정한다.
        String source = "0123456789abcdefgijABCDEFHIJ!@#$%^&*()ZZZ";
        int length = source.length();
        List list = new ArrayList(length/LIMIT + 10); // 크기를 약간 여유 있게 잡는다.

        for(int i=0; i < length; i+=LIMIT) {
            if(i+LIMIT < length )
                list.add(source.substring(i, i+LIMIT));
            else
                list.add(source.substring(i));
        }

        for(int i=0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    } // main()
}

```



긴 문자열 데이터를 원하는 길이로 잘라서 ArrayList에 담은 다음 출력하는 예제이다. 단순히 문자열을 특정크기로 잘라서 출력할 것이라면, charAt(int i)와 for문을 이용하면 되겠지만 ArrayList에 잘라서 담아놓음으로써 ArrayList의 기능을 이용해서 다양한 작업을 간단하게 처리할 수 있다.

List list = new ArrayList(length/LIMIT + 10); // 크기를 약간 여유 있게 잡는다.

ArrayList를 생성할 때, 저장할 요소의 개수를 고려해서 실제 저장할 개수보다 약간 여유 있는 크기로 하는 것이 좋다. 생성할 때 지정한 크기보다 더 많은 객체를 저장하면 자동적으로 크기가 늘어나기는 하지만 이 과정에서 처리시간이 많이 소요되기 때문이다.

▼ 예제 11-3/ch11/VectorEx1.java

```
import java.util.*;  
  
class VectorEx1 {  
    public static void main(String[] args) {  
        Vector v = new Vector(5);      // 용량(capacity)이 5인 Vector를 생성한다.  
        v.add("1");  
        v.add("2");  
        v.add("3");  
        print(v);  
  
        v.trimToSize(); // 빈 공간을 없앤다. (용량과 크기가 같아진다.)  
        System.out.println("== After trimToSize() ==");  
        print(v);  
  
        v.ensureCapacity(6);  
        System.out.println("== After ensureCapacity(6) ==");  
        print(v);  
  
        v.setSize(7);  
        System.out.println("== After setSize(7) ==");  
        print(v);  
  
        v.clear();  
        System.out.println("== After clear() ==");  
        print(v);  
    }  
  
    public static void print(Vector v) {  
        System.out.println(v);  
        System.out.println("size :" + v.size());  
        System.out.println("capacity :" + v.capacity());  
    }  
}
```

▼ 실행결과

```
[1, 2, 3]  
size :3  
capacity :5  
== After trimToSize() ==  
[1, 2, 3]  
size :3  
capacity :3  
== After ensureCapacity(6) ==  
[1, 2, 3]  
size :3
```

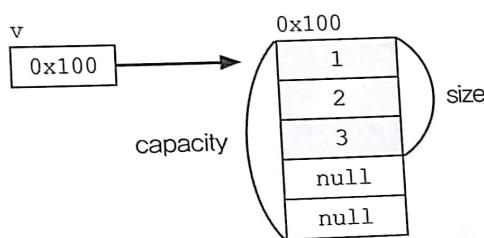
```

capacity :6
==== After setSize(7) ====
[1, 2, 3, null, null, null, null]
size :7
capacity :12
==== After clear() ====
[]
size :0
capacity :12

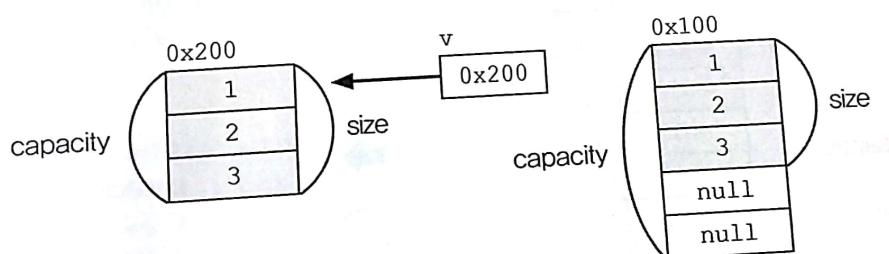
```

이 예제는 Vector의 용량(capacity)과 크기(size)에 관한 것인데, 각 실행과정을 그림과 함께 단계별로 살펴보자.

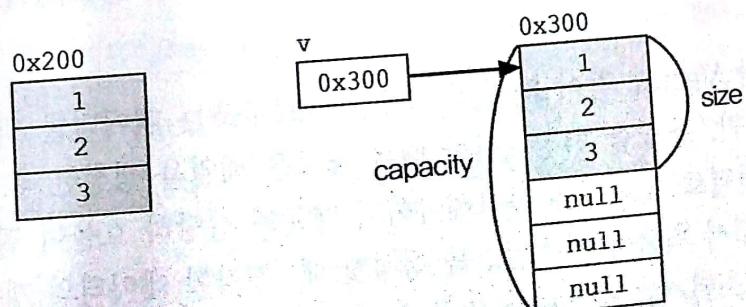
1. 다음 capacity가 5인 Vector인스턴스 v를 생성하고, 3개의 객체를 저장한 후의 상태를 그림으로 나타낸 것이다.



2. *v.trimToSize()*를 호출하면 *v*의 빈 공간을 없애서 size와 capacity를 같게 한다. 배열은 크기를 변경할 수 없기 때문에 새로운 배열을 생성해서 그 주소값을 변수 *v*에 할당한다. 기존의 Vector인스턴스는 더 이상 사용할 수 없으며, 후에 가비지컬렉터(garbage collector)에 의해서 메모리에서 제거된다.

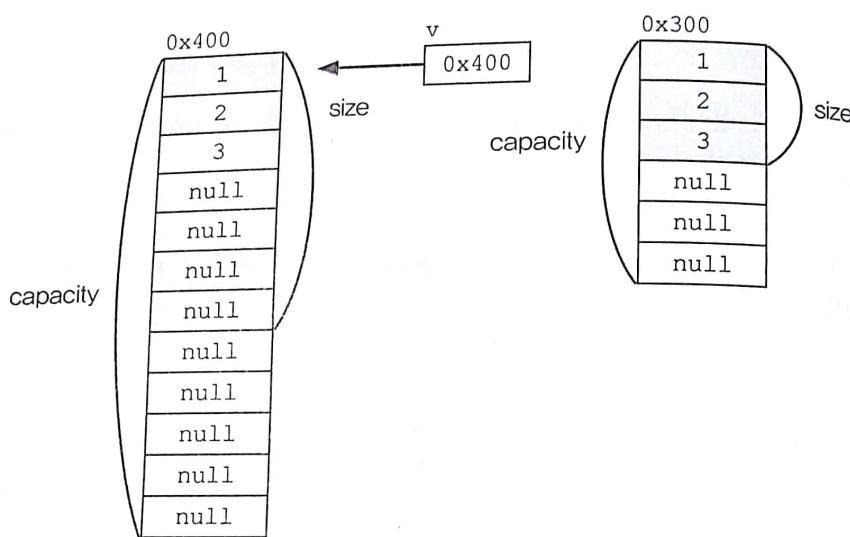


3. *v.ensureCapacity(6)*는 *v*의 capacity가 최소한 6이 되도록 한다. 만일 *v*의 capacity가 6이상이라면 아무 일도 일어나지 않는다. 현재는 *v*의 capacity가 3이므로 크기가 6인 배열을 생성해서 *v*의 내용을 복사했다. 기존의 인스턴스를 다시 사용하는 것이 아니라 새로운 인스턴스를 생성하였음에 주의하자.

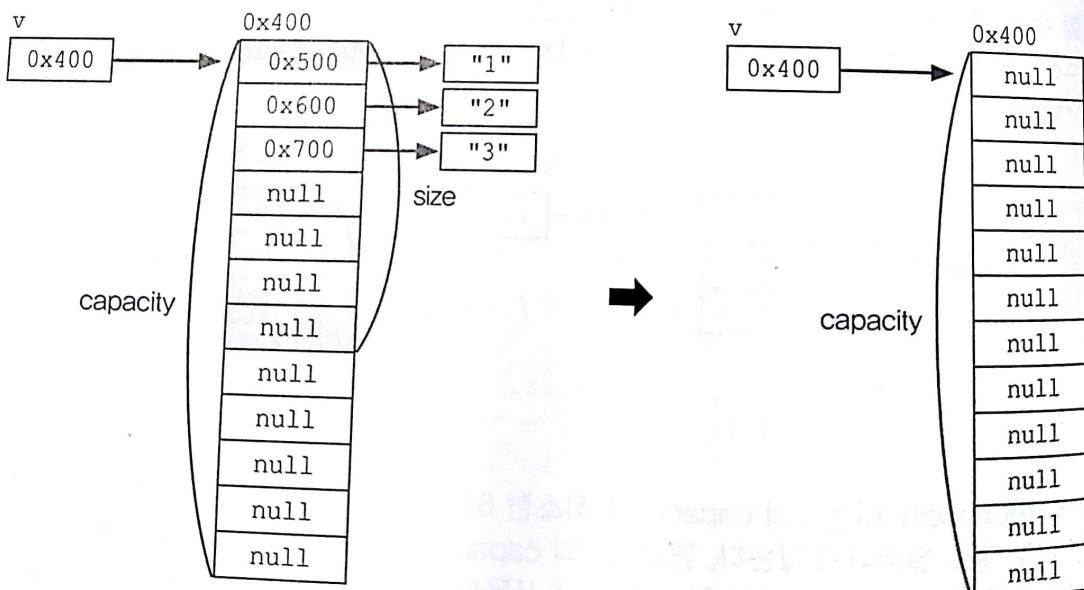


4. `v.setSize(7)`는 `v`의 `size`가 7이 되도록 한다. 만일 `v`의 `capacity`가 충분하면 새로 인스턴스를 생성하지 않아도 되지만 지금은 `capacity`가 6이므로 새로운 인스턴스를 생성해야 한다. `Vector`는 `capacity`가 부족할 경우 자동적으로 기존의 크기보다 2배의 크기로 증가된다. 그래서 `v`의 `capacity`는 12가 된다.

| 참고 | 생성자 `Vector(int initialCapacity, int capacityIncrement)`를 사용해서 인스턴스를 생성한 경우에는 `capacityIncrement`만큼 증가하게 된다.



5. `v.clear()`는 `v`의 모든 요소를 삭제한다. 아래의 왼쪽그림의 상태에서 오른쪽 그림과 같은 상태가 되는 것이다.



| 참고 | `Vector`는 `Object` 배열이기 때문에 실제로는 마지막 그림처럼 주소가 저장되어야 더 정확한 것이지만, 편의상 이전의 그림들은 간략하게 표현했다.

`ArrayList`나 `Vector` 같이 배열을 이용한 자료구조는 데이터를 읽어오고 저장하는 데는 효율이 좋지만, 용량을 변경해야 할 때는 새로운 배열을 생성한 후 기존의 배열로부터 새로 생성된 배열로 데이터를 복사해야 하기 때문에 상당히 효율이 떨어진다는 단점을 가지고 있다. 그래서 처음에 인스턴스를 생성할 때, 저장할 데이터의 개수를 잘 고려하여 충분한 용량의 인스턴스를 생성하는 것이 좋다.

지금까지 ArrayList에 대한 기본 내용들은 모두 살펴보았다. 이제는 조금 더 깊숙이 들어가 Array클래스의 내부를 분석해보자.

다음의 예제는 Vector클래스의 실제코드를 바탕으로 이해하기 쉽게 재구성한 것이다.

▼ 예제 11-4/ch11/MyVector.java

```

import java.util.*;

public class MyVector implements List {
    Object[] data = null; // 객체를 담기 위한 객체배열을 선언한다.
    int capacity = 0; // 용량
    int size = 0; // 크기

    public MyVector(int capacity) {
        if (capacity < 0)
            throw new IllegalArgumentException("유효하지 않은 값입니다. :" + capacity);

        this.capacity = capacity;
        data = new Object[capacity];
    }

    public MyVector() {
        this(10); // 크기를 지정하지 않으면 크기를 10으로 한다.
    }

    // 최소한의 저장공간(capacity)를 확보하는 메서드
    public void ensureCapacity(int minCapacity) {
        if (minCapacity - data.length > 0)
            setCapacity(minCapacity);
    }

    public boolean add(Object obj) {
        // 새로운 객체를 저장하기 전에 저장할 공간을 확보한다.
        ensureCapacity(size+1);
        data[size++] = obj;
        return true;
    }

    public Object get(int index) {
        if(index < 0 || index >= size)
            throw new IndexOutOfBoundsException("범위를 벗어났습니다.");
        return data[index];
    }

    public Object remove(int index) {
        Object oldObj = null;

        if(index < 0 || index >= size)
            throw new IndexOutOfBoundsException("범위를 벗어났습니다.");
        oldObj = data[index];

