# 07

# 컬렉션과 제네릭

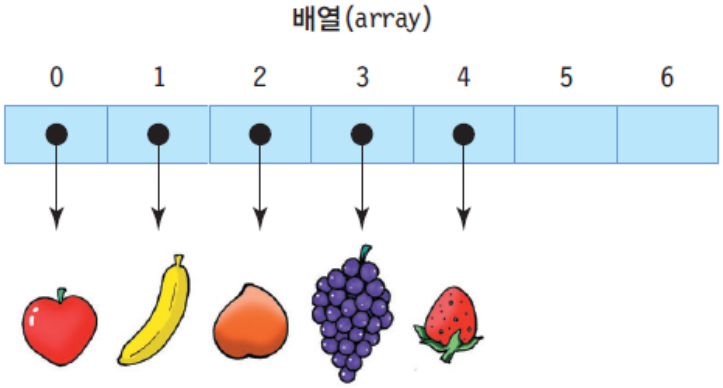
# 학습 목표

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



- 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

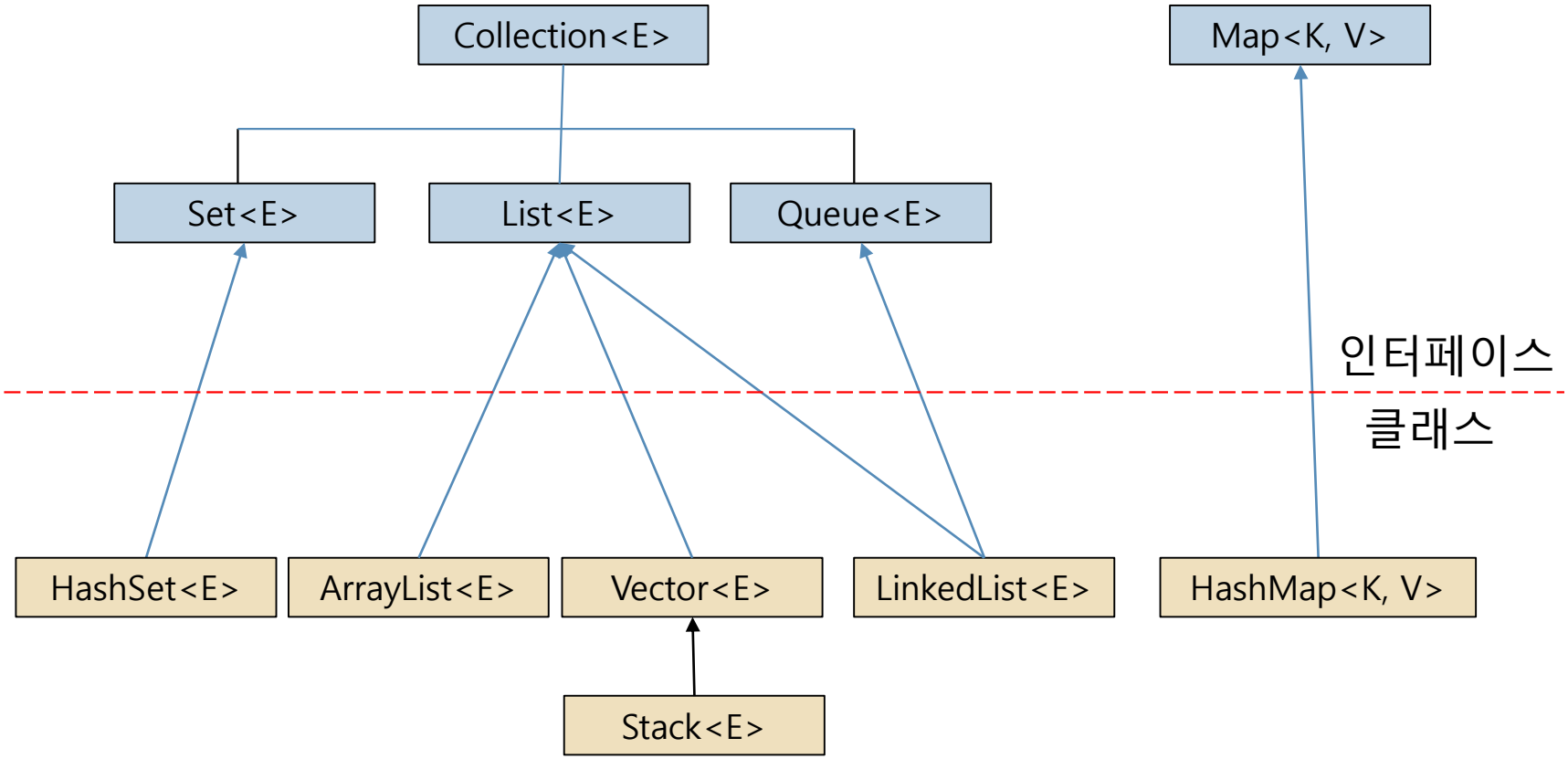



- 고정 크기 이상의 객체를 관리할 수 없다.
- 배열의 중간에 객체가 삭제되면 응용프로그램에서 자리를 옮겨야 한다.


