

컬렉션과 제너릭

학습 목표

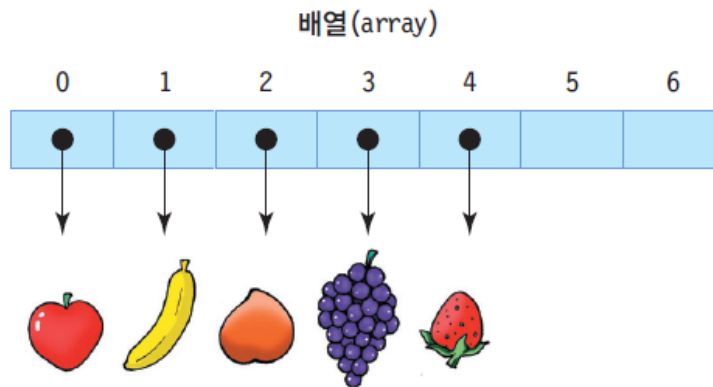
1. 컬렉션과 제네릭 개념
2. Vector<E> 활용
3. ArrayList<E> 활용
4. HashMap<K,V> 활용
5. Iterator<E> 활용
6. 사용자 제네릭 클래스 만들기

컬렉션(collection)의 개념

3

□ 컬렉션

- 요소(element)라고 불리는 가변 개수의 객체들의 저장소
 - 객체들의 컨테이너라고도 불림
 - 요소의 개수에 따라 크기 자동 조절
 - 요소의 삽입, 삭제에 따른 요소의 위치 자동 이동
- 고정 크기의 배열을 다루는 어려움 해소
- 다양한 객체들의 삽입, 삭제, 검색 등의 관리 용이



- 고정 크기 이상의 객체를 관리할 수 없다.
- 배열의 중간에 객체가 삭제되면 응용프로그램에서 자리를 옮겨야 한다.

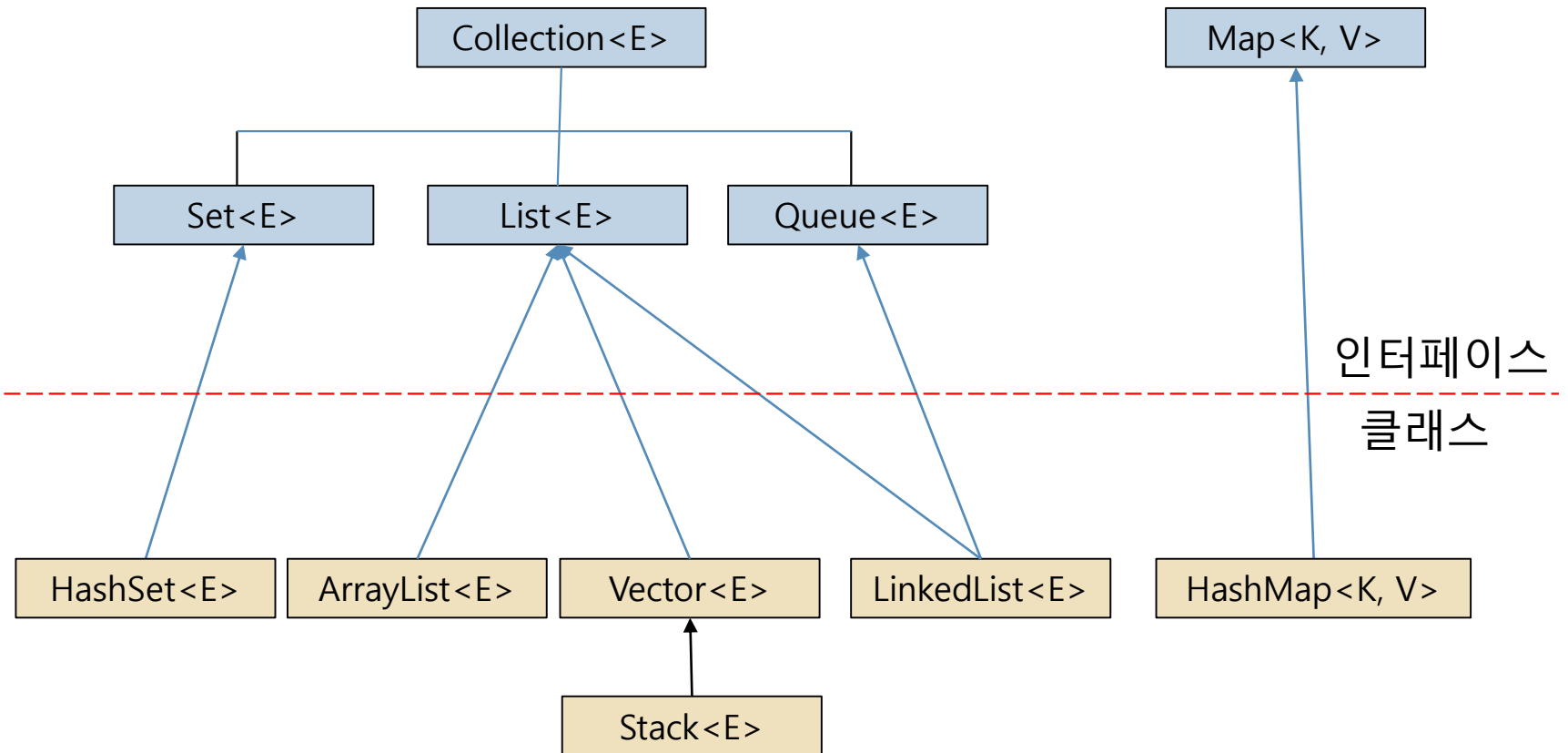
컬렉션(collection)



- 가변 크기로서 객체의 개수를 염려할 필요 없다.
- 컬렉션 내의 한 객체가 삭제되면 컬렉션이 자동으로 자리를 옮겨준다.

컬렉션 자바 인터페이스와 클래스

4



컬렉션의 특징

5

1. 컬렉션은 제네릭(generics) 기법으로 구현

▣ 제네릭

- 특정 타입만 다루지 않고, 여러 종류의 타입으로 변신할 수 있도록 클래스나 메소드를 일반화시키는 기법
- 클래스나 인터페이스 이름에 <E>, <K>, <V> 등 타입매개변수 포함

▣ 제네릭 컬렉션 사례 : 벡터 Vector<E>

- <E>에서 E에 구체적인 타입을 주어 구체적인 타입만 다루는 벡터로 활용
- 정수만 다루는 컬렉션 벡터 Vector<Integer>
- 문자열만 다루는 컬렉션 벡터 Vector<String>

2. 컬렉션의 요소는 객체만 가능

- ▣ int, char, double 등의 기본 타입으로 구체화 불가
- ▣ 컬렉션 사례



```
Vector<int> v = new Vector<int>(); // 컴파일 오류. int는 사용 불가  
Vector<Integer> v = new Vector<Integer>(); // 정상 코드
```

제네릭은 형판과 같은 개념

6

□ 제네릭

- ▣ 클래스나 메소드를 형판에서 찍어내듯이 생산할 수 있도록 일반화된 형판을 만드는 기법

금을 넣으면 금 사과,
은을 넣으면 은 사과가
만들어져요.



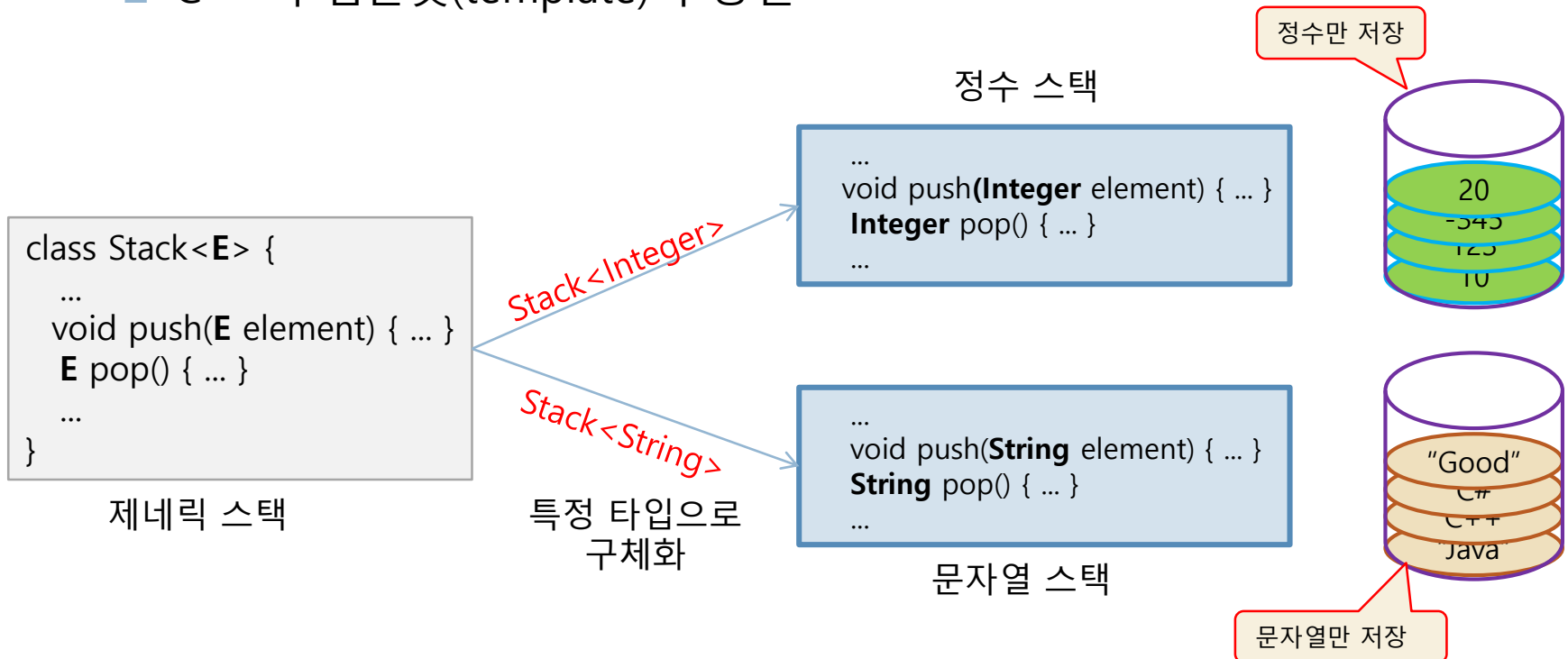
제네릭은 찍어내듯이
코드를 생산할 수 있는
형판입니다. 클래스나
메소드 모두 제네릭으로
만들어 찍어낼 수 있어요.