        // 삭제하고자 하는 객체가 마지막 객체가 아니라면, 배열복사를 통해 빈자리를 채워줘야 한다.
        if(index != size-1) {
            System.arraycopy(data, index+1, data, index, size-index-1);
        }
        // 마지막 데이터를 null로 한다. 배열은 0 부터 시작하므로 마지막 요소는 index가 size-1이다.
        data[size-1] = null;
        size--;
        return oldObj;
    }
}

```

```
public boolean remove(Object obj) {
    for(int i=0; i< size; i++) {
        if(obj.equals(data[i])) {
            remove(i);
            return true;
        }
    }
    return false;
}

public void trimToSize() {
    setCapacity(size);
}

private void setCapacity(int capacity) {
    if(this.capacity==capacity) return; // 크기가 같으면 변경하지 않는다.
    Object[] tmp = new Object[capacity];
    System.arraycopy(data, 0, tmp, 0, size);
    data = tmp;
    this.capacity = capacity;
}

public void clear(){
    for (int i = 0; i < size; i++)
        data[i] = null;
    size = 0;
}

public Object[] toArray(){
    Object[] result = new Object[size];
    System.arraycopy(data, 0, result, 0, size);

    return result;
}

public boolean isEmpty() { return size==0; }
public int capacity() { return capacity; }
public int size() { return size; }
/*****************/
/*      List인터페이스로부터 상속받은 메서드들      */
/*****************/
// public int size();
// public boolean isEmpty();
public boolean contains(Object o){ return false; }
// public Iterator iterator(){ return null; }
public Object[] toArray();
// public Object[] toArray(Object a[]){ return null; }
// public boolean add(Object o);
public boolean remove(Object o);
public boolean containsAll(Collection c){ return false; }
public boolean addAll(Collection c){ return false; }
public boolean addAll(int index, Collection c){ return false; }
public boolean removeAll(Collection c){ return false; }
// public boolean retainAll(Collection c){ return false; }
public void clear();
// public boolean equals(Object o){ return false; }
// public int hashCode();
// public Object get(int index);
```

```

    public Object set(int index, Object element){ return null; }
    public void add(int index, Object element){}
//  public Object remove(int index);
    public int indexOf(Object o){ return -1; }
    public int lastIndexOf(Object o){ return -1; }
    public ListIterator listIterator(){ return null; }
    public ListIterator listIterator(int index){ return null; }
    public List subList(int fromIndex, int toIndex){ return null; }
    default void sort(Comparator c) { /* 내용생략 */ } // JDK1.8부터
    default Spliterator spliterator() { /* 내용생략 */ } // JDK1.8부터
    default void replaceAll(UnaryOperator operator) /* 내용생략 */ // JDK1.8부터
}

```

List인터페이스의 메서드 중 주석처리한 것은 코드를, 정상적으로 동작하도록 구현한 것이고, 주석처리하지 않은 것은 컴파일만 가능하도록 최소한으로 구현한 것이다.

| 참고 | 인터페이스를 구현할 때 인터페이스에 정의된 모든 메서드를 구현해야 한다. 일부 메서드를 구현했다면 후속클래스로 선언해야한다. 그러나 JDK1.8부터 List인터페이스에 3개의 디폴트 메서드가 추가되었으며, 이들은 구현하지 않아도 된다.

```

    public boolean contains(Object o){ return false; }
    public boolean equals(Object o){ return false; }
    public Object set(int index, Object element){ return null; }
    public void add(int index, Object element){}
    public int indexOf(Object o){ return -1; }
    public int lastIndexOf(Object o){ return -1; }
    public String toString() { return ""; }

```

이 중에서 위의 메서드들은 여러분 스스로 구현할 수 있을 정도의 수준이므로 올바르게 동작하도록 코드를 직접 작성하고 테스트 해보자. 또는 메서드의 실제 구현내용을 모두 삭제하고 본인이 직접 다시 코드를 작성해 보는 것도 좋은 공부가 될 것이다.

지금까지 학습해온 내용으로 충분히 이해할 수 있는 수준이라, 굳이 별도의 설명이 필요 없으리라 생각한다. remove메서드만큼은 이해하기 어려울 수도 있기 때문에 단계별로 자세히 설명하겠다.

```

public Object remove(int index) {
    Object oldObj = null;

    if(index < 0 || index >= size)
        throw new IndexOutOfBoundsException("범위를 벗어났습니다.");
    oldObj = data[index];

    if(index != size-1) {
        System.arraycopy(data, index+1, data, index, size-index-1);

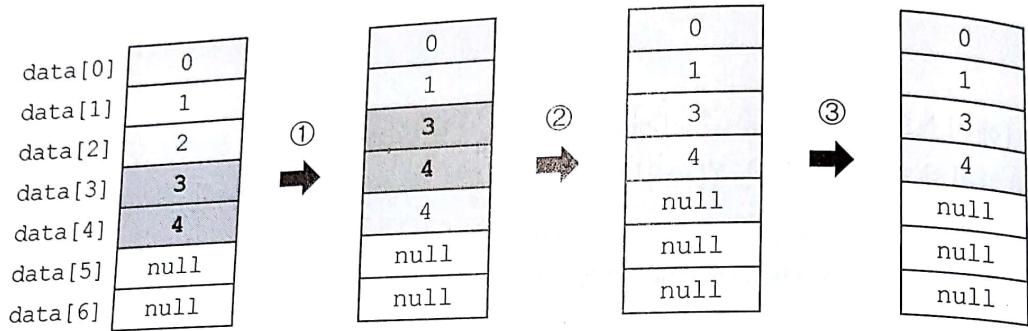
        data[size-1] = null;
        size--;
    }

    return oldObj;
}

```

Object remove(int index) 메서드는 지정된 위치(index)에 있는 객체를 삭제하고 삭제한
객체를 반환하도록 작성하였다. 삭제할 객체의 바로 아래에 있는 데이터를 한 칸씩 위로
복사해서 삭제할 객체를 덮어쓰는 방식으로 처리한다. 만일 삭제할 객체가 마지막 데이터
라면, 복사할 필요없이 단순히 null로 변경해주기만 하면 된다.

아래의 그림은 MyVector 클래스에 0~4의 값이 저장되어 있는 상태에서 세 번째 데이터
를 삭제하기 위해 remove(2)를 호출했다고 가정하고, 그 수행과정의 일부를 단계별로 나
타낸 것이다.



1. 삭제할 데이터의 아래에 있는 데이터를 한 칸씩 위로 복사해서 삭제할 데이터를 덮어쓴다.

현재 5개의 데이터가 저장되어 있으므로 size는 5이고 삭제할 객체의 index는 2이므로 System.arraycopy(data, index+1, data, index, size-index-1)은 System.arraycopy(data, 3, data, 2, 2)가 된다.

System.arraycopy(data, 3, data, 2, 2)

data[3]에서 data[2]로 2개의 데이터를 복사하라는 의미이다.

2. 데이터가 모두 한 칸씩 위로 이동하였으므로 마지막 데이터는 null로 변경해야 한다.

data[size-1] = null;

3. 데이터가 삭제되어 데이터의 개수가 줄었으므로 size의 값을 1 감소시킨다.

size--;

소스코드가 잘 이해되지 않을 때는 이와 같이 상황을 설정해서, 단계별 변화를 그림으로
그려보면 많은 도움이 된다. 머릿속으로만 생각하는 것은 한계가 있으니 꼭 그림으로 그
리는 습관을 들이자.

위 과정을 통해 배워야 할 것은 배열에 객체를 순차적으로 저장할 때와 객체를 마지막에
저장된 것부터 삭제하면 System.arraycopy()를 호출하지 않기 때문에 작업시간이 짧지
만, 배열의 중간에 위치한 객체를 추가하거나 삭제하는 경우 System.arraycopy()를 호
출해서 다른 데이터의 위치를 이동시켜 줘야 하기 때문에 다루는 데이터의 개수가 많을수
록 작업시간이 오래 걸린다는 것이다.

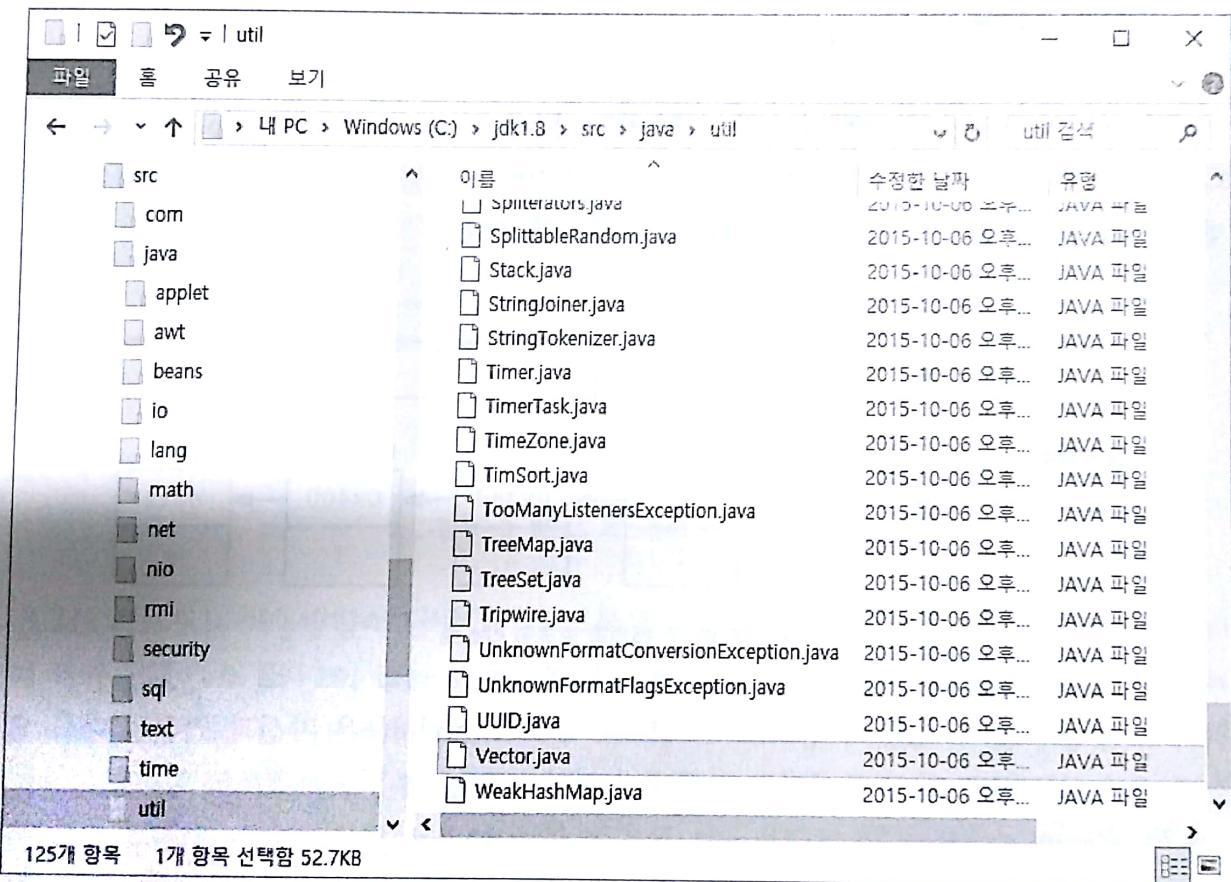
| 참고 | 여기서도 예제11-3에서처럼 그림을 단순화하기 위해서 객체의 주소 대신 값이 저장되어 있는 것처럼 표현했다.

■ 알아두면 좋아요! – Java API소스보기

Vector클래스와 같이 Java API에서 제공하는 기본 클래스의 실제 소스를 보고 싶다면, JDK를 설치한 디렉토리의 src.zip파일을 찾을 수 있다.

src.zip이라는 파일의 압축을 푼 다음, 패키지별로 찾아 들어가면 원하는 클래스의 실제 소스를 볼 수 있다.

예를 들어 Vector클래스는 java.util패키지에 있으므로 'src\java\util\Vector.java'가 소스파일이다. Java API의 소스는 오랜 경력의 전문프로그래머들에 의해서 작성된 것이기 때문에 어떻게 작성하였는지 보고 따라하는 것은 프로그래밍실력을 향상시키는데 많은 도움이 될 것이다.



▲ 그림11-6 Java API의 소스보기

1.3 LinkedList

배열은 가장 기본적인 형태의 자료구조로 구조가 간단하며 사용하기 쉽고 데이터를 읽어오는데 걸리는 시간(접근시간, access time)이 가장 빠르다는 장점을 가지고 있지만 다음과 같은 단점도 가지고 있다.

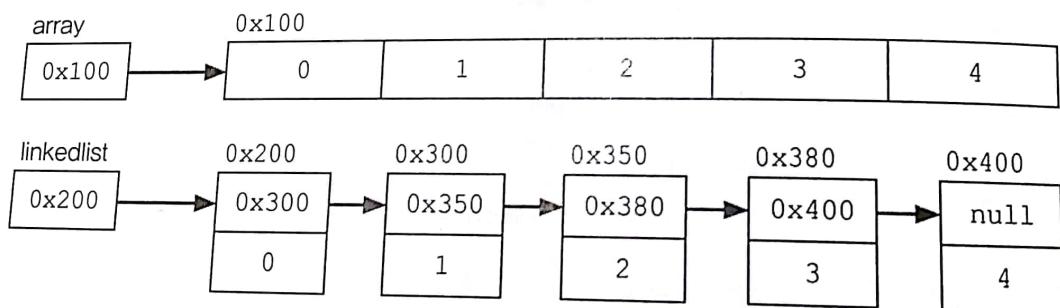
1. 크기를 변경할 수 없다.

- 크기를 변경할 수 없으므로 새로운 배열을 생성해서 데이터를 복사해야 한다.
- 실행속도를 향상시키기 위해서는 충분히 큰 크기의 배열을 생성해야 하므로 메모리가 낭비된다.

2. 비순차적인 데이터의 추가 또는 삭제에 시간이 많이 걸린다.

- 차례대로 데이터를 추가하고 마지막에서부터 데이터를 삭제하는 것은 빠르지만,
- 배열의 중간에 데이터를 추가하려면, 빈자리를 만들기 위해 다른 데이터들을 복사해서 이동해야 한다.

이러한 배열의 단점을 보완하기 위해서 링크드 리스트(linked list)라는 자료구조가 고안되었다. 배열은 모든 데이터가 연속적으로 존재하지만 링크드 리스트는 불연속적으로 존재하는 데이터를 서로 연결(link)한 형태로 구성되어 있다.



▲ 그림 11-7 배열과 링크드 리스트

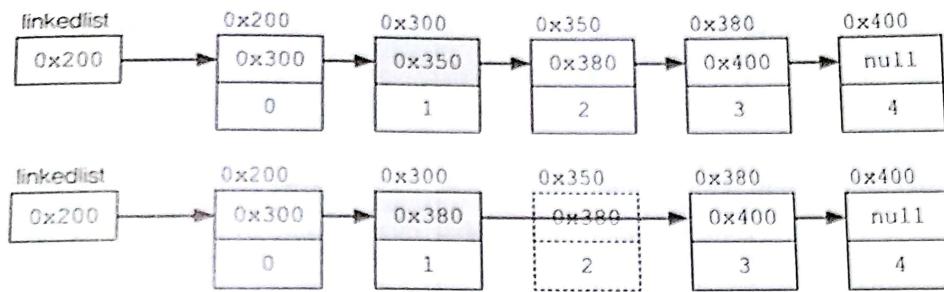
위의 그림에서 알 수 있듯이 링크드 리스트의 각 요소(node)들은 자신과 연결된 다음 요소에 대한 참조(주소값)와 데이터로 구성되어 있다.