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

# 컬렉션 자바 인터페이스와 클래스





# 컬렉션의 특징



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

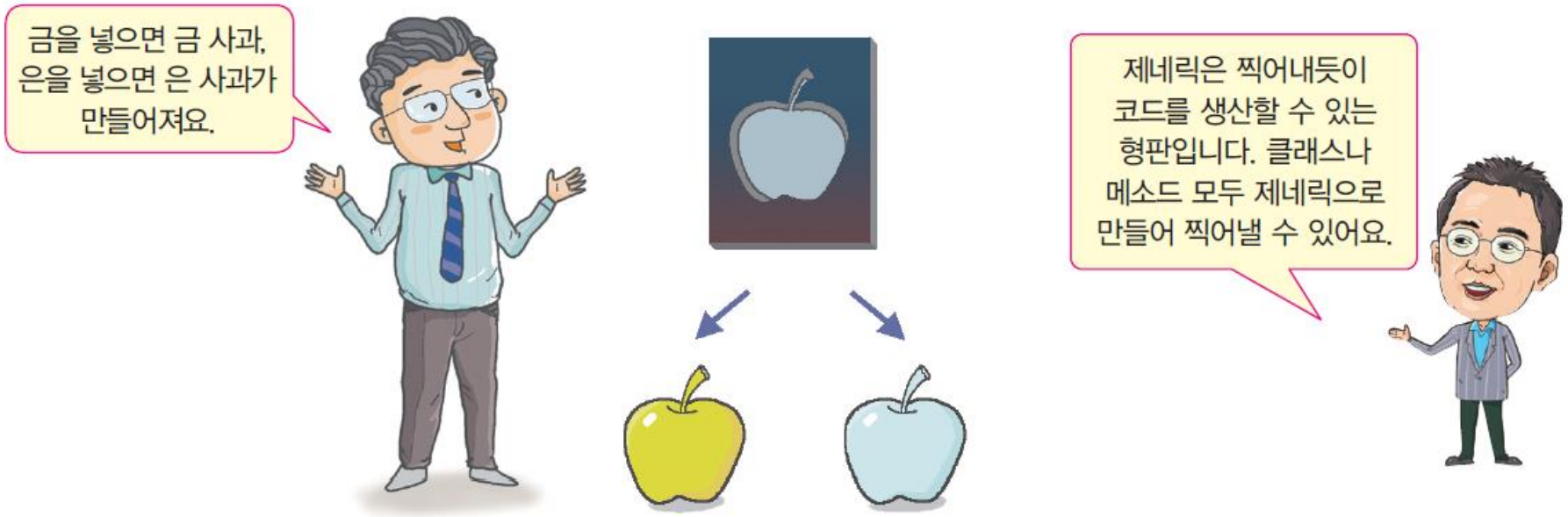