제네릭의 기본 개념

7

- 제네릭 프로그래밍(generic programming)
 - ▣ JDK 1.5부터 도입(2004년 기점)
 - ▣ 다양한 종류의 데이터 타입을 다룰 수 있도록 일반화된 타입 매개 변수로 클래스(인터페이스)나 메소드를 작성하는 기법
 - ▣ C++의 템플릿(template)과 동일



제네릭 타입

8

□ 제네릭 타입이란?

- ▣ 타입을 파라미터로 가지는 클래스와 인터페이스
- ▣ 선언 시 클래스 또는 인터페이스 이름 뒤에 "<>" 부호 붙임
- ▣ "<>" 사이에는 타입 파라미터 위치

▣ 타입 파라미터

- 일반적으로 대문자 알파벳 한 문자로 표현
- 개발 코드에서는 타입 파라미터 자리에 구체적인 타입을 지정해야

제네릭 타입

9

□ 제네릭 타입 사용 여부에 따른 비교

▣ 제네릭 타입을 사용하지 않은 경우

- Object 타입 사용 → 빈번한 타입 변환 발생 → 프로그램 성능 저하

```
public class Box {  
    private Object object;  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

```
Box box = new Box();  
box.set("hello");           //String 타입을 Object 타입으로 자동 타입 변환해서 저장  
String str = (String) box.get(); //Object 타입을 String 타입으로 강제 타입 변환해서 얻음
```

제네릭 타입

10

□ 제네릭 타입 사용 여부에 따른 비교

▣ 제네릭 타입 사용한 경우

- 클래스 선언할 때 타입 파라미터 사용
- 컴파일 시 타입 파라미터가 구체적인 클래스로 변경

```
public class Box<T> {  
    private T t;  
    public T get() { return t; }  
    public void set(T t) { this.t = t; }  
}
```

```
Box<String> box = new Box<String>();
```

```
public class Box<String> {  
    private String t;  
    public void set(String t) { this.t = t; }  
    public String get() { return t; }  
}
```

```
Box<String> box = new Box<String>();  
box.set("hello");  
String str = box.get();
```

```
Box<Integer> box = new Box<Integer>();
```

```
public class Box<Integer> {  
    private Integer t;  
    public void set(Integer t) { this.t = t; }  
    public Integer get() { return t; }  
}
```

```
Box<Integer> box = new Box<Integer>();  
box.set(6);  
int value = box.get();
```

제네릭 타입 – 멀티 타입 파라미터

11

- 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능
 - ▣ 각 타입 파라미터는 콤마로 구분
 - Ex) class<K, V, ...> { ... }
 - interface<K, V, ...> { ... }

```
public class Product<T, M> {  
    private T kind;  
    private M model;  
  
    public T getKind() { return this.kind; }  
    public M getModel() { return this.model; }  
  
    public void setKind(T kind) { this.kind = kind; }  
    public void setModel(M model) { this.model = model; }  
}
```

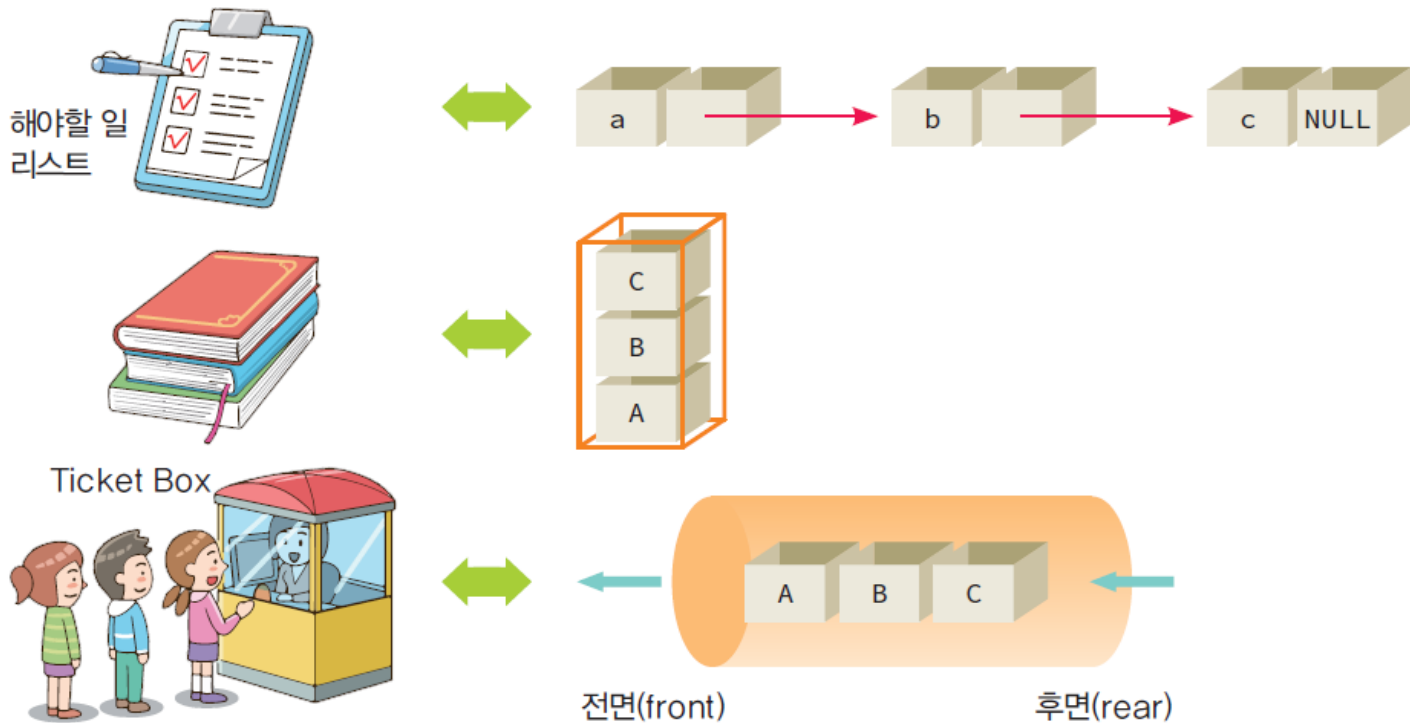
```
Product<Tv, String> product = new Product<Tv, String>();
```