```

class Node {
    Node next;      // 다음 요소의 주소를 저장
    Object obj;     // 데이터를 저장
}

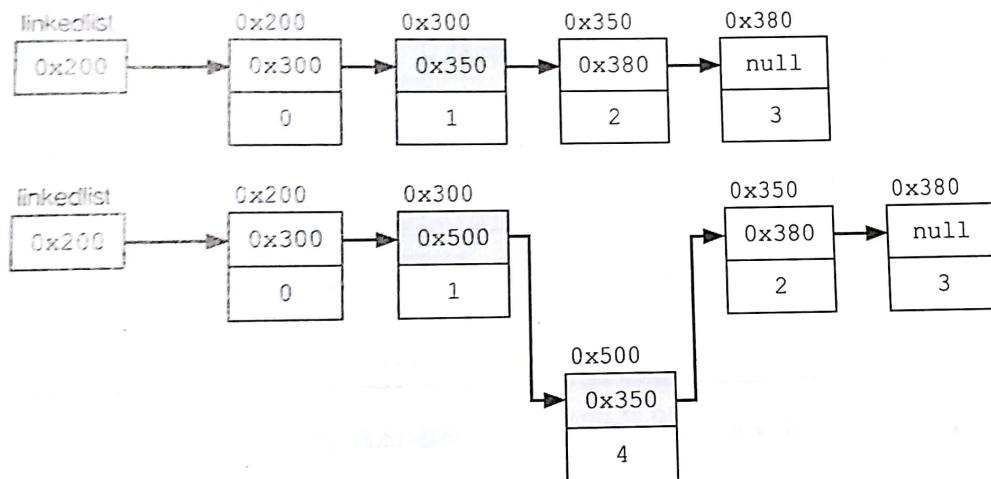
```

링크드 리스트에서의 데이터 삭제는 간단하다. 삭제하고자 하는 요소의 이전요소가 삭제하면 삭제가 이루어지는 것이다. 배열처럼 데이터를 이동하기 위해 복사하는 과정이 없기 때문에 처리속도가 매우 빠르다.



▲ 그림 11-8 링크드 리스트에서의 데이터 삭제

새로운 데이터를 추가할 때는 새로운 요소를 생성한 다음 추가하고자 하는 위치의 이전 요소의 참조를 새로운 요소에 대한 참조로 변경해주고, 새로운 요소가 그 다음 요소를 참조하도록 변경하기만 하면 되므로 처리속도가 매우 빠르다.



▲ 그림 11-9 링크드 리스트에서의 데이터 추가

링크드 리스트는 이동방향이 단방향이기 때문에 다음 요소에 대한 접근은 쉽지만 이전요소에 대한 접근은 어렵다. 이 점을 보완한 것이 더블 링크드 리스트(이중 연결리스트, doubly linked list)이다.

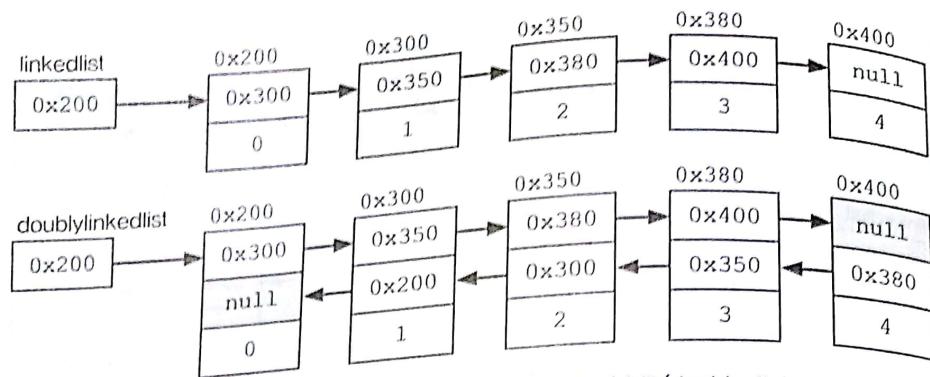
더블 링크드 리스트는 단순히 링크드 리스트에 참조변수를 하나 더 추가하여 다음 요소에 대한 참조뿐 아니라 이전 요소에 대한 참조가 가능하도록 했을 뿐, 그 외에는 링크드 리스트와 같다.

더블 링크드 리스트는 링크드 리스트보다 각 요소에 대한 접근과 이동이 쉽기 때문에 링크드 리스트보다 더 많이 사용된다.

```

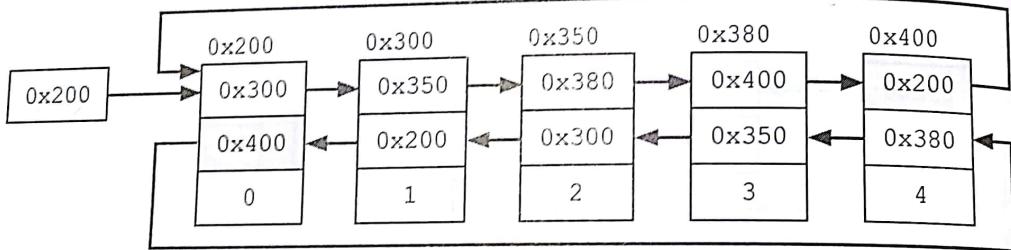
class Node {
    Node next; // 다음 요소의 주소를 저장
    Node previous; // 이전 요소의 주소를 저장
    Object obj; // 데이터를 저장
}

```



▲ 그림 11-10 링크드 리스트(linked list)와 더블 링크드 리스트(doubly-linked list)

더블 링크드 리스트의 접근성을 보다 향상시킨 것이 ‘더블 써큘러 링크드 리스트(이중 원형 연결리스트, doubly circular linked list)’인데, 단순히 더블 링크드 리스트의 첫 번째 요소와 마지막 요소를 서로 연결시킨 것이다. 이렇게 하면, 마지막 요소의 다음 요소가 첫 번째 요소가 되고, 첫 번째 요소의 이전 요소가 마지막 요소가 된다. 마치 TV의 마지막 채널에서 채널을 증가시키면 첫 번째 채널로 이동하고 첫 번째 채널에서 채널을 감소시키면 마지막 채널로 이동하는 것과 같다.



▲ 그림 11-11 더블 써큘러 링크드리스트(이중 원형 연결리스트, doubly circular linked list)

실제로 `LinkedList` 클래스는 이름과 달리 ‘링크드 리스트’가 아닌 ‘더블 링크드 리스트’로 구현되어 있는데, 이는 링크드 리스트의 단점인 낮은 접근성(accessability)을 높이기 위한 것이다. 지금까지 링크드 리스트의 기본원리에 대해서 살펴보았는데, 이제 본론으로 들어가 `LinkedList` 클래스에 대해 알아보도록 하자.

생성자 또는 메서드	설명
<code>LinkedList()</code>	<code>LinkedList</code> 객체를 생성
<code>LinkedList(Collection c)</code>	주어진 컬렉션을 포함하는 <code>LinkedList</code> 객체를 생성
<code>boolean add(Object o)</code>	지정된 객체(<code>o</code>)를 <code>LinkedList</code> 의 끝에 추가. 저장에 성공하면 <code>true</code> , 실패하면 <code>false</code>
<code>void add(int index, Object element)</code>	지정된 위치(<code>index</code>)에 객체(<code>element</code>)를 추가
<code>boolean addAll(Collection c)</code>	주어진 컬렉션에 포함된 모든 요소를 <code>LinkedList</code> 의 끝에 추가한다. 성공하면 <code>true</code> , 실패하면 <code>false</code>
<code>boolean addAll(int index, Collection c)</code>	지정된 위치(<code>index</code>)에 주어진 컬렉션에 포함된 모든 요소를 추가. 성공하면 <code>true</code> , 실패하면 <code>false</code>
<code>void clear()</code>	<code>LinkedList</code> 의 모든 요소를 삭제
<code>boolean contains(Object o)</code>	지정된 객체가 <code>LinkedList</code> 에 포함되었는지 알려줌
<code>boolean containsAll(Collection c)</code>	지정된 컬렉션의 모든 요소가 포함되었는지 알려줌

Object get(int index)	지정된 위치(index)의 객체를 반환
int indexOf(Object o)	지정된 객체가 저장된 위치(앞에서 몇 번째)를 반환
boolean isEmpty()	LinkedList가 비어있는지 알려준다. 비어있으면 true
Iterator iterator()	Iterator를 반환한다.
int lastIndexOf(Object o)	지정된 객체의 위치(index)를 반환(끝부터 역순검색)
ListIterator listIterator()	ListIterator를 반환한다.
ListIterator listIterator(int index)	지정된 위치에서부터 시작하는 ListIterator를 반환
Object remove(int index)	지정된 위치(index)의 객체를 LinkedList에서 제거
boolean remove(Object o)	지정된 객체를 LinkedList에서 제거. 성공하면 true, 실패하면 false
boolean removeAll(Collection c)	지정된 컬렉션의 요소와 일치하는 요소를 모두 삭제
boolean retainAll(Collection c)	지정된 컬렉션의 모든 요소가 포함되어 있는지 확인
Object set(int index, Object element)	지정된 위치(index)의 객체를 주어진 객체로 바꿈
int size()	LinkedList에 저장된 객체의 수를 반환
List subList(int fromIndex, int toIndex)	LinkedList의 일부를 List로 반환
Object[] toArray()	LinkedList에 저장된 객체를 배열로 반환
Object[] toArray(Object[] a)	LinkedList에 저장된 객체를 주어진 배열에 저장하여 반환
Object element()	LinkedList의 첫 번째 요소를 반환
boolean offer(Object o)	지정된 객체(o)를 LinkedList의 끝에 추가. 성공하면 true, 실패하면 false
Object peek()	LinkedList의 첫 번째 요소를 반환
Object poll()	LinkedList의 첫 번째 요소를 반환. LinkedList에서는 제거된다.
Object remove()	LinkedList의 첫 번째 요소를 제거
void addFirst(Object o)	LinkedList의 맨 앞에 객체(o)를 추가
void addLast(Object o)	LinkedList의 맨 끝에 객체(o)를 추가
Iterator descendingIterator()	역순으로 조회하기 위한 DescendingIterator를 반환
Object getFirst()	LinkedList의 첫번째 요소를 반환
Object getLast()	LinkedList의 마지막 요소를 반환
boolean offerFirst(Object o)	LinkedList의 맨 앞에 객체(o)를 추가. 성공하면, true
boolean offerLast(Object o)	LinkedList의 맨 끝에 객체(o)를 추가. 성공하면, true
Object peekFirst()	LinkedList의 첫번째 요소를 반환
Object peekLast()	LinkedList의 마지막 요소를 반환
Object pollFirst()	LinkedList의 첫번째 요소를 반환하면서 제거
Object pollLast()	LinkedList의 마지막 요소를 반환하면서 제거
Object pop()	removeFirst()와 동일
void push(Object o)	addFirst()와 동일
Object removeFirst()	LinkedList의 첫번째 요소를 제거
Object removeLast()	LinkedList의 마지막 요소를 제거
boolean removeFirstOccurrence(Object o)	LinkedList에서 첫번째로 일치하는 객체를 제거
boolean removeLastOccurrence(Object o)	LinkedList에서 마지막으로 일치하는 객체를 제거

▲ 표 11-7 LinkedList의 생성자와 메서드

! 참고 I. LinkedList는 Queue인터페이스(JDK1.5)와 Deque인터페이스(JDK1.6)를 구현하도록 변경되었는데, 표11-7의 마지막 22개의 메서드는 Queue인터페이스(회색바탕)와 Deque인터페이스(흰색바탕)를 구현하면서 추가된 것이다.

LinkedList 역시 List인터페이스를 구현했기 때문에 ArrayList와 내부구현방법만 다를 뿐 제공하는 메서드의 종류와 기능은 거의 같기 때문에 이에 대한 설명은 따로 필요없을 것 같다.

대신 두 가지 예제를 통해서 ArrayList와 LinkedList의 성능차이를 비교하고, 그 결과를 설명하고자 한다. 이 예제들을 통해서 ArrayList와 LinkedList의 장단점을 이해하고 필요한 상황에 적절한 것을 선택해서 사용할 수 있도록 하자.

▼ 예제 11-5/ch11/ArrayListLinkedListTest.java