오류

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

# 제네릭은 형판과 같은 개념



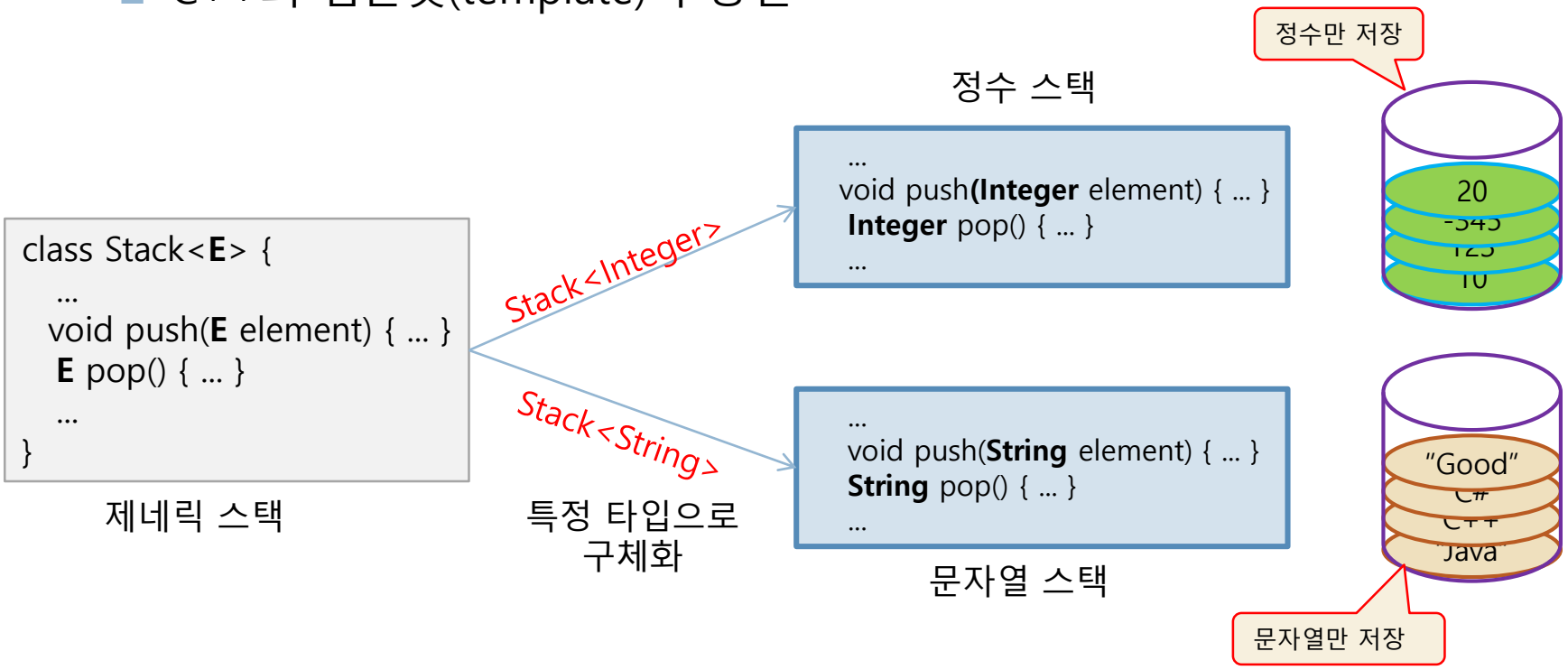
- 제네릭
  - 클래스나 메소드를 형판에서 찍어내듯이 생산할 수 있도록 일반화된 형판을 만드는 기법