```
Product<Tv, String> product = new Product<>();
```

컬렉션

12

- 컬렉션은 자바에서 자료 구조를 구현한 클래스
 - 자료 구조 : 리스트, 스택, 큐, 집합, 해쉬 테이블 등



Vector<E> 클래스

13

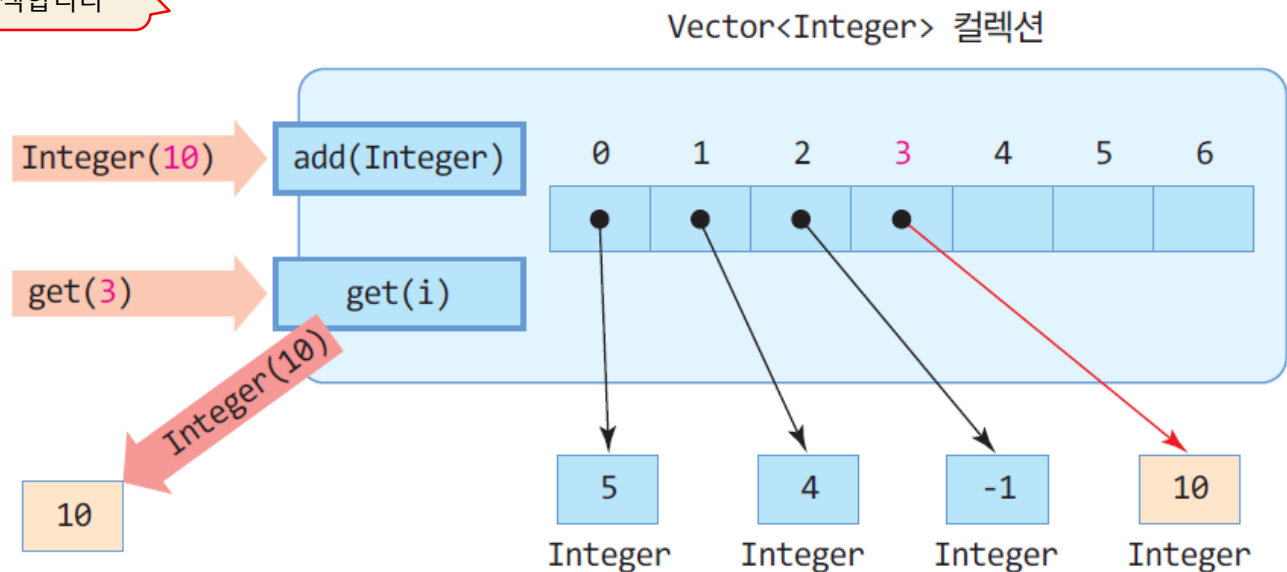
- Java.util 패키지에 있는 컬렉션의 일종으로 가변 크기의 배열을 구현하고 있음
- 벡터 Vector<E>의 특성
 - ▣ <E>에 사용할 요소의 특정 타입으로 구체화
 - ▣ 배열을 가변 크기로 다룰 수 있게 하는 컨테이너
 - 배열의 길이 제한 극복
 - 요소의 개수가 넘치면 자동으로 길이 조절
 - ▣ 요소 객체들을 삽입, 삭제, 검색하는 컨테이너
 - 삽입, 삭제에 따라 자동으로 요소의 위치 조정
 - ▣ Vector에 삽입 가능한 것
 - 객체, null
 - 기본 타입의 값은 Wrapper 객체로 만들어 저장
 - ▣ Vector에 객체 삽입
 - 벡터의 맨 뒤, 중간에 객체 삽입 가능
 - ▣ Vector에서 객체 삭제
 - 임의의 위치에 있는 객체 삭제 가능

Vector<Integer> 벡터 컬렉션 내부

14

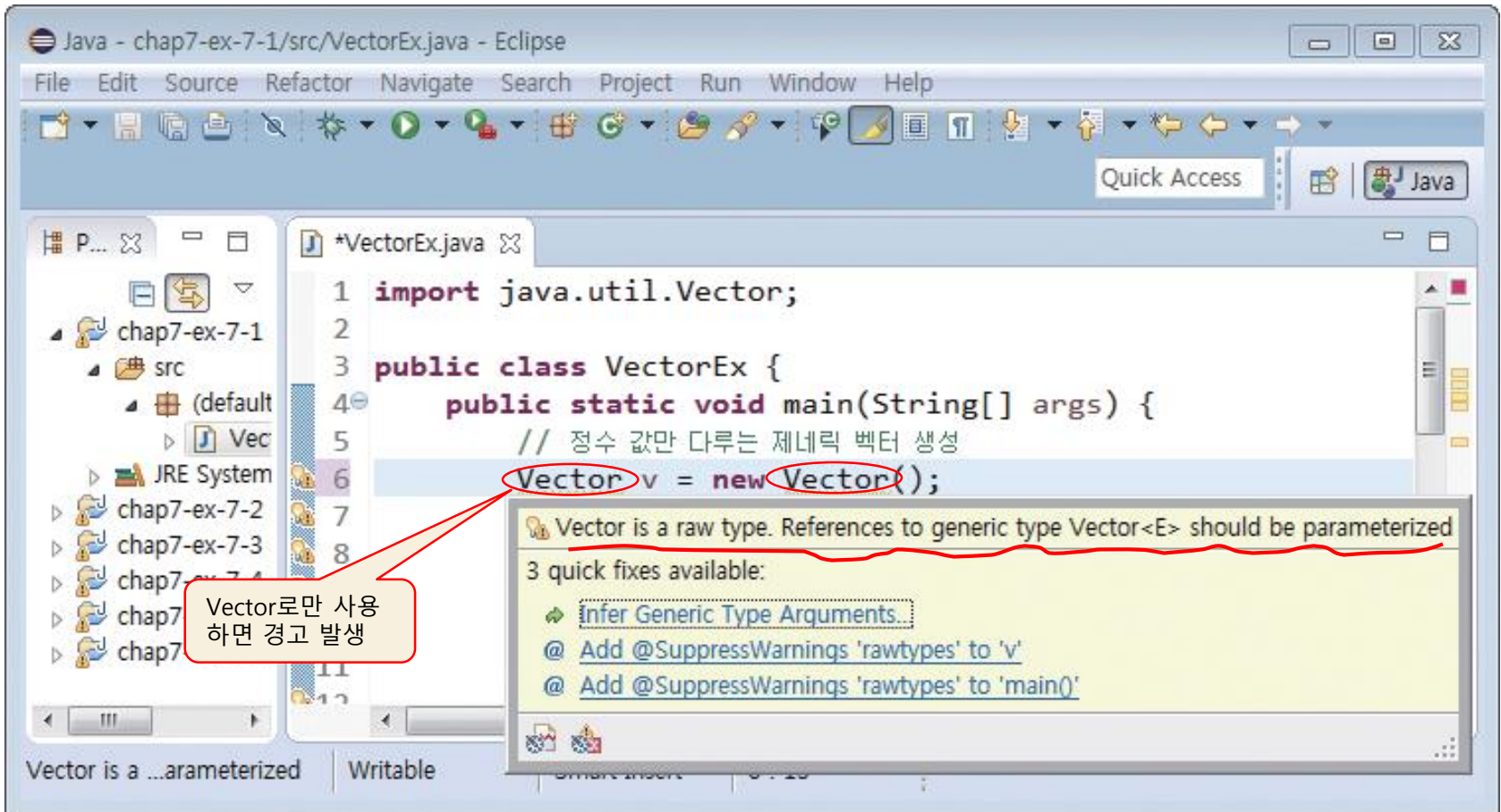
```
Vector<Integer> v = new Vector<Integer>();
```

add()를 이용하여 요소를 삽입하고
get()을 이용하여 요소를 검색합니다



타입 매개 변수 사용하지 않는 경우 경고 발생

15



Vector<Integer>나 Vector<String> 등 타입 매개 변수를 사용하여야 함

Vector<E> 클래스의 주요 메소드

16

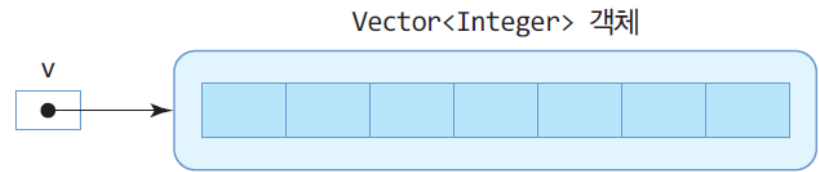
메소드	설명
<code>boolean add(E element)</code>	벡터의 맨 뒤에 element 추가
<code>void add(int index, E element)</code>	인덱스 index에 element를 삽입
<code>int capacity()</code>	벡터의 현재 용량 리턴
<code>boolean addAll(Collection<? extends E> c)</code>	컬렉션 c의 모든 요소를 벡터의 맨 뒤에 추가
<code>void clear()</code>	벡터의 모든 요소 삭제
<code>boolean contains(Object o)</code>	벡터가 지정된 객체 o를 포함하고 있으면 true 리턴
<code>E elementAt(int index)</code>	인덱스 index의 요소 리턴
<code>E get(int index)</code>	인덱스 index의 요소 리턴
<code>int indexOf(Object o)</code>	o와 같은 첫 번째 요소의 인덱스 리턴. 없으면 -1 리턴
<code>boolean isEmpty()</code>	벡터가 비어 있으면 true 리턴
<code>E remove(int index)</code>	인덱스 index의 요소 삭제
<code>boolean remove(Object o)</code>	객체 o와 같은 첫 번째 요소를 벡터에서 삭제
<code>void removeAllElements()</code>	벡터의 모든 요소를 삭제하고 크기를 0으로 만들
<code>int size()</code>	벡터가 포함하는 요소의 개수 리턴
<code>Object[] toArray()</code>	벡터의 모든 요소를 포함하는 배열 리턴

Vector<Integer> 컬렉션 활용 사례

용량 7인 벡터

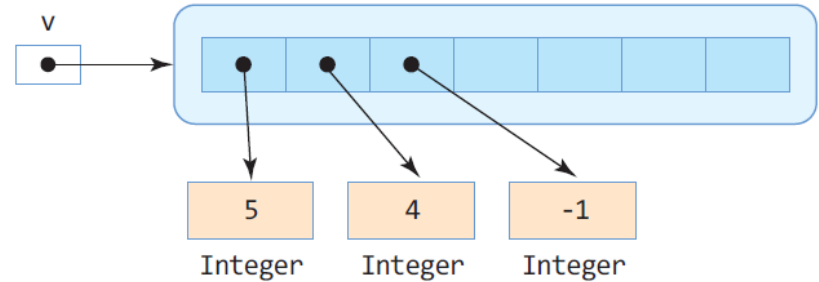
벡터 생성

```
Vector<Integer> v = new Vector<Integer>(7);
```



요소 삽입

```
v.add(5);  
v.add(new Integer(4));  
v.add(-1);
```



요소 개수 n
벡터의 용량 c

```
int n = v.size(); // n은 3  
int c = v.capacity(); // c는 7
```

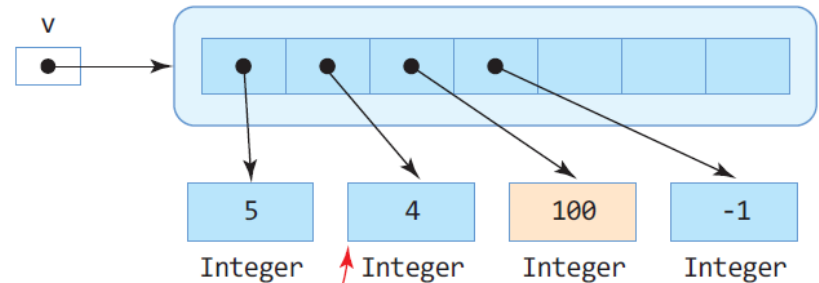
n = 3
c = 7

요소 중간 삽입

```
v.add(2, 100);
```

오류

```
v.add(5, 100);  
// v.size()보다 큰 곳에 삽입 불가능, 오류
```



요소 얻어내기

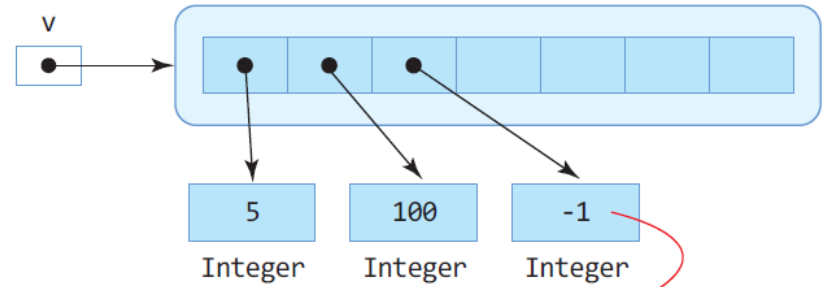
```
Integer obj = v.get(1);  
int i = obj.intValue();
```

obj
i = 4

Vector<Integer> 컬렉션 활용 사례(계속)

요소 삭제 `v.remove(1);`

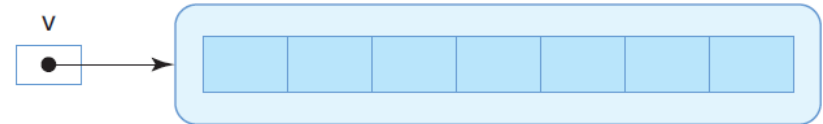
오류 `v.remove(4);`
// 인덱스 4에 요소 객체가 없으므로 오류



마지막 요소 `int last = v.lastElement();`

`last = -1`

모든 요소 삭제 `v.removeAllElements();`



컬렉션과 자동 박싱/언박싱

19

□ JDK 1.5 이전

- 기본 타입 데이터를 Wrapper 객체로 만들어 삽입