```
import java.util.*;  
  
public class ArrayListLinkedListTest {  
    public static void main(String args[]) {  
        // 추가할 데이터의 개수를 고려하여 충분히 많아야한다.  
        ArrayList al = new ArrayList(2000000);  
        LinkedList ll = new LinkedList();  
  
        System.out.println("= 순차적으로 추가하기 =");  
        System.out.println("ArrayList :" + add1(al));  
        System.out.println("LinkedList :" + add1(ll));  
        System.out.println();  
  
        System.out.println("= 중간에 추가하기 =");  
        System.out.println("ArrayList :" + add2(al));  
        System.out.println("LinkedList :" + add2(ll));  
        System.out.println();  
  
        System.out.println("= 중간에서 삭제하기 =");  
        System.out.println("ArrayList :" + remove2(al));  
        System.out.println("LinkedList :" + remove2(ll));  
        System.out.println();  
  
        System.out.println("= 순차적으로 삭제하기 =");  
        System.out.println("ArrayList :" + remove1(al));  
        System.out.println("LinkedList :" + remove1(ll));  
    }  
  
    public static long add1(List list) {  
        long start = System.currentTimeMillis();  
        for(int i=0; i<1000000; i++) list.add(i+"");  
        long end = System.currentTimeMillis();  
        return end - start;  
    }  
  
    public static long add2(List list) {  
        long start = System.currentTimeMillis();  
        for(int i=0; i<10000; i++) list.add(500, "X");  
        long end = System.currentTimeMillis();  
        return end - start;  
    }  
  
    public static long remove1(List list) {  
        long start = System.currentTimeMillis();  
        for(int i=list.size()-1; i >= 0; i--) list.remove(i);  
        long end = System.currentTimeMillis();  
        return end - start;  
    }
```

```

        public static long remove2(List list) {
            long start = System.currentTimeMillis();
            for(int i=0; i<10000;i++) list.remove(i);
            long end = System.currentTimeMillis();
            return end - start;
        }
    }

```

▼ 실행결과

= 순차적으로 추가하기 =
 ArrayList :284
 LinkedList :406

= 중간에 추가하기 =
 ArrayList :3453
 LinkedList :16

= 중간에서 삭제하기 =
 ArrayList :2641
 LinkedList :234

= 순차적으로 삭제하기 =
 ArrayList :0
 LinkedList :31

보다 명확한 차이를 얻기 위해 데이터의 개수를 크게 설정했다.

결론 1 순차적으로 추가/삭제하는 경우에는 ArrayList가 LinkedList보다 빠르다.

단순히 저장하는 시간만을 비교할 수 있도록 하기 위해서 ArrayList를 생성할 때는 저장할 데이터의 개수만큼 충분한 초기용량을 확보해서, 저장공간이 부족해서 새로운 ArrayList를 생성해야하는 상황이 일어나지 않도록 했다. 만일 ArrayList의 크기가 충분하지 않으면, 새로운 크기의 ArrayList를 생성하고 데이터를 복사하는 일이 발생하게 되므로 순차적으로 데이터를 추가해도 ArrayList보다 LinkedList가 더 빠를 수 있다.

순차적으로 삭제한다는 것은 마지막 데이터부터 역순으로 삭제해나간다는 것을 의미하며, ArrayList는 마지막 데이터부터 삭제할 경우 각 요소들의 재배치가 필요하지 않기 때문에 상당히 빠르다. 단지 마지막 요소의 값을 null로만 바꾸면 되니까.

결론 2 중간 데이터를 추가/삭제하는 경우에는 LinkedList가 ArrayList보다 빠르다.

중간 요소를 추가 또는 삭제하는 경우, LinkedList는 각 요소간의 연결만 변경해주면 되기 때문에 처리속도가 상당히 빠르다. 반면에 ArrayList는 각 요소들을 재배치하여 추가할 공간을 확보하거나 빈 공간을 채워야하기 때문에 처리속도가 늦다.

예제에서는 ArrayList와 LinkedList의 차이를 비교하기 위해 데이터의 개수를 크게 잡았는데, 사실 데이터의 개수가 그리 크지 않다면 어느 것을 사용해도 큰 차이가 나지는 않는다. 그래도 ArrayList와 LinkedList의 장단점을 잘 이해하고 상황에 따라 적합한 것을 선택해서 사용하는 것이 좋다.

| 참고 | 위의 예제에서 데이터의 개수와 ArrayList의 초기용량의 크기를 변경해가며 테스트해보고 결과를 비교분석해보자.

▼ 예제 11-6/ch11/ArrayListLinkedListTest2.java

```
import java.util.*;  
public class ArrayListLinkedListTest2 {  
    public static void main(String args[]) {  
        ArrayList al = new ArrayList(1000000);  
        LinkedList ll = new LinkedList();  
        add(al);  
        add(ll);  
        System.out.println("= 접근시간테스트 =");  
        System.out.println("ArrayList :" + access(al));  
        System.out.println("LinkedList :" + access(ll));  
    }  
    public static void add(List list) {  
        for(int i=0; i<100000; i++) list.add(i+"");  
    }  
    public static long access(List list) {  
        long start = System.currentTimeMillis();  
        for(int i=0; i<10000; i++) list.get(i);  
        long end = System.currentTimeMillis();  
        return end - start;  
    }  
}
```

▼ 실행결과

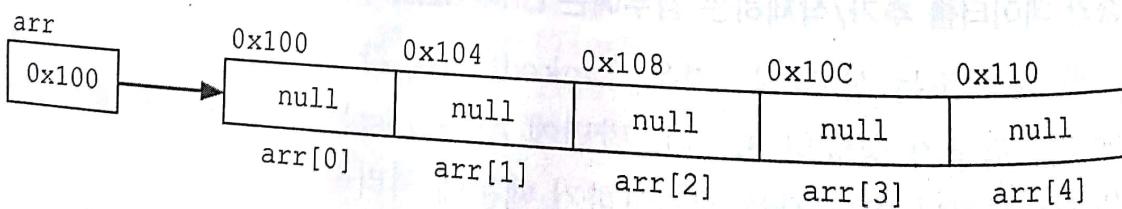
```
= 접근시간테스트 =  
ArrayList :1  
LinkedList :317
```

배열의 경우 만일 인덱스가 n인 원소의 값을 얻어 오고자 한다면 단순히 아래와 같은 수식을 계산함으로써 해결된다.

$$\text{인덱스가 } n\text{인 데이터의 주소} = \text{배열의 주소} + n * \text{데이터 타입의 크기}$$

아래와 같이 Object배열이 선언되었을 때 arr[2]에 저장된 값을 읽으려 한다면 n은 2, 모든 참조형 변수의 크기는 4byte이고 생성된 배열의 주소는 0x100이므로 3번째 데이터가 저장되어 있는 주소는 $0x100 + 2 * 4 = 0x108$ 이 된다.

```
Object[] arr = new Object[5];
```



▲ 그림 11-12 배열의 메모리구조

배열은 각 요소들이 연속적으로 메모리상에 존재하기 때문에 이처럼 간단한 계산만으로 원하는 요소의 주소를 얻어서 저장된 데이터를 곧바로 읽어올 수 있지만, LinkedList는

불연속적으로 위치한 각 요소들이 서로 연결된 것이라 처음부터 n 번째 데이터까지 차례대로 따라가야만 원하는 값을 얻을 수 있다.

그래서 `LinkedList`는 저장해야하는 데이터의 개수가 많아질수록 데이터를 읽어 오는 시간, 즉 접근시간(access time)이 길어진다는 단점이 있다.

컬렉션	읽기(접근시간)	추가 / 삭제	비고
<code>ArrayList</code>	빠르다	느리다	순차적인 추가삭제는 더 빠름. 비효율적인 메모리 사용
<code>LinkedList</code>	느리다	빠르다	데이터가 많을수록 접근성이 떨어짐

▲ 표 11-8 `ArrayList`과 `LinkedList`의 비교

다루고자 하는 데이터의 개수가 변하지 않는 경우라면, `ArrayList`가 최상의 선택이 되겠지만, 데이터 개수의 변경이 갖다면 `LinkedList`를 사용하는 것이 더 나은 선택이 될 것이다.

두 클래스의 장점을 이용해서 두 클래스를 조합해서 사용하는 방법도 생각해 볼 수 있다. 처음에 작업하기 전에 데이터를 저장할 때는 `ArrayList`를 사용한 다음, 작업할 때는 `LinkedList`로 데이터를 옮겨서 작업하면 좋은 효율을 얻을 수 있을 것이다.

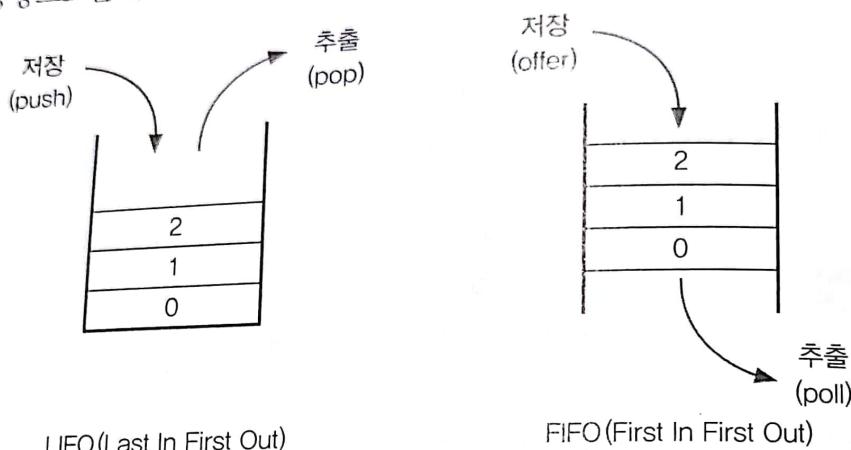
```
ArrayList al = new ArrayList(1000000);
for(int i=0; i<100000; i++) al.add(i+"");
LinkedList ll = new LinkedList(al);
for(int i=0; i<1000; i++) ll.add(500, "X");
```

컬렉션 프레임워크 속한 대부분의 컬렉션 클래스들은 이처럼 서로 변환이 가능한 생성자를 제공하므로 이를 이용하면 간단히 다른 컬렉션 클래스로 데이터를 옮길 수 있다.

1.4 Stack과 Queue

자바에서 제공하는 Stack과 Queue에 대해서 알아보기 이전에 스택(stack)과 큐(queue)

의 기본 개념과 특징에 대해서 먼저 살펴보도록 하자.
스택은 마지막에 저장한 데이터를 가장 먼저 꺼내게 되는 LIFO(Last In First Out) 구조로 되어 있고, 큐는 처음에 저장한 데이터를 가장 먼저 꺼내게 되는 FIFO(First In First Out) 구조로 되어 있다. 쉽게 얘기하자면 스택은 동진통과 같은 구조로 양 옆과 바닥이 막혀 있어서 한 방향으로만 빼 수 있는 구조이고, 큐는 양 옆만 막혀 있고 위아래로 뚫려 있어서 한 방향으로는 넣고 한 방향으로는 빼는 파이프와 같은 구조로 되어 있다.



▲ 그림 11-13 스택(stack)과 큐(queue)

예를 들어 스택에 0, 1, 2의 순서로 데이터를 넣었다면 꺼낼 때는 2, 1, 0의 순서로 꺼내게 된다. 즉, 넣은 순서와 꺼낸 순서가 뒤집어지게 되는 것이다. 이와 반대로 큐에 0, 1, 2의 순서로 데이터를 넣었다면 꺼낼 때 역시 0, 1, 2의 순서로 꺼내게 된다. 순서의 변경 없이 먼저 넣은 것을 먼저 꺼내게 되는 것이다.

그렇다면 스택과 큐를 구현하기 위해서는 어떤 컬렉션 클래스를 사용하는 것이 좋을까? 순차적으로 데이터를 추가하고 삭제하는 스택에는 ArrayList와 같은 배열기반의 컬렉션 클래스가 적합하지만, 큐는 데이터를 꺼낼 때 항상 첫 번째 저장된 데이터를 삭제하므로, ArrayList와 같은 배열기반의 컬렉션 클래스를 사용한다면 데이터를 꺼낼 때마다 빈 공간을 채우기 위해 데이터의 복사가 발생하므로 비효율적이다. 그래서 큐는 ArrayList보다 데이터의 추가/삭제가 쉬운 LinkedList로 구현하는 것이 더 적합하다.

메서드	설명
boolean empty()	Stack이 비어있는지 알려준다.
Object peek()	Stack의 맨 위에 저장된 객체를 반환. pop()과 달리 Stack에서 객체를 꺼내지는 않음. (비었을 때는 EmptyStackException 발생)
Object pop()	Stack의 맨 위에 저장된 객체를 꺼낸다. (비었을 때는 EmptyStackException 발생)
Object push(Object item)	Stack에 객체(item)를 저장한다.
int search(Object o)	Stack에서 주어진 객체(o)를 찾아서 그 위치를 반환. 못 찾으면 -1을 반환. (배열과 달리 위치는 0이 아닌 1부터 시작)

메서드	설명
boolean add(Object o)	지정된 객체를 Queue에 추가한다. 성공하면 true를 반환. 저장공간이 부족하면 IllegalStateException 발생
Object remove()	Queue에서 객체를 꺼내 반환. 비어있으면 NoSuchElementException 발생
Object element()	삭제없이 요소를 읽어온다. peek와 달리 Queue가 비었을 때 NoSuchElementException 발생
boolean offer(Object o)	Queue에 객체를 저장. 성공하면 true, 실패하면 false를 반환
Object poll()	Queue에서 객체를 꺼내서 반환. 비어있으면 null을 반환
Object peek()	삭제없이 요소를 읽어온다. Queue가 비어있으면 null을 반환

▲ 표 11-10 Queue의 메서드

▼ 예제 11-7/ch11/StackQueueEx.java

```

import java.util.*;

class StackQueueEx {
    public static void main(String[] args) {
        Stack st = new Stack();
        Queue q = new LinkedList(); // Queue인터페이스의 구현체인 LinkedList를 사용

        st.push("0");
        st.push("1");
        st.push("2");

        q.offer("0");
        q.offer("1");
        q.offer("2");

        System.out.println("= Stack =");
        while(!st.empty()) {
            System.out.println(st.pop());
        }

        System.out.println("= Queue =");
        while(!q.isEmpty()) {
            System.out.println(q.poll());
        }
    }
}

```

▼ 실행결과

```

= Stack =
2
1
0
= Queue =
0
1
2

```

스택과 큐에 각각 “0”, “1”, “2”를 같은 순서로 넣고 꺼내었을 때의 결과가 다른 것을 알 수 있다. 큐는 먼저 넣은 것이 먼저 꺼내지는 구조(FIFO)이기 때문에 넣을 때와 같은 순서이고, 스택은 먼저 넣은 것이 나중에 꺼내지는 구조(LIFO)이기 때문에 넣을 때의 순서와 반대로 꺼내진 것을 알 수 있다.

자바에서는 스택을 Stack 클래스로 구현하여 제공하고 있지만 큐는 Queue 인터페이스로만 정의해 놓았을 뿐 별도의 클래스를 제공하고 있지 않다. 대신 Queue 인터페이스를 구현한 클래스들이 있어서 이들 중의 하나를 선택해서 사용하면 된다.

메서드	설명
boolean add(Object o)	저장된 객체를 Queue에 추가한다. 성공하면 true를 반환. 저장공간이 부족하면 IllegalStateException 발생
Object remove()	Queue에서 객체를 끄내 반환. 비어있으면 NoSuchElementException 발생
Object element()	삭제없이 요소를 읽어온다. peek와 달리 Queue가 비었을 때 NoSuchElementException 발생
boolean offer(Object o)	Queue에 객체를 저장. 성공하면 true, 실패하면 false를 반환
Object poll()	Queue에서 객체를 끄내서 반환. 비어있으면 null을 반환
Object peek()	삭제없이 요소를 읽어 온다. Queue가 비어있으면 null을 반환

▲ 표 11-10 Queue의 메서드

▼ 예제 11-7/ch11/StackQueueEx.java

```
import java.util.*;

class StackQueueEx {
    public static void main(String[] args) {
        Stack st = new Stack();
        Queue q = new LinkedList(); // Queue인터페이스의 구현체인 LinkedList를 사용

        st.push("0");
        st.push("1");
        st.push("2");

        q.offer("0");
        q.offer("1");
        q.offer("2");

        System.out.println("= Stack =");
        while(!st.empty()) {
            System.out.println(st.pop());
        }

        System.out.println("= Queue =");
        while(!q.isEmpty()) {
            System.out.println(q.poll());
        }
    }
}
```

▼ 실행결과

```
= Stack =
2
1
0
= Queue =
0
1
2
```

스택과 큐에 각각 “0”, “1”, “2”를 같은 순서로 넣고 끄내었을 때의 결과가 다른 것을 알 수 있다. 큐는 먼저 넣은 것이 먼저 끄내지는 구조(FIFO)이기 때문에 넣을 때와 같은 순서이고, 스택은 먼저 넣은 것이 나중에 끄내지는 구조(LIFO)이기 때문에 넣을 때의 순서와 반대로 끄내진 것을 알 수 있다.

자바에서는 스택을 Stack 클래스로 구현하여 제공하고 있지만 큐는 Queue 인터페이스로만 정의해 놓았을 뿐 별도의 클래스를 제공하고 있지 않다. 대신 Queue 인터페이스를 구현한 클래스들이 있어서 이들 중의 하나를 선택해서 사용하면 된다.

■ 알아두면 좋아요! – 인터페이스를 구현한 클래스 찾기

위의 예제에서와 같이 Queue인터페이스의 기능을 사용하고자 할 때는 다음과 같이 Java API문서를 참고하면 된다. 아래 그림의 하단에 보면 'All Known Implementing Classes'라는 항목이 있는데 여기에 나열된 클래스들이 바로 Queue인터페이스를 구현한 클래스들이다.

```
java.util
Interface Queue<E>

Type Parameters:
E - the type of elements held in this collection

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque,
ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue,
LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue,
SynchronousQueue
```

▲ 그림 11-14 Java API문서에서 찾은 Queue

각 클래스들은 각자 나름대로의 용도가 있겠지만 적어도 Queue인터페이스에 정의된 메서드를 모두 작성해 놓았으며, 이 메서드들에 대해서는 내용은 좀 다를 수 있겠지만 대부분 거의 같은 기능을 한다. 그래서 Queue인터페이스에 정의된 기능을 사용하고 싶다면, 'All known Implementing Classes'에 적혀있는 클래스들 중에서 적당한 것을 하나 골라 잡아서 'Queue q = new LinkedList();'와 같은 식으로 객체를 생성해서 사용하면 된다.

Stack 직접 구현하기

Stack은 컬렉션 프레임워크 이전부터 존재하던 것이기 때문에 ArrayList가 아닌 Vector로부터 상속받아 구현하였다. Stack의 실제코드를 이해하기 쉽게 약간 수정해서 MyStack을 만들어 보았다. 스택을 자바코드로 어떻게 구현하였는지 이해하는데 많은 도움이 될 것이다.

▼ 예제 11-8/ch11/MyStack.java

```
import java.util.*;
class MyStack extends Vector {
    public Object push(Object item) {
        addElement(item);
        return item;
    }
    public Object pop() {
        Object obj = peek();
        // Stack에 저장된 마지막 Object를 반환
        return obj;
    }
}
```

```

// 만약 Stack이 비어있으면 peek() 메서드가 EmptyStackException을 발생시킨다.
// 마지막 요소를 삭제한다. 배열의 index가 0부터 시작하므로 1을 빼준다.
removeElementAt(size() - 1);
return obj;
}

public Object peek() {
    int len = size();

    if (len == 0)
        throw new EmptyStackException();
    // 마지막 요소를 반환한다. 배열의 index가 0부터 시작하므로 1을 빼준다.
    return elementAt(len - 1);
}

public boolean empty() {
    return size() == 0;
}

public int search(Object o) {
    int i = lastIndexOf(o);           // 끝에서부터 객체를 찾는다.
                                       // 반환값은 저장된 위치(배열의 index)이다.

    if (i >= 0) { // 객체를 찾은 경우
        return size() - i; // Stack은 맨 위에 저장된 객체의 index를 1로 정의하기 때문에
                           // 계산을 통해서 구한다.
    }
    return -1; // 해당 객체를 찾지 못하면 -1을 반환한다.
}
}

```

Vector에 구현되어 있는 메서드를 이용하기 때문에 코드도 간단하고 어렵지 않다. 추가적인 설명이 없어도 이해하는데 어려움이 있으리라 생각한다.

| 참고 | EmptyStackException은 RuntimeException으로 따로 예외처리를 해주지 않아도 된다.

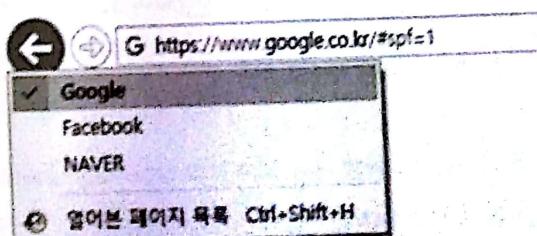
스택과 큐의 활용

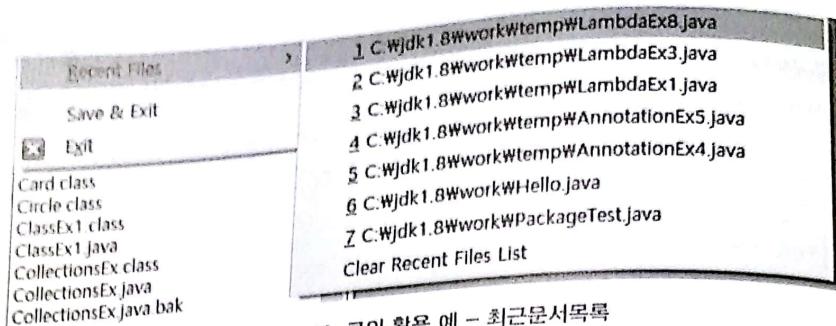
지금까지 스택과 큐의 개념과 구현에 대해서 알아보았는데 이제는 스택과 큐를 어떻게 활용할 것인가에 대해서 살펴보자.

우리가 쉽게 찾아볼 수 있는 스택과 큐의 활용 예는 다음과 같다.

스택의 활용 예 - 수식계산, 수식괄호검사, 워드프로세서의 undo/redo,
웹브라우저의 뒤로/앞으로

큐의 활용 예 - 최근사용문서, 인쇄작업 대기목록, 버퍼(buffer)





스택과 큐는 실제 프로그래밍에서 빈번하게 사용되는 자료구조라고는 하지만 어디에 사용되었고 막상 어떻게 활용해야 할지를 생각해보면 잘 떠오르지 않을 것이다.

이제 스택과 큐를 어떻게 활용하는지 감을 잡을 수 있는 예제를 몇 가지 학습해보고 나면 스택과 큐를 이해하고 활용하는데 더 많은 도움이 될 것이다.

▼ 예제 11-9/ch11/StackEx1.java