```
Vector<Integer> v = new Vector<Integer>();  
v.add(new Integer(4));
```

- 컬렉션으로부터 요소를 얻어올 때, Wrapper 클래스로 캐스팅 필요

```
Integer n = (Integer)v.get(0);  
int k = n.intValue(); // k = 4
```

□ JDK 1.5부터

- 자동 박싱/언박싱이 작동하여 기본 타입 값 삽입 가능

```
Vector<Integer> v = new Vector<Integer> ();  
v.add(4); // 4 → new Integer(4)로 자동 박싱  
int k = v.get(0); // Integer 타입이 int 타입으로 자동 언박싱, k = 4
```

- 그러나, 타입 매개 변수를 기본 타입으로 구체화할 수는 없음



```
Vector<int> v = new Vector<int> (); // 컴파일 오류
```

예제 7-1 : 정수만 다루는 Vector<Integer> 컬렉션 활용

20

정수만 다루는 Vector<Integer> 제네릭 벡터를 생성하고 활용하는 사례를 보인다.
다음 코드에 대한 결과는 무엇인가?

```
import java.util.Vector;

public class VectorEx {
    public static void main(String[] args) {
        // 정수 값만 다루는 제네릭 벡터 생성
        Vector<Integer> v = new Vector<Integer>();
        v.add(5); // 5 삽입
        v.add(4); // 4 삽입
        v.add(-1); // -1 삽입

        // 벡터 중간에 삽입하기
        v.add(2, 100); // 4와 -1 사이에 정수 100 삽입
        System.out.println("벡터 내의 요소 객체 수 : " + v.size());
        System.out.println("벡터의 현재 용량 : " + v.capacity());

        // 모든 요소 정수 출력하기
        for(int i=0; i<v.size(); i++) {
            int n = v.get(i); // 벡터의 i 번째 정수
            System.out.println(n);
        }
    }
}
```

```
// 벡터 속의 모든 정수 더하기
int sum = 0;
for(int i=0; i<v.size(); i++) {
    int n = v.elementAt(i); // 벡터의 i 번째 정수
    sum += n;
}
System.out.println("벡터에 있는 정수 합 : " + sum);
}
```

벡터 내의 요소 객체 수 : 4
벡터의 현재 용량 : 10
5
4
100
-1
벡터에 있는 정수 합 : 108

예제 7-2 : Point 클래스의 객체들만 저장하는 벡터 만들기

21

점 (x, y)를 표현하는 Point 클래스의 객체만 다루는 벡터의 활용을 보여라.

```
import java.util.Vector;
```

```
class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public String toString() {  
        return "(" + x + "," + y + " ";  
    }  
}
```

```
public class PointVectorEx {  
    public static void main(String[] args) {  
        Vector<Point> v = new Vector<Point>();  
  
        // 3 개의 Point 객체 삽입  
        v.add(new Point(2, 3));  
        v.add(new Point(-5, 20));  
        v.add(new Point(30, -8));  
  
        v.remove(1); // 인덱스 1의 Point(-5, 20) 객체 삭제  
  
        // 벡터에 있는 Point 객체 모두 검색하여 출력  
        for(int i=0; i<v.size(); i++) {  
            Point p = v.get(i); // 벡터의 i 번째 Point 객체 얻어내기  
            System.out.println(p); // p.toString()을 이용하여 객체 p 출력  
        }  
    }  
}
```

(2,3)
(30,-8)

컬렉션 인터페이스와 컬렉션 클래스

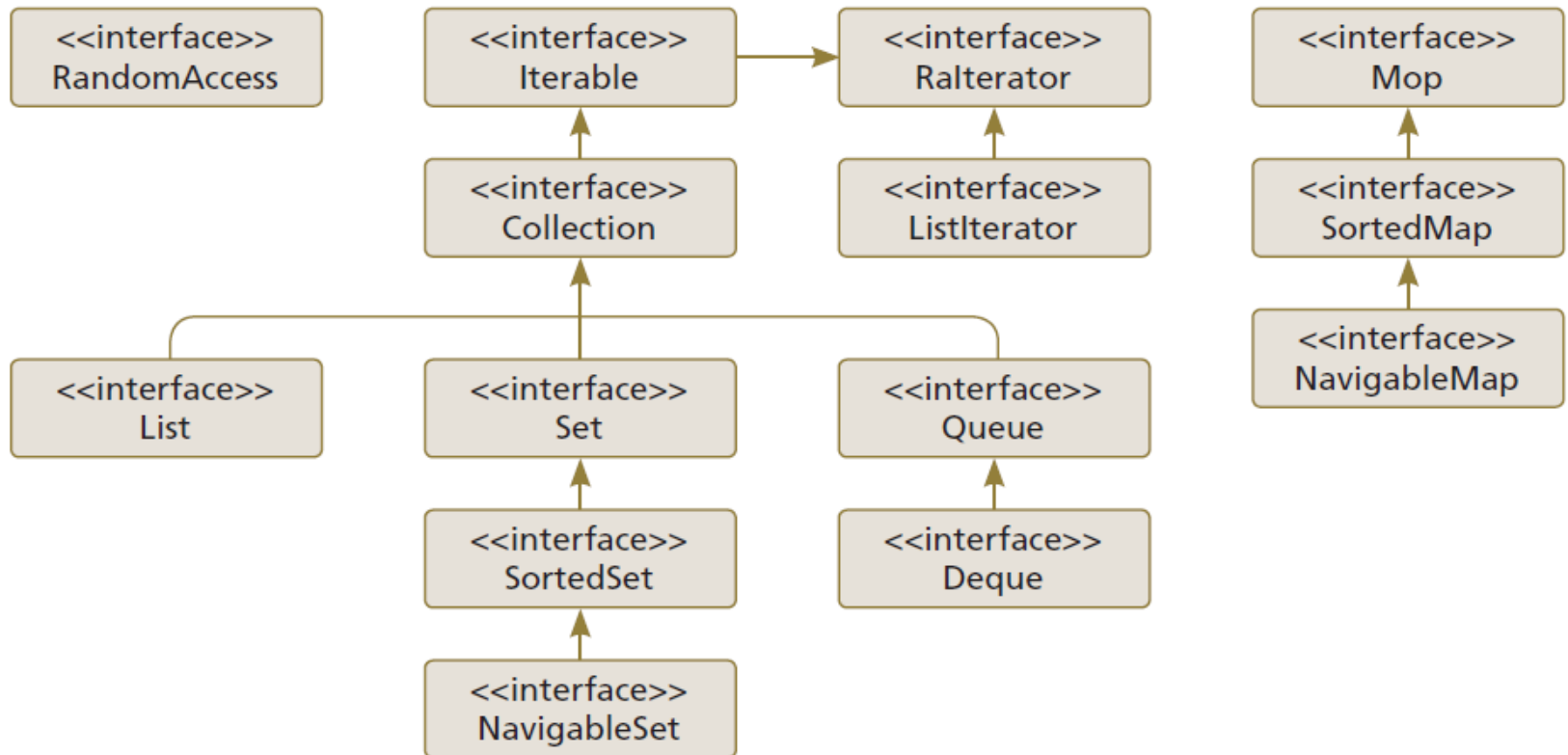
22

- 컬렉션 인터페이스와 컬렉션 클래스로 나누어서 제공
 - ▣ 컬렉션 인터페이스를 구현한 클래스도 함께 제공하므로 이것을 간단하게 사용할 수도 있음.
 - ▣ 각자 필요에 맞추어 인터페이스를 자신의 클래스로 구현할 수도 있음

인터페이스	설명
Collection	모든 자료 구조의 부모 인터페이스로서 객체의 모임을 나타낸다.
Set	집합(중복된 원소를 가지지 않는)을 나타내는 자료 구조
List	순서가 있는 자료 구조로 중복된 원소를 가질 수 있다.
Map	키와 값들이 연관되어 있는 사전과 같은 자료 구조
Queue	극장에서의 대기줄과 같이 들어온 순서대로 나가는 자료구조

Collection 인터페이스

23



Collection 인터페이스

24

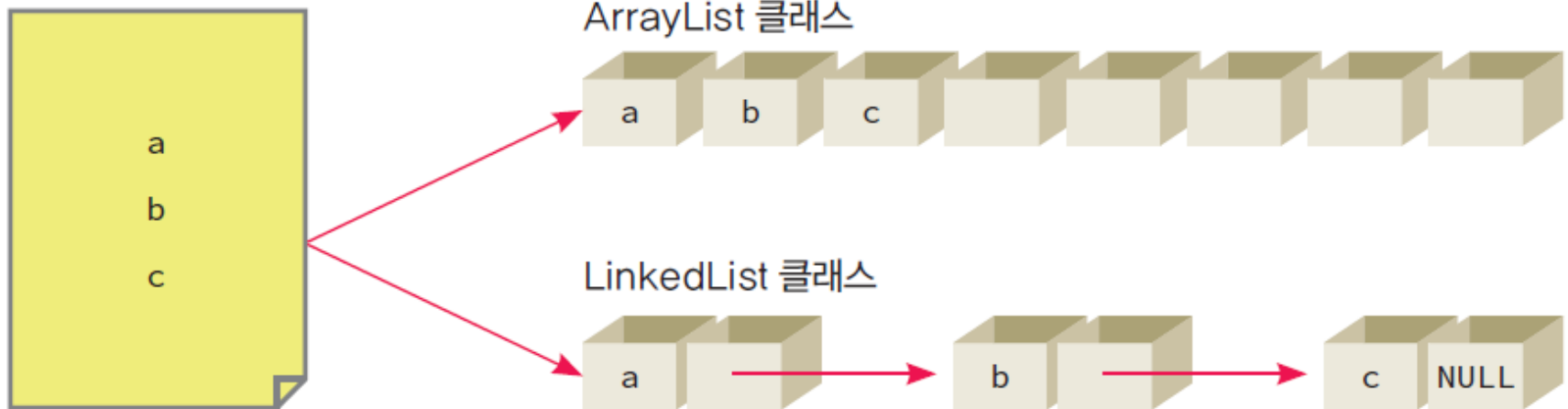
메소드	설명
boolean isEmpty() boolean contains(Object obj) boolean containsAll(Collection<?> c)	공백 상태이면 true 반환 obj를 포함하고 있으면 true 반환
boolean add(E element) boolean addAll(Collection<? extends E> from)	원소를 추가한다.
boolean remove(Object obj) boolean removeAll(Collection<?> c) boolean retainAll(Collection<?> c) void clear()	원소를 삭제한다.
Iterator<E> iterator() Stream<E> stream() Stream<E> parallelStream()	원소 방문
int size()	원소의 개수 반환
Object[] toArray() <T> T[] toArray(T[] a)	컬렉션을 배열로 변환

List 인터페이스

25

- 리스트(List)는 순서를 가지는 요소들의 모임으로 중복된 요소를 가질 수 있음

List 인터페이스

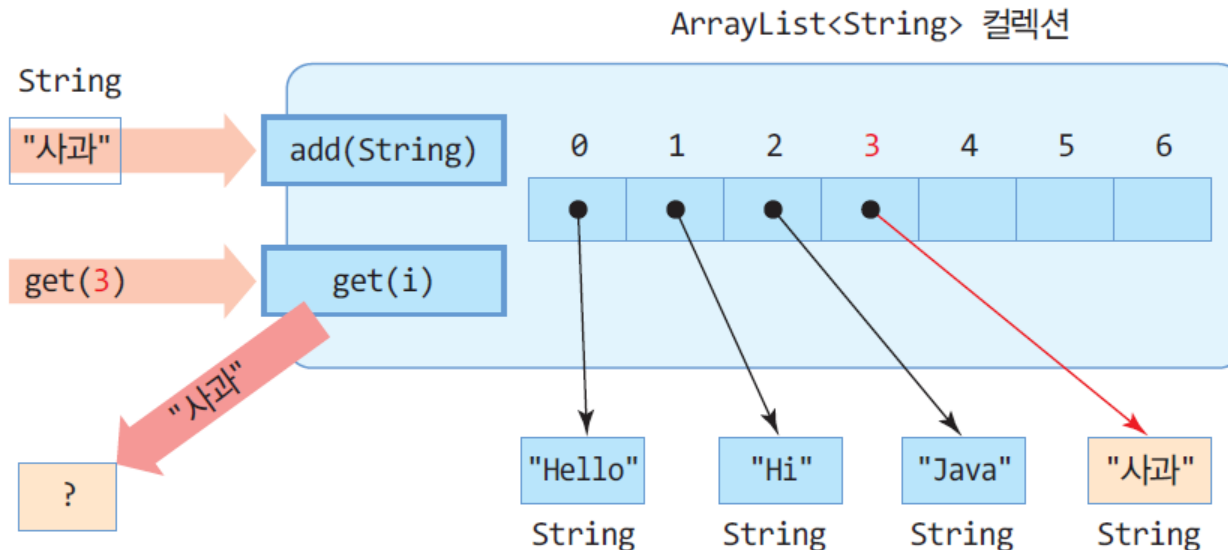


ArrayList<E>

26

- ArrayList를 배열(Array)의 향상된 버전 또는 가변 크기의 배열이라고 생각하면 된다.
- ArrayList의 생성

```
ArrayList<String> list = new ArrayList<String>();
```



ArrayList<E> 클래스의 주요 메소드

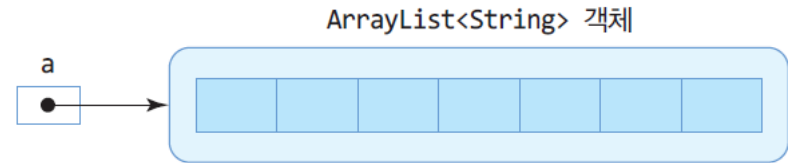
27

메소드	설명
<code>boolean add(E element)</code>	ArrayList의 맨 뒤에 element 추가
<code>void add(int index, E element)</code>	인덱스 index에 지정된 element 삽입
<code>boolean addAll(Collection<? extends E> c)</code>	컬렉션 c의 모든 요소를 ArrayList의 맨 뒤에 추가
<code>void clear()</code>	ArrayList의 모든 요소 삭제
<code>boolean contains(Object o)</code>	ArrayList가 지정된 객체를 포함하고 있으면 true 리턴
<code>E elementAt(int index)</code>	index 인덱스의 요소 리턴
<code>E get(int index)</code>	index 인덱스의 요소 리턴
<code>int indexOf(Object o)</code>	o와 같은 첫 번째 요소의 인덱스 리턴. 없으면 -1 리턴
<code>boolean isEmpty()</code>	ArrayList가 비어 있으면 true 리턴
<code>E remove(int index)</code>	index 인덱스의 요소 삭제
<code>boolean remove(Object o)</code>	o와 같은 첫 번째 요소를 ArrayList에서 삭제
<code>int size()</code>	ArrayList가 포함하는 요소의 개수 리턴
<code>Object[] toArray()</code>	ArrayList의 모든 요소를 포함하는 배열 리턴

ArrayList<String> 컬렉션 활용 사례

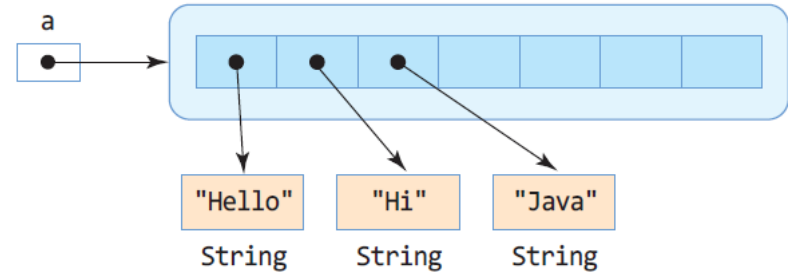
ArrayList 생성

```
ArrayList<String> a = new ArrayList<String>(7);
```



요소 삽입

```
a.add("Hello");  
a.add("Hi");  
a.add("Java");
```



요소 개수 n

```
int n = a.size(); // n은 3
```

벡터의 용량 c

```
int c = a.capacity(); // capacity() 메소드 없음
```

오류

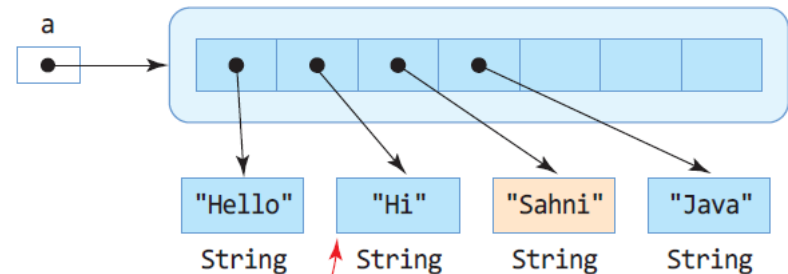
n = 3

요소 중간 삽입

```
a.add(2, "Sahni");
```

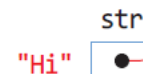
오류

```
a.add(5, "Sahni");  
// a.size()보다 큰 위치에 삽입 불가능, 오류
```




요소 알아내기

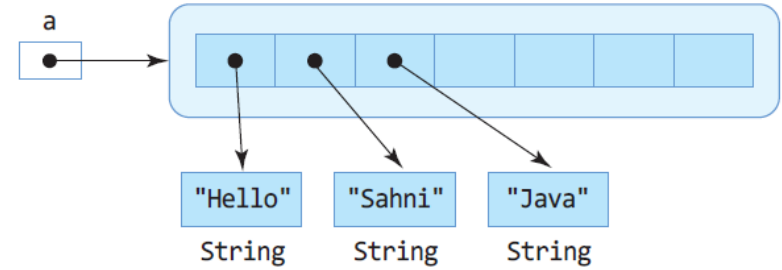
```
String str = a.get(1);
```



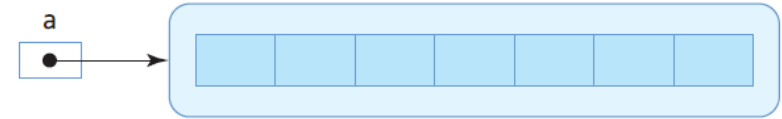
ArrayList<String> 컬렉션 활용 사례(계속)

요소 삭제 `a.remove(1);`

 ~~`a.remove(4);`~~ // 오류



모든 요소 삭제 `a.clear();`



예제 7-3 : 문자열만 다루는 ArrayList<String> 활용

30

이름을 4개 입력받아 ArrayList에 저장하고, ArrayList에 저장된 이름을 모두 출력한 후, 제일 긴 이름을 출력하라.