```
import java.util.*;

public class StackEx1 {
    public static Stack back = new Stack();
    public static Stack forward = new Stack();

    public static void main(String[] args) {
        goURL("1.네이트");
        goURL("2.야후");
        goURL("3.네이버");
        goURL("4.다음");

        printStatus();

        goBack();
        System.out.println("= 뒤로가기 버튼을 누른 후 =");
        printStatus();

        goBack();
        System.out.println("= '뒤로' 버튼을 누른 후 =");
        printStatus();

        goForward();
        System.out.println("= '앞으로' 버튼을 누른 후 =");
        printStatus();

        goURL("codechobo.com");
        System.out.println("= 새로운 주소로 이동 후 =");
        printStatus();

        public static void printStatus() {
            System.out.println("back:" + back);
            System.out.println("forward:" + forward);
        }
    }
}
```

```

        System.out.println("현재화면은 '" + back.peek() + "' 입니다.");
        System.out.println();
    }

    public static void goURL(String url){
        back.push(url);
        if(!forward.empty())
            forward.clear();
    }

    public static void goForward(){
        if(!forward.empty())
            back.push(forward.pop());
    }

    public static void goBack(){
        if(!back.empty())
            forward.push(back.pop());
    }
}

```

▼ 실행결과

back: [1.네이트, 2.야후, 3.네이버, 4.다음]

forward: []

현재화면은 '4.다음' 입니다.

= 뒤로가기 버튼을 누른 후 =

back: [1.네이트, 2.야후, 3.네이버]

forward: [4.다음]

현재화면은 '3.네이버' 입니다.

= '뒤로' 버튼을 누른 후 =

back: [1.네이트, 2.야후]

forward: [4.다음]

현재화면은 '2.야후' 입니다.

= '앞으로' 버튼을 누른 후 =

back: [1.네이트, 2.야후, 3.네이버]

forward: [4.다음]

현재화면은 '3.네이버' 입니다.

= 새로운 주소로 이동 후 =

back: [1.네이트, 2.야후, 3.네이버, codechobo.com]

forward: []

현재화면은 'codechobo.com' 입니다.

이 예제는 웹브라우저의 '뒤로', '앞으로' 버튼의 기능을 구현한 것이다. 이 기능을 구현하기 위해서는 2개의 스택을 사용해야한다. 웹브라우저로 위의 예제에서 구성한 시나리오대로 동작해보면 쉽게 이해할 수 있을 것이다.

| 참고 | 워드프로세서의 되돌리기 기능(undo/redo) 역시 이와 같은 방식으로 되어있다.

▼ 예제 11-10/ch11/ExpValidCheck.java

```

import java.util.*;

public class ExpValidCheck {
    public static void main(String[] args) {
        if(args.length!=1){
            System.out.println("Usage : java ExpValidCheck \"EXPRESSION\"");
            System.out.println("Example : java ExpValidCheck \"((2+3)*1)+3\"");
            System.exit(0);
        }
        Stack st = new Stack();
        String expression = args[0];
        System.out.println("expression:"+expression);
        try {
            for(int i=0; i < expression.length(); i++){
                char ch = expression.charAt(i);

                if(ch=='('){
                    st.push(ch+"");
                } else if(ch==')') {
                    st.pop();
                }
            }
            if(st.isEmpty()){
                System.out.println("괄호가 일치합니다.");
            } else {
                System.out.println("괄호가 일치하지 않습니다.");
            }
        } catch (EmptyStackException e) {
            System.out.println("괄호가 일치하지 않습니다.");
        } // try
    }
}

```

▼ 실행결과

```

c:\jdk1.8\work\ch11>java ExpValidCheck
Usage : java ExpValidCheck "EXPRESSION"
Example: java ExpValidCheck "((2+3)*1)+3"
c:\jdk1.8\work\ch11>java ExpValidCheck (2+3)*1
expression:(2+3)*1
괄호가 일치합니다.

```

입력한 수식의 괄호가 올바른지를 체크하는 예제이다. '('를 만나면 스택에 넣고 ')'를 만나면 스택에서 '('를 꺼낸다. ')'를 만나서 '('를 꺼내려 할 때 스택이 비어있거나 수식을 검사하고 난 후에도 스택이 비어있지 않으면 괄호가 잘못된 것이다.

로 try-catch문을 이용해서 EmptyStackException이 발생하므로 예제를 출력하도록 했다.

▼ 예제 11-11/ch11/QueueEx1.java

```

import java.util.*;

class QueueEx1 {
    static Queue q = new LinkedList();
    static final int MAX_SIZE = 5; // Queue에 최대 5개까지만 저장되도록 한다.

    public static void main(String[] args) {
        System.out.println("help를 입력하면 도움말을 볼 수 있습니다.");
        while(true) {
            System.out.print(">>");
            try {
                // 화면으로부터 라인단위로 입력받는다.
                Scanner s = new Scanner(System.in);
                String input = s.nextLine().trim();

                if("").equals(input)) continue;

                if(input.equalsIgnoreCase("q")) {
                    System.exit(0);
                } else if(input.equalsIgnoreCase("help")) {
                    System.out.println(" help - 도움말을 보여줍니다.");
                    System.out.println(" q 또는 Q - 프로그램을 종료합니다.");
                    System.out.println(" history - 최근에 입력한 명령어를 "
                        + MAX_SIZE +"개 보여줍니다.");
                } else if(input.equalsIgnoreCase("history")) {
                    int i=0;
                    // 입력받은 명령어를 저장하고,
                    save(input);

                    // LinkedList의 내용을 보여준다.
                    LinkedList tmp = (LinkedList)q;
                    ListIterator it = tmp.listIterator();

                    while(it.hasNext())
                        System.out.println(++i+"."+it.next());
                } else {
                    save(input);
                    System.out.println(input);
                } // if(input.equalsIgnoreCase("q"))
            } catch(Exception e) {
                System.out.println("입력오류입니다.");
            }
        } // while(true)
    } // main()

    public static void save(String input) {
        // queue에 저장한다.
        if(!"".equals(input))
            q.offer(input);

        // queue의 최대크기를 넘으면 제일 처음 입력된 것을 삭제한다.
        if(q.size() > MAX_SIZE) // size()는 Collection인터페이스에 정의
            q.remove();
    }
} // end of class

```

▼ 실행결과

```
c:\jdk1.8\work\ch11>java QueueEx1
help를 입력하면 도움말을 볼 수 있습니다.
>>help
    help - 도움말을 보여줍니다.
    q 또는 Q - 프로그램을 종료합니다.
    history - 최근에 입력한 명령어를 5개 보여줍니다.
>>dir
dir
>>cd
cd
>>mkdir
mkdir
>>dir
dir
>>history
1.dir
2.cd
3.mkdir
4.dir
5.history
>>q
c:\jdk1.8\work\ch11>
```

이 예제는 유닉스의 history명령어를 Queue를 이용해서 구현한 것이다. history명령어는 사용자가 입력한 명령어의 이력을 순서대로 보여 준다. 여기서는 최근 5개의 명령어만을 보여주는데 MAX_SIZE의 값을 변경함으로써 더 많은 명령어 입력기록을 남길 수 있다. 대부분의 프로그램이 최근에 열어 본 문서들의 목록을 보여 주는 기능을 제공하는데, 이 기능도 위의 예제를 응용하면 쉽게 구현할 수 있을 것이다.

PriorityQueue

Queue인터페이스의 구현체 중의 하나로, 저장한 순서에 관계없이 우선순위(priority)가 높은 것부터 꺼내게 된다는 특징이 있다. 그리고 null은 저장할 수 없다. null을 저장하면 NullPointerException이 발생한다.

PriorityQueue는 저장공간으로 배열을 사용하며, 각 요소를 '힙(heap)'이라는 자료구조의 형태로 저장한다. 힙은 잠시 후에 배열 이진 트리의 한 종류로 가장 큰 값이나 가장 작은 값을 빠르게 찾을 수 있다는 특징이 있다.

참고 자료구조 힙(heap)은 앞서 배운 JVM의 힙(heap)과 이름만 같을 뿐은 다른 것이다.

▼ 예제 11-12/ch11/PriorityQueueEx.java

```
import java.util.*;
class PriorityQueueEx {
    public static void main(String[] args) {
        Queue pq = new PriorityQueue();
        pq.offer(3); // pq.offer(new Integer(3)); 오토박싱
        pq.offer(1);
        pq.offer(5);
        pq.offer(2);
        pq.offer(4);
        System.out.println(pq); // pq의 내부 배열을 출력
```

▼ 실행결과

```
c:\jdk1.8\work\ch11>java QueueEx1
help를 입력하면 도움말을 볼 수 있습니다.
>>help
    help - 도움말을 보여줍니다.
    q 또는 Q - 프로그램을 종료합니다.
    history - 최근에 입력한 명령어를 5개 보여줍니다.
>>dir
dir
>>cd
cd
>>mkdir
mkdir
>>dir
dir
>>history
1.dir
2.cd
3.mkdir
4.dir
5.history
>>q
c:\jdk1.8\work\ch11>
```

이 예제는 유닉스의 history 명령어를 Queue를 이용해서 구현한 것이다. history 명령어는 사용자가 입력한 명령어의 이력을 순서대로 보여 준다. 여기서는 최근 5개의 명령어만을 보여주는데 MAX_SIZE의 값을 변경함으로써 더 많은 명령어 입력기록을 남길 수 있다. 대부분의 프로그램이 최근에 열어 본 문서들의 목록을 보여 주는 기능을 제공하는데, 이 기능도 위의 예제를 응용하면 쉽게 구현할 수 있을 것이다.

PriorityQueue

Queue 인터페이스의 구현체 중의 하나로, 저장한 순서에 관계없이 우선순위(priority)가 높은 것부터 꺼내게 된다는 특징이 있다. 그리고 null은 저장할 수 없다. null을 저장하면 NullPointerException이 발생한다.

PriorityQueue는 저장공간으로 배열을 사용하며, 각 요소를 ‘힙(heap)’이라는 자료구조의 형태로 저장한다. 힙은 잠시 후에 배열을 이진 트리의 한 종류로 가장 큰 값이나 가장 작은 값을 빠르게 찾을 수 있다는 특징이 있다.

【참고】 자료구조 힙(heap)은 앞서 배운 JVM의 힙(heap)과 이름만 같을 뿐은 다른 것이다.

▼ 예제 11-12/ch11/PriorityQueueEx.java

```
import java.util.*;
class PriorityQueueEx {
    public static void main(String[] args) {
        Queue pq = new PriorityQueue();
        pq.offer(3); // pq.offer(new Integer(3)); 오토박싱
        pq.offer(1);
        pq.offer(5);
        pq.offer(2);
        pq.offer(4);
        System.out.println(pq); // pq의 내부 배열을 출력
```