```
import java.util.*;

public class ArrayListEx {
    public static void main(String[] args) {
        // 문자열만 삽입가능한 ArrayList 컬렉션 생성
        ArrayList<String> a = new ArrayList<String>();

        // 키보드로부터 4개의 이름 입력받아 ArrayList에 삽입
        Scanner scanner = new Scanner(System.in);
        for(int i=0; i<4; i++) {
            System.out.print("이름을 입력하세요>>");
            String s = scanner.next(); // 키보드로부터 이름 입력
            a.add(s); // ArrayList 컬렉션에 삽입
        }

        // ArrayList에 들어 있는 모든 이름 출력
        for(int i=0; i<a.size(); i++) {
            // ArrayList의 i 번째 문자열 얻어오기
            String name = a.get(i);
            System.out.print(name + " ");
        }
    }
}
```

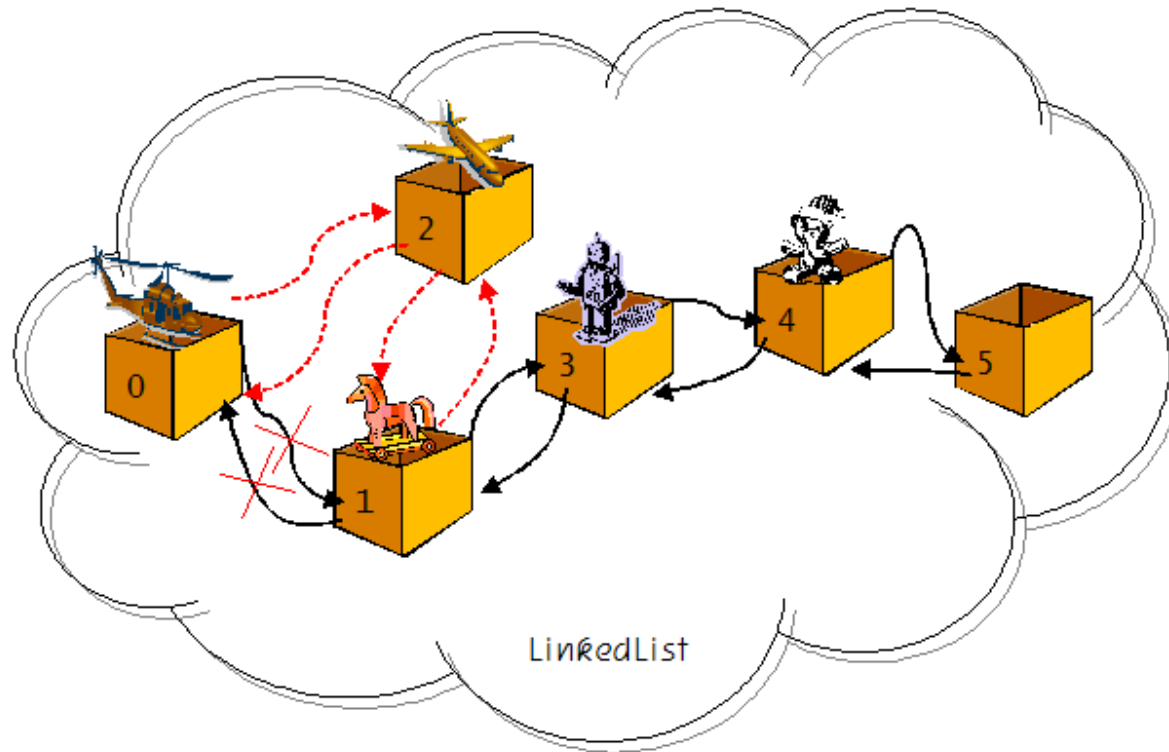
```
// 가장 긴 이름 출력
int longestIndex = 0;
for(int i=1; i<a.size(); i++) {
    if(a.get(longestIndex).length() < a.get(i).length())
        longestIndex = i;
}
System.out.println("\n가장 긴 이름은 : " +
    a.get(longestIndex));
}
```

```
이름을 입력하세요>>Mike
이름을 입력하세요>>Jane
이름을 입력하세요>>Ashley
이름을 입력하세요>>Helen
Mike Jane Ashley Helen
가장 긴 이름은 : Ashley
```

LinkedList

31

- 빈번하게 삽입과 삭제가 일어나는 경우에 사용



LinkedList

32

```
import java.util.*;
public class LinkedListTest {
    public static void main(String args[]) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("MILK");
        list.add("BREAD");
        list.add("BUTTER");
        list.add(1, "APPLE");    // 인덱스 1에 "APPLE"을 삽입
        list.set(2, "GRAPE");    // 인덱스 2의 원소를 "GRAPE"로 대체
        list.remove(3);          // 인덱스 3의 원소를 삭제한다.
        for (int i = 0; i < list.size(); i++)
            System.out.println(list.get(i));
    }
}
```


컬렉션의 순차 검색을 위한 Iterator

33

□ Iterator<E> 인터페이스

▣ 리스트 구조의 컬렉션에서 요소의 순차 검색을 위한 인터페이스

- Vector<E>, ArrayList<E>, LinkedList<E>가 상속받는 인터페이스

□ Iterator 객체 얻어내기

▣ 컬렉션의 iterator() 메소드 호출

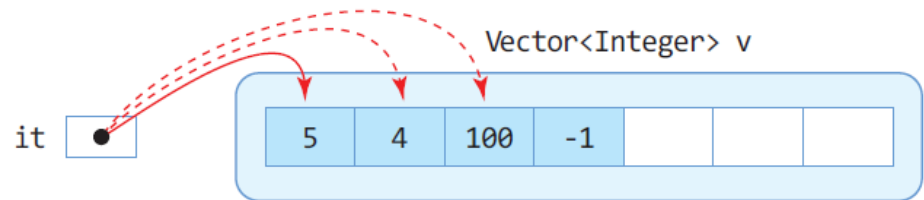
- 해당 컬렉션을 순차 검색할 수 있는
Iterator 객체 리턴

▣ 컬렉션 검색 코드

```
Vector<Integer> v = new Vector<Integer>();  
Iterator<Integer> it = v.iterator();
```

```
while(it.hasNext()) { // 모든 요소 방문  
    int n = it.next(); // 다음 요소 리턴  
    ...  
}
```

메소드	설명
boolean hasNext()	다음 반복에서 사용될 요소가 있으면 true 리턴
E next()	다음 요소 리턴
void remove()	마지막으로 리턴된 요소 제거



Vector<Integer> 객체와 Iterator 객체의 관계

예제 7-4 : Iterator<Integer>를 이용하여 정수 벡터 검색

34

예제 7-1의 코드 중에서 벡터 검색 부분을 Iterator<Integer>를 이용하여 수정하라.

```
import java.util.*;

public class IteratorEx {
    public static void main(String[] args) {
        // 정수 값만 다루는 제네릭 벡터 생성
        Vector<Integer> v = new Vector<Integer>();
        v.add(5); // 5 삽입
        v.add(4); // 4 삽입
        v.add(-1); // -1 삽입
        v.add(2, 100); // 4와 -1 사이에 정수 100 삽입

        // Iterator를 이용한 모든 정수 출력하기
        Iterator<Integer> it = v.iterator(); // Iterator 객체 얻기
        while(it.hasNext()) {
            int n = it.next();
            System.out.println(n);
        }
    }
}
```

```
// Iterator를 이용하여 모든 정수 더하기
int sum = 0;
it = v.iterator(); // Iterator 객체 얻기
while(it.hasNext()) {
    int n = it.next();
    sum += n;
}
System.out.println("벡터에 있는 정수 합 : " + sum);
}
```

```
5
4
100
-1
벡터에 있는 정수 합 : 108
```

Iterator 사용 예

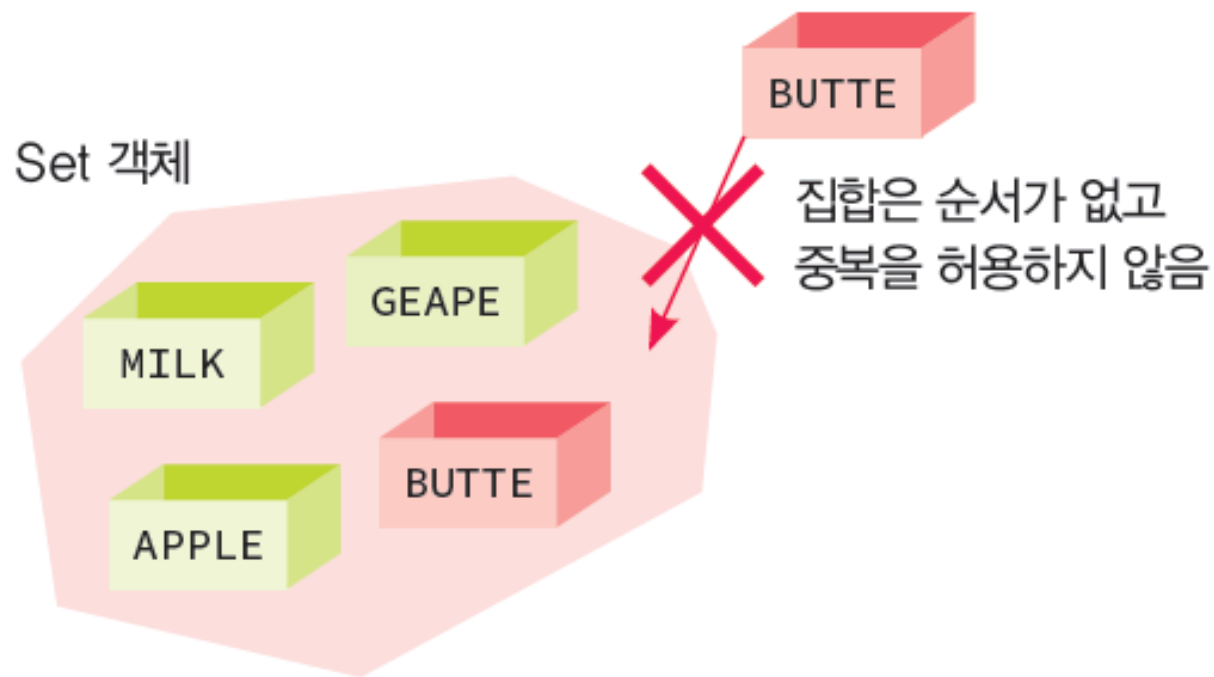
35

```
ArrayList<String> list = new ArrayList<String>();  
list.add("하나 ");  
list.add("둘 ");  
list.add("셋 ");  
list.add("넷 ");  
String s;  
Iterator e = list.iterator();  
while(e.hasNext())  
{  
    s = (String)e.next();           // 반복자는 Object 타입을 반환!  
    System.out.println(s);  
}
```

Set

36

- 집합(Set)은 원소의 중복을 허용하지 않는다.



Set 인터페이스를 구현하는 방법

37

□ HashSet

- HashSet은 해쉬 테이블에 원소를 저장하기 때문에 성능면에서 가장 우수하다. 하지만 원소들의 순서가 일정하지 않은 단점이 있다.

□ TreeSet

- 레드-블랙 트리(red-black tree)에 원소를 저장한다. 따라서 값에 따라서 순서가 결정되며 하지만 HashSet보다는 느리다.

□ LinkedHashSet

- 해쉬 테이블과 연결 리스트를 결합한 것으로 원소들의 순서는 삽입되었던 순서와 같다.

Set 예제

38

```
import java.util.*;
public class SetTest {
    public static void main(String args[]) {
        HashSet<String> set = new HashSet<String>();
        set.add("Milk");
        set.add("Bread");
        set.add("Butter");
        set.add("Cheese");
        set.add("Ham");
        set.add("Ham");
        System.out.println(set);
    }
}
```

Set 예제

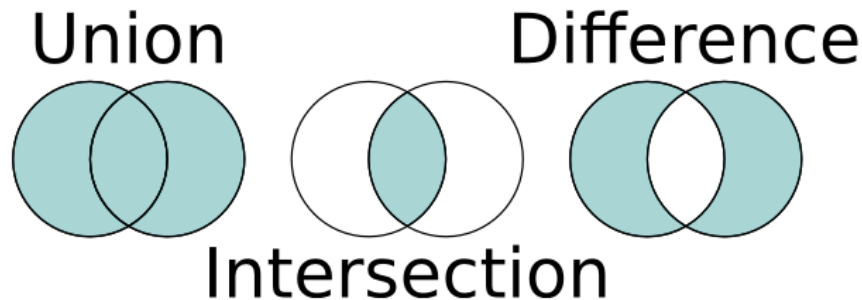
39

```
import java.util.*;
public class FindDupplication {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        String[] sample = { "단어", "중복", "구절", "중복" };
        for (String a : sample)
            if (!s.add(a))
                System.out.println("중복된 단어 " + a);
        System.out.println(s.size() + " 중복되지 않은 단어: " + s);
    }
}
```

대량 연산 메소드

40

- `s1.containsAll(s2)`
 - ▣ 만약 `s2`가 `s1`의 부분 집합이면 참이다.
- `s1.addAll(s2)`
 - ▣ `s1`을 `s1`과 `s2`의 합집합으로 만든다.
- `s1.retainAll(s2)`
 - ▣ `s1`을 `s1`과 `s2`의 교집합으로 만든다.
- `s1.removeAll(s2)`
 - ▣ `s1`을 `s1`과 `s2`의 차집합으로 만든다.



예제

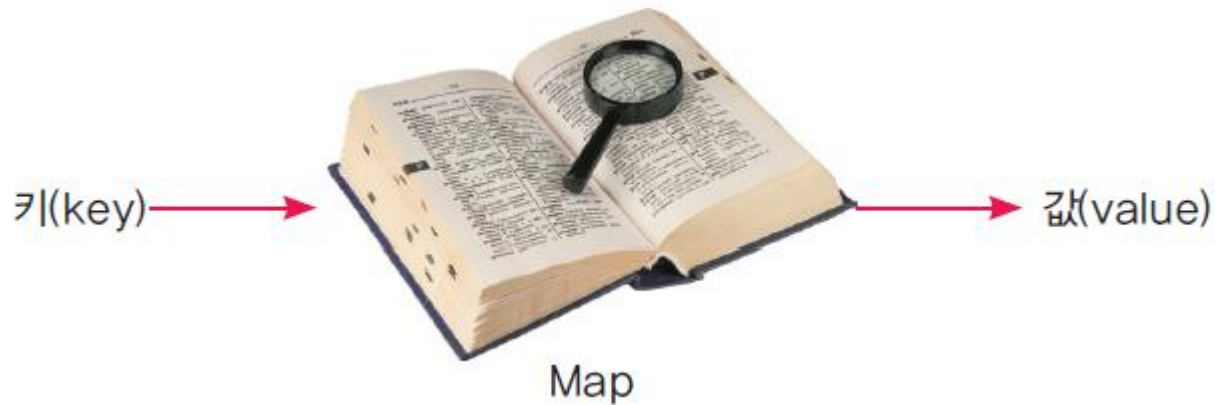
41

```
public class SetTest1 {  
    public static void main(String[] args) {  
        Set<String> s1 = new HashSet<String>();  
        Set<String> s2 = new HashSet<String>();  
        s1.add("A");  
        s1.add("B");  
        s1.add("C");  
        s2.add("A");  
        s2.add("D");  
        Set<String> union = new HashSet<String>(s1);  
        union.addAll(s2);  
        Set<String> intersection = new HashSet<String>(s1);  
        intersection.retainAll(s2);  
        System.out.println("합집합 " + union);  
        System.out.println("교집합 " + intersection);  
    }  
}
```

Map

42

- Map은 많은 데이터 중에서 원하는 데이터를 빠르게 찾을 수 있는 자료 구조이다.
- 맵은 사전과 같은 자료 구조이다.



HashMap<K,V>

43

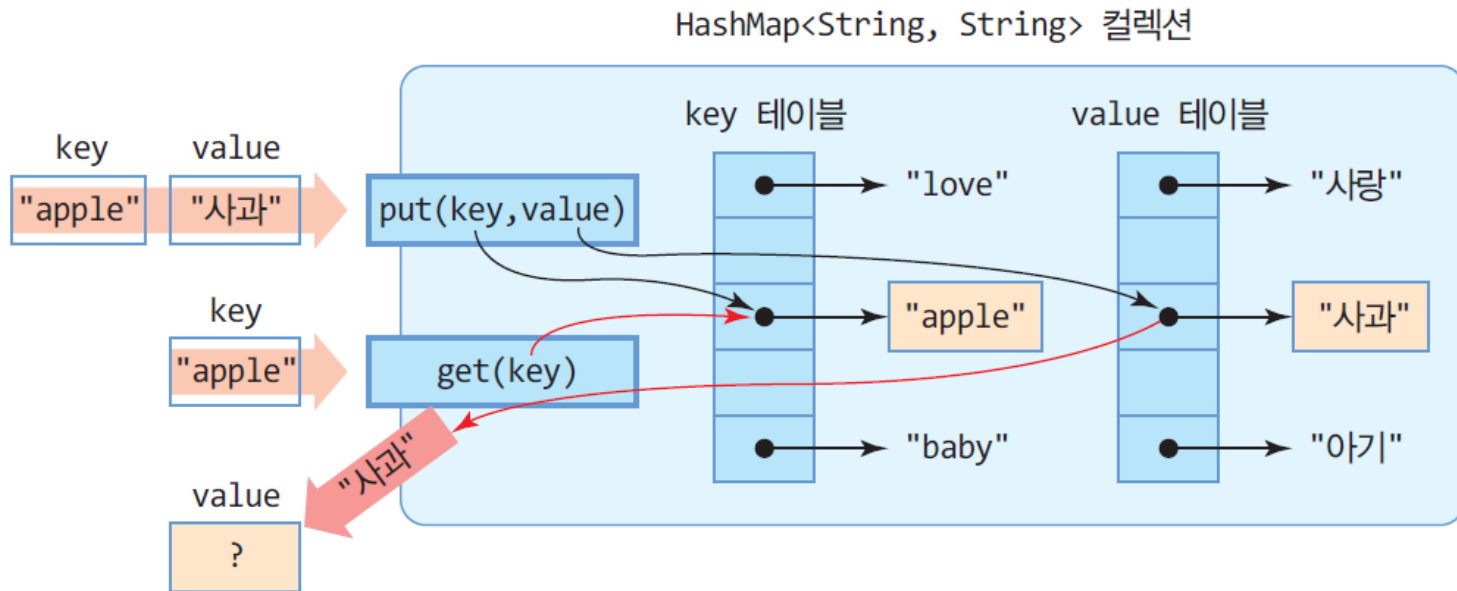
- ▣ 키(key)와 값(value)의 쌍으로 구성되는 요소를 다루는 컬렉션
 - K : 키로 사용할 요소의 타입
 - V : 값으로 사용할 요소의 타입
 - 키와 값이 한 쌍으로 삽입
 - '값' 을 검색하기 위해서는 반드시 '키' 이용
- ▣ 삽입 및 검색이 빠른 특징
 - 요소 삽입 : put() 메소드
 - 요소 검색 : get() 메소드
- ▣ 예) HashMap<String, String> 생성, 요소 삽입, 요소 검색

```
HashMap<String, String> h = new HashMap<String, String>(); // 해시맵 객체 생성  
h.put("apple", "사과"); // "apple" 키와 "사과" 값의 쌍을 해시맵에 삽입  
String kor = h.get("apple"); // "apple" 키로 값 검색. kor는 "사과"
```

HashMap<String, String>의 내부 구성

44