```

Object obj = null;
// PriorityQueue에 저장된 요소를 하나씩 끄낸다.
while((obj = pq.poll())!=null)
    System.out.println(obj);
}
}

```

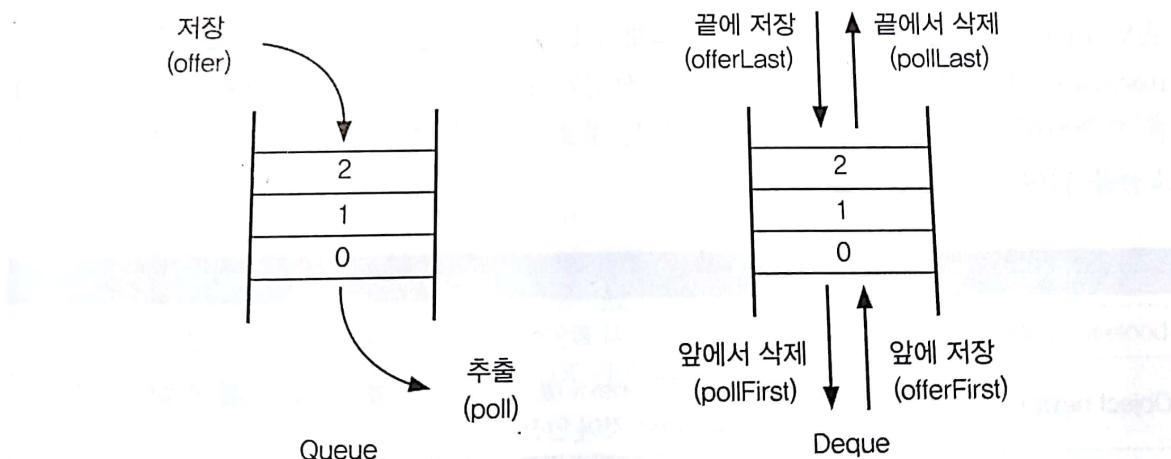
▼ 실행결과
[1, 2, 5, 3, 4]
1
2
3
4
5

저장순서가 3.1.5.2.4인데도 출력결과는 1,2,3,4,5이다. 우선순위는 숫자가 작을수록 높은 것이므로 1이 가장 먼저 출력된 것이다. 물론 숫자뿐만 아니라 객체를 저장할 수도 있는데 그럴 경우 각 객체의 크기를 비교할 수 있는 방법을 제공해야 한다. 예제에서는 정수를 사용했는데, 컴파일러가 Integer로 오토박싱 해준다. Integer와 같은 Number의 자손들은 자체적으로 숫자를 비교하는 방법을 정의하고 있기 때문에 비교 방법을 지정해 주지 않아도 된다.

참조변수 pq를 출력하면, PriorityQueue가 내부적으로 가지고 있는 배열의 내용이 출력되는데, 저장한 순서와 다르게 저장되었다. 앞서 설명한 것과 같이 힙이라는 자료구조의 형태로 저장된 것이라서 그렇다. 이 자료구조에 대한 설명은 책의 범위를 넘어서므로 자세한 설명은 생략한다.

Deque(Double-Ended Queue)

Queue의 변형으로, 한 쪽 끝으로만 추가/삭제할 수 있는 Queue와 달리, Deque(덱, 또는 디큐라고 읽음)은 양쪽 끝에 추가/삭제가 가능하다. Deque의 조상은 Queue이며, 구현체로는 ArrayDeque과 LinkedList 등이 있다.



▲ 그림 11-17 큐(queue)와 덱(deque)

덱은 스택과 큐를 하나로 합쳐놓은 것과 같으며 스택으로 사용할 수도 있고, 큐로 사용할 수도 있다. 위의 그림과 아래의 표를 같이 보면 어렵지 않게 이해할 수 있을 것이다.

Deque	Queue	Stack
offerLast()	offer()	push()
pollLast()	-	pop()
pollFirst()	poll()	-
peekFirst()	peek()	
peekLast()	-	peek()

▲ 표 11-11 데(deque)의 메서드에 대응하는 큐와 스택의 메서드

1.5 Iterator, ListIterator, Enumeration

Iterator, ListIterator, Enumeration은 모두 컬렉션에 저장된 요소를 접근하는데 사용되는 인터페이스이다. Enumeration은 Iterator의 구현이며, ListIterator는 Iterator의 기능을 향상 시킨 것이다.

Iterator

컬렉션 프레임워크에서는 컬렉션에 저장된 요소들을 읽어오는 방법을 표준화하였다. 컬렉션에 저장된 각 요소에 접근하는 기능을 가진 Iterator인터페이스를 정의하고, Collection 인터페이스에는 'Iterator(Iterator를 구현한 클래스의 인스턴스)'를 반환하는 iterator()를 정의하고 있다.

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}  
  
public interface Collection {  
    ...  
    public Iterator iterator();  
    ...  
}
```

iterator()는 Collection인터페이스에 정의된 메서드이므로 Collection인터페이스의 자손인 List와 Set에도 포함되어 있다. 그래서 List나 Set인터페이스를 구현하는 컬렉션은 iterator()가 각 컬렉션의 특징에 알맞게 작성되어 있다. 컬렉션 클래스에 대해 iterator()를 호출하여 Iterator를 얻은 다음 반복문, 주로 while문을 사용해서 컬렉션 클래스의 요소들을 읽어 올 수 있다.

메서드	설명
boolean hasNext()	읽어 올 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
Object next()	다음 요소를 읽어 온다. next()를 호출하기 전에 hasNext()를 호출해서 읽어 올 요소가 있는지 확인하는 것이 안전하다.
void remove()	next()로 읽어 온 요소를 삭제한다. next()를 호출한 다음에 remove()를 호출해야 한다.(선택적 기능)

▲ 표11-12 Iterator인터페이스의 메서드

ArrayList에 저장된 요소들을 출력하기 위한 코드는 다음과 같이 작성할 수 있다.

```
List list = new ArrayList(); // 다른 컬렉션으로 변경할 때는 이 부분만 고치면 된다.  
Iterator it = list.iterator();  
while(it.hasNext()) {  
    System.out.println(it.next());
```

ArrayList대신 List인터페이스를 구현한 다른 컬렉션 클래스에 대해서도 이와 동일한 코드를 사용할 수 있다. 첫 줄에서 ArrayList대신 List인터페이스를 구현한 다른 컬렉션 클래스의 객체를 생성하도록 변경하기만 하면 된다.

Iterator를 이용해서 컬렉션의 요소를 읽어오는 방법을 표준화했기 때문에 이처럼 코드의 재사용성을 높이는 것이 가능한 것이다. 이처럼 공통 인터페이스를 정의해서 표준을 정의하고 구현하여 표준을 따르도록 함으로써 코드의 일관성을 유지하여 재사용성을 극대화하는 것이 객체지향 프로그래밍의 중요한 목적 중의 하나이다.

Q. 참조변수의 타입이 ArrayList타입이 아니라 List타입으로 한 이유는 뭔가요?

A. List에 없고 ArrayList에만 있는 메서드를 사용하는 게 아니라면, List타입의 참조변수로 선언하는 것이 좋습니다. 만일 List인터페이스를 구현한 다른 클래스, 예를 들면 LinkedList로 바꿔야 한다면 선언문 하나만 변경하면 나머지 코드는 검토하지 않아도 됩니다. 참조변수의 타입이 List이므로 List에 정의되지 않은 메서드는 사용되지 않았을 것이 확실하기 때문입니다. 그러나 참조변수의 타입을 ArrayList로 했다면, 선언문 이후의 문장들을 검토해야 합니다. List에 정의되지 않은 메서드를 호출했을 수 있으니까요.

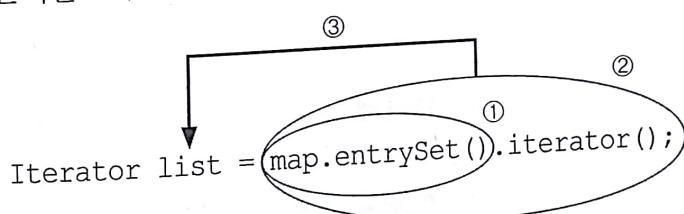
Map인터페이스를 구현한 컬렉션 클래스는 키(key)와 값(value)을 쌍(pair)으로 저장하고 있기 때문에 iterator()를 직접 호출할 수 없고, 그 대신 keySet()이나 entrySet()과 같은 메서드를 통해서 키와 값을 각각 따로 Set의 형태로 얻어 온 후에 다시 iterator()를 호출해야 Iterator를 얻을 수 있다.

```
Map map = new HashMap();
...
Iterator it = map.keySet().iterator();
```

Iterator list = map.entrySet().iterator(); 는 아래의 두 문장을 하나로 합친 것이라고 이해하면 된다.

```
Set eSet = map.entrySet();
Iterator list = eSet.iterator();
```

이 문장들의 실행순서를 그림으로 그려보면 다음과 같다.



① map.entrySet()의 실행결과가 Set이므로

Iterator list = map.entrySet().iterator(); → Iterator list = Set인스턴스.iterator();

② map.entrySet()를 통해 얻은 Set인스턴스의 iterator()를 호출해서 Iterator인스턴스를 얻는다;

Iterator list = Set인스턴스.iterator(); → Iterator list = Iterator인스턴스;

③ 마지막으로 Iterator인스턴스의 참조가 list에 저장된다.

StringBuffer를 사용할 때 이와 유사한 코드를 많이 보았을 것이다.

```
StringBuffer sb = new StringBuffer();
sb.append("A");
sb.append("B");
sb.append("C");
```

위와 같은 코드를 아래와 같이 간단히 쓸 수 있는데 그 이유는 바로 append메서드가 수행 결과로 StringBuffer를 리턴하기 때문이다. 만일 void append(String str)과 같이 void를 리턴하도록 선언되어 있다면 위의 코드를 아래와 같이 쓸 수 없었을 것이다. append메서드의 호출결과가 void이기 때문에 또 다시 void에 append메서드를 호출 할 수 없기 때문이다.

```
StringBuffer sb = new StringBuffer();
sb.append("A").append("B").append("C"); //StringBuffer append(String str)
```

▼ 예제 11-13/ch11/IteratorEx1.java

```
import java.util.*;

class IteratorEx1 {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        Iterator it = list.iterator();

        while(it.hasNext()) {
            Object obj = it.next();
            System.out.println(obj);
        }
    } // main
}
```

▼ 실행결과
1
2
3
4
5

List클래스들은 저장순서를 유지하기 때문에 Iterator를 이용해서 읽어 온 결과 역시 저장순서와 동일하지만 Set클래스들은 각 요소간의 순서가 유지 되지 않기 때문에 Iterator를 이용해서 저장된 요소들을 읽어 와도 처음에 저장된 순서와 같지 않다.

ListIterator와 Enumeration

Enumeration은 컬렉션 프레임워크 만들어지기 이전에 사용하던 것으로 Iterator의 구버전이라고 생각하면 된다. 이전 버전으로 작성된 소스와의 호환을 위해서 남겨 두고 있을 뿐이므로 가능하면 Enumeration 대신 Iterator를 사용하자.

ListIterator는 Iterator를 상속받아서 기능을 추가한 것으로, 컬렉션의 요소에 접근할 때 Iterator는 단방향으로만 이동할 수 있는데 반해 ListIterator는 양방향으로의 이동이 가능하다. 다만 ArrayList나 LinkedList와 같이 List 인터페이스를 구현한 컬렉션에서만 사용할 수 있다.

Enumeration Iterator의 구버전

ListIterator Iterator에 양방향 조회기능추가(List를 구현한 경우만 사용가능)

다음은 Enumeration, Iterator, ListIterator의 메서드에 대한 설명이다. Enumeration과 Iterator는 메서드 이름만 다를 뿐 기능은 같고, ListIterator는 Iterator에 이전방향으로의 접근기능을 추가한 것일 뿐이라는 것을 알 수 있다.

메서드	설명
boolean hasMoreElements()	읽어 올 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다. Iterator의 hasNext()와 같다.
Object nextElement()	다음 요소를 읽어 온다. nextElement()를 호출하기 전에 hasMoreElements()를 호출해서 읽어온 요소가 남아있는지 확인하는 것이 안전하다. Iterator의 next()와 같다.

▲ 표 11-13 Enumeration 인터페이스의 메서드

메서드	설명
void add(Object o)	컬렉션에 새로운 객체(o)를 추가한다.(선택적 기능)
boolean hasNext()	읽어 올 다음 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
boolean hasPrevious()	읽어 올 이전 요소가 남아있는지 확인한다. 있으면 true, 없으면 false를 반환한다.
Object next()	다음 요소를 읽어 온다. next()를 호출하기 전에 hasNext()를 호출해서 읽어온 요소가 있는지 확인하는 것이 안전하다.
Object previous()	이전 요소를 읽어 온다. previous()를 호출하기 전에 hasPrevious()를 호출해서 읽어온 요소가 있는지 확인하는 것이 안전하다.
int nextIndex()	다음 요소의 index를 반환한다.
int previousIndex()	이전 요소의 index를 반환한다.
void remove()	next() 또는 previous()로 읽어온 요소를 삭제한다. 반드시 next()나 previous()를 먼저 호출한 다음에 이 메서드를 호출해야 한다.(선택적 기능)
void set(Object o)	next() 또는 previous()로 읽어온 요소를 지정된 객체(o)로 변경한다. 반드시 next()나 previous()를 먼저 호출한 다음에 이 메서드를 호출해야 한다.(선택적 기능)

▲ 표 11-14 ListIterator의 메서드

▼ 예제 11-14/ch11/ListIteratorEx1.java

```

import java.util.*;

class ListIteratorEx1 {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        ListIterator it = list.listIterator();

        while(it.hasNext()) {
            System.out.print(it.next()); // 순방향으로 진행하면서 읽어온다.
        }
        System.out.println();

        while(it.hasPrevious()) {
            System.out.print(it.previous()); // 역방향으로 진행하면서 읽어온다.
        }
        System.out.println();
    }
}

```

▼ 실행결과
12345
54321

ListIterator의 사용방법을 보여주는 간단한 예제이다. Iterator는 단방향으로만 이동하기 때문에 컬렉션의 마지막 요소에 다다르면 더 이상 사용할 수 없지만, ListIterator는 양방향으로 이동하기 때문에 각 요소간의 이동이 자유롭다. 다만 이동하기 전에 반드시 hasNext()나 hasPrevious()를 호출해서 이동할 수 있는지 확인해야 한다.