```
HashMap<String, String> map = new HashMap<String, String>();
```



HashMap<K,V>의 주요 메소드

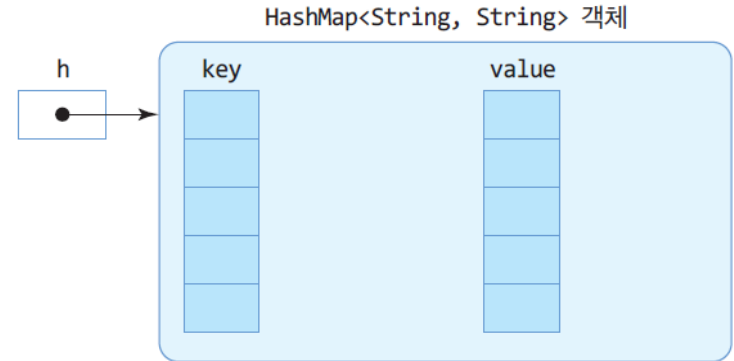
45

메소드	설명
<code>void clear()</code>	HashMap의 모든 요소 삭제
<code>boolean containsKey(Object key)</code>	지정된 키(key)를 포함하고 있으면 true 리턴
<code>boolean containsValue(Object value)</code>	하나 이상의 키를 지정된 값(value)에 매핑시킬 수 있으면 true 리턴
<code>V get(Object key)</code>	지정된 키(key)에 매핑되는 값 리턴. 키에 매핑되는 어떤 값도 없으면 null 리턴
<code>boolean isEmpty()</code>	HashMap이 비어 있으면 true 리턴
<code>Set<K> keySet()</code>	HashMap에 있는 모든 키를 담은 Set<K> 컬렉션 리턴
<code>V put(K key, V value)</code>	key와 value를 매핑하여 HashMap에 저장
<code>V remove(Object key)</code>	지정된 키(key)와 이에 매핑된 값을 HashMap에서 삭제
<code>int size()</code>	HashMap에 포함된 요소의 개수 리턴

HashMap<String, String> 컬렉션 활용 사례

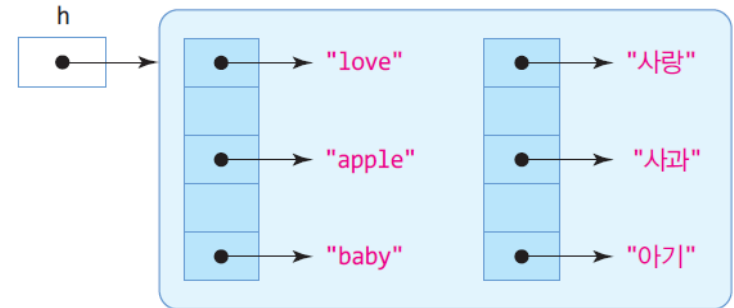
해시맵 생성

```
HashMap<String, String> h =  
new HashMap<String, String>();
```



(키, 값) 삽입

```
h.put("baby", "아기");  
h.put("love", "사랑");  
h.put("apple", "사과");
```



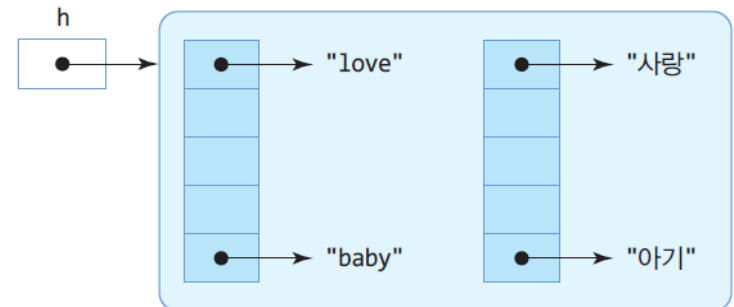
키로 값 읽기

```
String kor = h.get("love");
```

kor = "사랑"

키로 요소 삭제

```
h.remove("apple");
```



요소 개수

```
int n = h.size();
```

n = 2

예제 7-5 : HashMap<String, String>로 (영어, 한글) 단어 쌍을 저장하고 검색하기

47

영어 단어와 한글 단어의 쌍을 HashMap에 저장하고, 영어 단어로 한글 단어를 검색하는 프로그램을 작성하라.

```
import java.util.*;
public class HashMapDicEx {
    public static void main(String[] args) {
        // 영어 단어와 한글 단어의 쌍을 저장하는 HashMap 컬렉션 생성
        HashMap<String, String> dic = new HashMap<String, String>();

        // 3 개의 (key, value) 쌍을 dic에 저장
        dic.put("baby", "아기"); // "baby"는 key, "아기"은 value
        dic.put("love", "사랑");
        dic.put("apple", "사과");

        // dic 해시맵에 들어 있는 모든 (key, value) 쌍 출력
        Set<String> keys = dic.keySet(); // 모든 키를 Set 컬렉션에 받아옴
        Iterator<String> it = keys.iterator(); // Set에 접근하는 Iterator 리턴
        while(it.hasNext()) {
            String key = it.next(); // 키
            String value = dic.get(key); // 값
            System.out.print("(" + key + "," + value + " ");
        }
        System.out.println();
    }
}
```

```
// 영어 단어를 입력받고 한글 단어 검색
Scanner scanner = new Scanner(System.in);
for(int i=0; i<3; i++) {
    System.out.print("찾고 싶은 단어는?");
    String eng = scanner.next();
    // 해시맵에서 '키' eng의 '값' kor 검색
    String kor = dic.get(eng);
    if(kor == null)
        System.out.println(eng +
            "는 없는 단어 입니다.");
    else
        System.out.println(kor);
    }
}
```

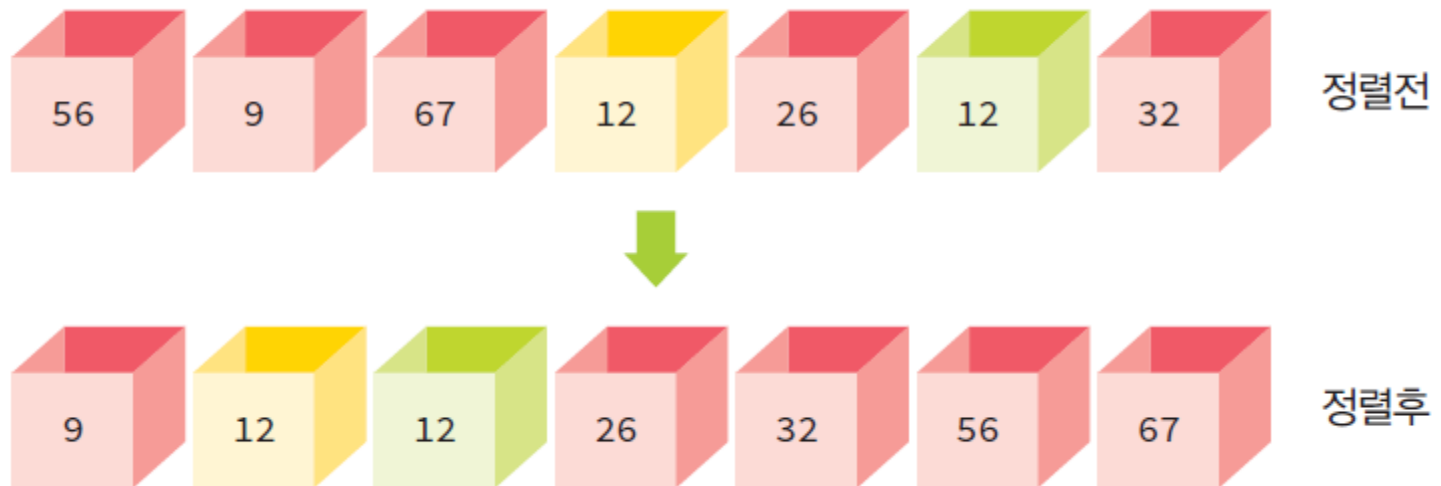
```
(love,사랑)(apple,사과)(baby,아기)
찾고 싶은 단어는?apple
사과
찾고 싶은 단어는?babo
babo는 없는 단어 입니다.
찾고 싶은 단어는?love
사랑
```

Collections 클래스

- Collections 클래스는 여러 유용한 알고리즘을 구현한 메소드들을 제공한다.
- 정렬(Sorting)
- 섞기(Shuffling)
- 탐색(Searching)

정렬

- 정렬은 데이터를 어떤 기준에 의하여 순서대로 나열하는 것이다.



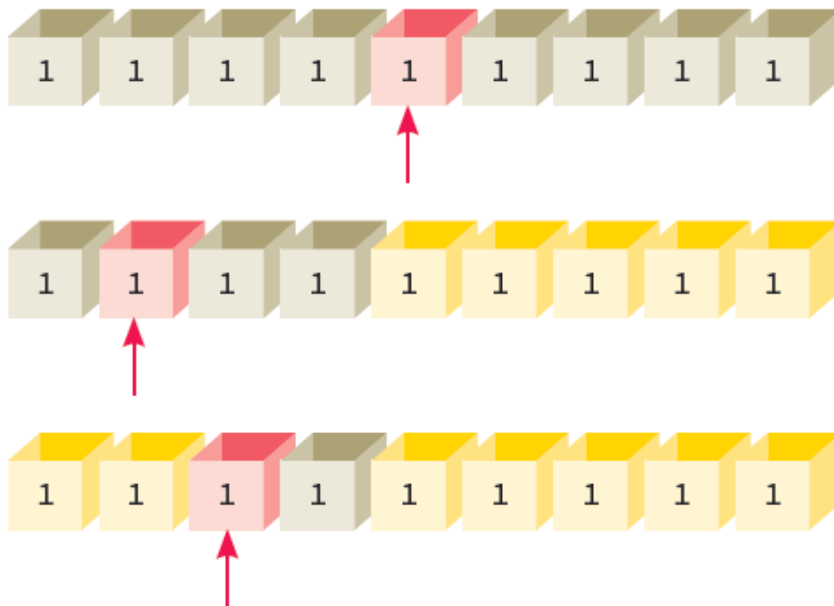
예제

```
import java.util.*;

public class Sort {
    public static void main(String[] args) {
        String[] sample = { "i", "walk", "the", "line" };
        List<String> list = Arrays.asList(sample);
        // 배열을 리스트로 변경
        Collections.sort(list);
        System.out.println(list);
    }
}
```

탐색

- 탐색이란 리스트 안에서 원하는 원소를 찾는 것이다.



예제

```
import java.util.*;

public class Search {
    public static void main(String[] args) {
        int key = 50;
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < 100; i++)
            list.add(i);          * 탐색전에 정렬되어야 함
        int index = Collections.binarySearch(list, key);
        System.out.println("탐색의 반환값 =" + index);
    }
}
```

사용자 클래스의 정렬

53

```
public class words implements Comparable<words> {  
    String eng;  
    String kor;  
    words(String eng, String kor){  
        this.eng = eng;  
        this.kor = kor;  
    }  
    @Override  
    public int compareTo(words arg0) {  
        // TODO Auto-generated method stub  
        return eng.compareTo(arg0.eng);  
    }  
}
```