표11-14에 있는 메서드 중에서 ‘선택적 기능(optional operation)’이라고 표시된 것들은 반드시 구현하지 않아도 된다. 예를 들어 Iterator인터페이스를 구현하는 클래스에서 remove()는 선택적인 기능이므로 구현하지 않아도 괜찮다. 그렇다하더라도 인터페이스로부터 상속받은 메서드는 추상메서드라 메서드의 몸통(body)을 반드시 만들어 주어야 하므로 다음과 같이 처리한다.

```

public void remove() {
    throw new UnsupportedOperationException();
}

```

단순히 public void remove() {};와 같이 구현하는 것보다 이처럼 예외를 던져서 구현되지 않은 기능이라는 것을 메서드를 호출하는 쪽에 알리는 것이 좋다. 그렇지 않으면 호출하는 쪽에서는 소스를 구해보기 전까지는 이 기능이 바르게 동작하지 않는 이유를 알 방법이 없다.

```

remove
void remove()
Removes from the list the last element that was returned by next() or previous()
(optional operation). This call can only be made once per call to next or previous. It
can be made only if add(E) has not been called after the last call to next or
previous.

Specified by:
remove in interface Iterator<E>

Throws:
UnsupportedOperationException - if the remove operation is not supported
by this list iterator
IllegalStateException - if neither next nor previous have been called, or
remove or add have been called after the last call to next or previous

```

▲ 그림 11-18 Java API 문서에서 찾은 ListIterator의 remove()

Java API 문서에서 remove() 메서드의 상세 내용을 보면 remove 메서드를 지원하지 않는 Iterator는 UnsupportedOperationException을 발생시킨다고 쓰여 있다. 즉, remove 메서드를 구현하지 않는 경우에는 UnsupportedOperationException을 발생시키도록 구현하라는 뜻이다.

위의 코드에서 remove 메서드의 선언부에 예외 처리를 하지 않은 이유는 UnsupportedOperationException이 RuntimeException의 자손이기 때문이다.

Iterator의 remove()는 단독으로 쓰일 수 없고, next()와 같이 써야 한다. 특정 위치의 요소를 삭제하는 것이 아니라 읽어온 것을 삭제한다. next()의 호출 없이 remove()를 호출하면, IllegalStateException이 발생한다.

‘마이크로소프트 아웃룩(Microsoft outlook)’과 같은 email 클라이언트에서 메일 서버에 있는 메일을 가져올 때 서버에 있는 메일을 읽어만 올 것인지(copy), 메일을 가져오면서 서버에서 삭제할 것(move)인지를 선택할 수 있다. 이와 같은 기능을 구현하고자 할 때 쓸 목적으로 remove()를 정의해 놓은 것이다.

단순히 서버에서 읽어오기만 할 때는 next()를 사용하면 되고, 읽어온 메일을 서버에 남기지 않고 지울 때는 next()와 함께 remove()를 사용하면 이와 같은 기능을 구현할 수 있다.

▼ 예제 11-15/ch11/IteratorEx2.java

```

import java.util.*;

public class IteratorEx2 {
    public static void main(String[] args) {
        ArrayList original = new ArrayList(10);
        ArrayList copy1 = new ArrayList(10);
        ArrayList copy2 = new ArrayList(10);

        for(int i=0; i < 10; i++)
            original.add(i+"");
        Iterator it = original.iterator();
    }
}

```

```

        while(it.hasNext())
            copy1.add(it.next());
        System.out.println("= Original에서 copy1로 복사(copy) =");
        System.out.println("original:"+original);
        System.out.println("copy1:"+copy1);
        System.out.println();
        it = original.iterator(); // Iterator는 재사용이 안되므로, 다시 얻어와야 한다.

        while(it.hasNext()){
            copy2.add(it.next());
            it.remove();
        }
    } // main
} // class

```

▼ 실행결과

```

= Original에서 copy1로 복사(copy) =
original:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
copy1:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

= Original에서 copy2로 이동(move) =
original: []
copy2:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

다음 예제는 전에 만들었던 예제11-4의 MyVector클래스를 상속받는 새로운 클래스가 Iterator를 구현하도록 한 것이다. 이 예제를 통해서 Iterator를 어떻게 구현하는지 배울 수 있을 것이다.

▼ 예제 11-16/ch11/MyVector2.java

```

import java.util.*;

public class MyVector2 extends MyVector implements Iterator {
    int cursor = 0;
    int lastRet = -1;

    public MyVector2(int capacity) {
        super(capacity);
    }

    public MyVector2() {
        this(10);
    }

    public String toString() {
        String tmp = "";
        Iterator it = iterator();
        for(int i=0; it.hasNext(); i++) {
            if(i!=0) tmp+=", ";
            tmp += it.next();           // tmp += next().toString();
        }
    }
}

```

```

        return "[" + tmp + "]";
    }

    public Iterator iterator() {
        cursor = 0;      // cursor와 lastRet를 초기화 한다.
        lastRet = -1;
        return this;
    }

    public boolean hasNext() {
        return cursor != size();
    }

    public Object next() {
        Object next = get(cursor);
        lastRet = cursor++;
        return next;
    }

    public void remove() {
        // 더이상 삭제할 것이 없으면 IllegalStateException을 발생시킨다.
        if(lastRet == -1) {
            throw new IllegalStateException();
        } else {
            remove(lastRet);
            cursor--;           // 삭제 후에 cursor의 위치를 감소시킨다.
            lastRet = -1;        // lastRet의 값을 초기화 한다.
        }
    }
} // class

```

| 참고 | 이 예제를 컴파일 하기 위해서는 MyVector.java가 있어야 한다.

cursor는 앞으로 읽어 올 요소의 위치를 저장하는데 사용되고, lastRet는 마지막으로 읽어 온 요소의 위치(index)를 저장하는데 사용된다.

그래서 lastRet는 cursor보다 항상 1이 작은 값이 저장되고 remove()를 호출하면 이미 next()를 통해서 읽은 위치의 요소, 즉 lastRet에 저장된 값의 위치에 있는 요소를 삭제하고 lastRet의 값을 -1로 초기화 한다.

만일 next()를 호출하지 않고 remove()를 호출하면 lastRet의 값은 -1이 되어 IllegalStateException이 발생한다. remove()는 next()로 읽어 온 객체를 삭제하는 것이기 때문에 remove()를 호출하기 전에는 반드시 next()가 호출된 상태이어야 한다.

```

public void remove() {
    // 더이상 삭제할 것이 없으면 IllegalStateException을 발생시킨다.
    if(lastRet == -1) {
        throw new IllegalStateException();
    } else {
        remove(lastRet);   // 최근에 읽어온 요소를 삭제한다.
        cursor--;          // cursor의 위치를 1감소시킨다.
        lastRet = -1;       // 읽어온 요소가 삭제되었으므로 초기화 한다.
    }
}

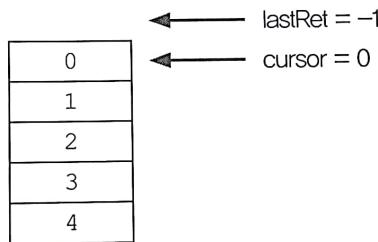
```

위의 코드에서 보면 remove(lastRet)를 호출하여 lastRet의 위치에 있는 객체를 삭제한 다음에 cursor의 값을 감소시킨다. 그리고 lastRet의 값을 초기화(-1)한다.
 그 이유는 remove메서드를 호출해서 객체를 삭제하고 나면, 삭제된 위치 이후의 객체들이 빈 공간을 채우기 위해 자동적으로 이동되기 때문에 cursor의 위치도 같이 이동시켜주어야 한다. 그리고 읽어온 요소가 삭제되었으므로 읽어온 요소의 위치를 저장하는 lastRet의 값은 -1로 초기화해야 한다. lastRet의 값이 -1이라는 것은 읽어온 값이 없다는 것을 의미한다.

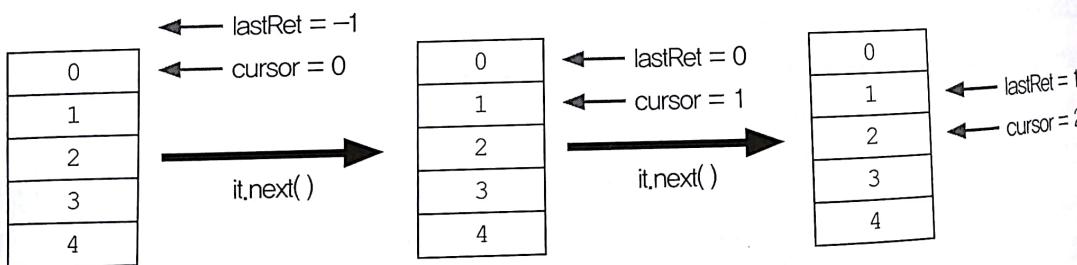
이해를 돋기 위해 그림과 함께 예를 들어 설명하겠다.

```
MyVector2 v = new MyVector2(5);
v.add("0"); v.add("1"); v.add("2"); v.add("3"); v.add("4");
Iterator it = v.iterator();
```

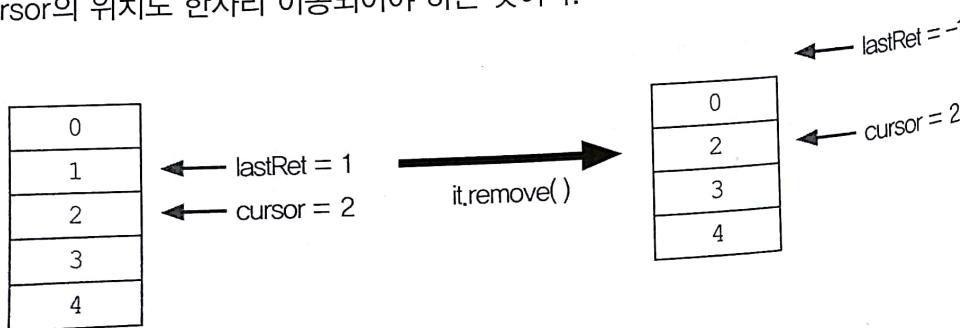
- 만일 위의 코드와 같이 0~4의 값이 저장되어 있는 MyVector2의 Iterator를 얻어왔다면 다음과 같은 그림의 상태일 것이다.



- 그리고 next()를 두 번 호출하면 다음과 같은 그림이 될 것이다. 첫 번째 요소인 0을 읽고 두 번째 요소인 1까지 읽어온 상태이다.



- remove()를 호출하면, 마지막으로 읽어온 요소인 1이 삭제된다. 이 때 lastRet의 값은 -1로 초기화되고 cursor의 값은 1감소된다. 데이터가 삭제되어 다른 데이터들이 한자리씩 이동하였기 때문에 cursor의 위치도 한자리 이동되어야 하는 것이다.



```
참고! lastRet의  
역하자.  
▼ 예제 11-17/  
import java  
class MyVec  
public  
MyVec  
v.a  
v.a  
v.a  
v.a  
v.a  
Sys  
Iter  
it  
it  
it  
it  
it  
it  
}  
}  
참고! 이  
MyVect  
저장한
```

예제11-
.java으
제되지

| 참고 | lastRet의 값이 -1이라는 것은 읽어 온 값이 없다는 의미이다. remove()는 읽어온 값이 있어야 호출될 수 있음을 기억하자.

▼ 예제 11-17/ch11/MyVector2Test.java

```
import java.util.*;
class MyVector2Test {
    public static void main(String args[]) {
        MyVector2 v = new MyVector2();
        v.add("0");
        v.add("1");
        v.add("2");
        v.add("3");
        v.add("4");

        System.out.println("삭제 전 : " + v);
        Iterator it = v.iterator();
        it.next();
        it.remove();
        it.next();
        it.remove();

        System.out.println("삭제 후 : " + v);
    }
}
```

▼ 실행결과

삭제 전 : [0, 1, 2, 3, 4]
삭제 후 : [2, 3, 4]

| 참고 | 이 예제를 실행하려면 MyVector2.java가 필요하다.

MyVector2클래스를 테스트하는 간단한 예제이다. MyVector2객체를 생성하고 데이터를 저장한 다음 Iterator를 통해서 첫 번째와 두 번째에 저장된 데이터를 삭제한다.

```
if(lastRet == -1) {
    throw new IllegalStateException();
} else {
    remove(lastRet); // 주석처리한다.
    cursor--; // 주석처리한다.
    lastRet = -1;
}
```

예제11-16 MyVector2.java의 remove()에서 위와 같이 주석처리하면, MyVector2Test.java의 실행결과는 다음과 같이 될 것이다. 0과 1, 첫 번째와 두 번째 요소가 순서대로 삭제되지 않고, 첫 번째와 세 번째 요소인 0과 2가 삭제된 것을 알 수 있다.

삭제 전 : [0, 1, 2, 3, 4]

삭제 후 : [1, 3, 4]