# 제네릭의 기본 개념

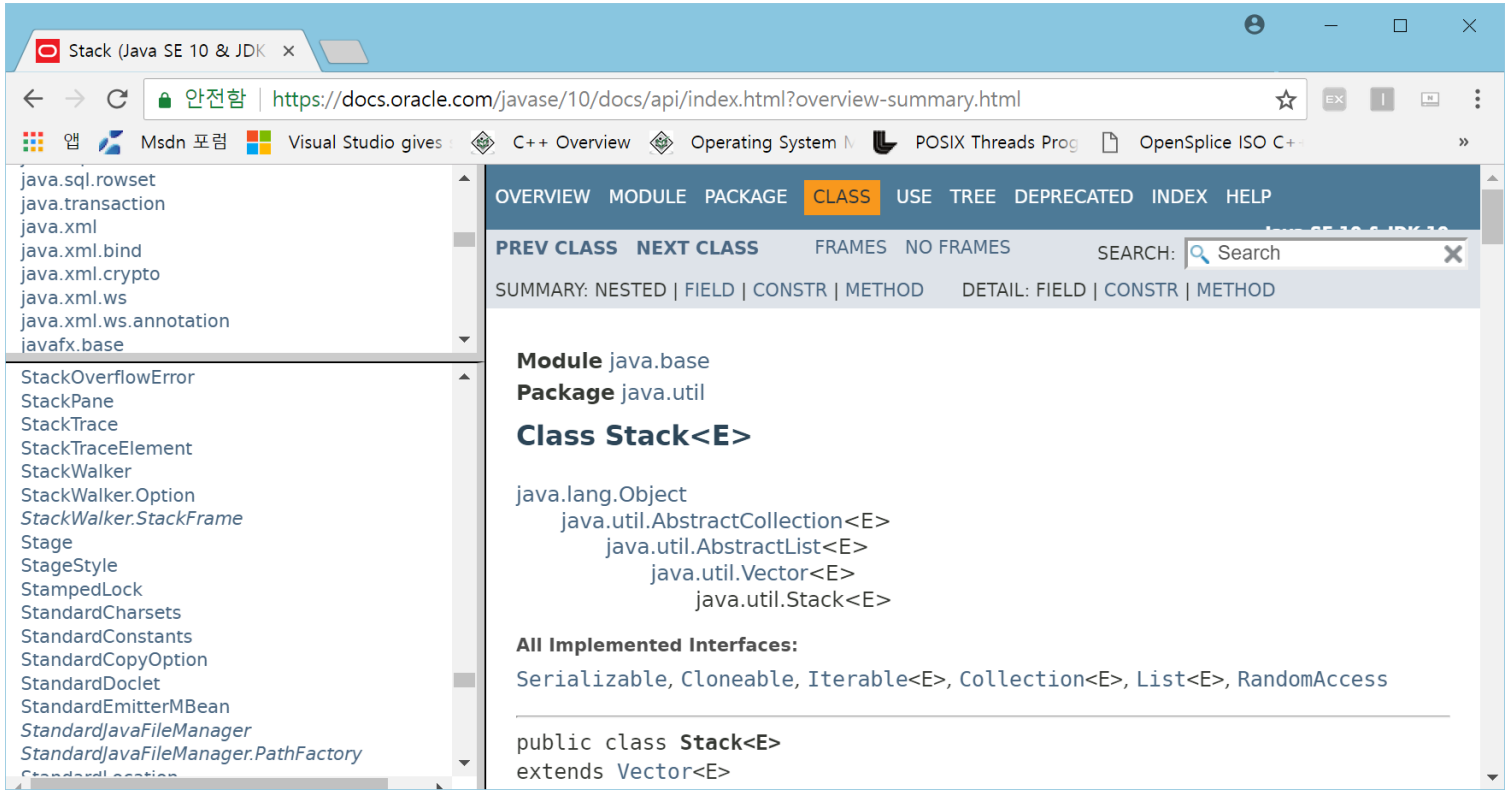


- 제네릭
  - JDK 1.5부터 도입(2004년 기점)
  - 모든 종류의 데이터 타입을 다룰 수 있도록 일반화된 타입 매개 변수로 클래스(인터페이스)나 메소드를 작성하는 기법
  - C++의 템플릿(template)과 동일



# 제네릭 Stack<E> 클래스





# Vector<E>



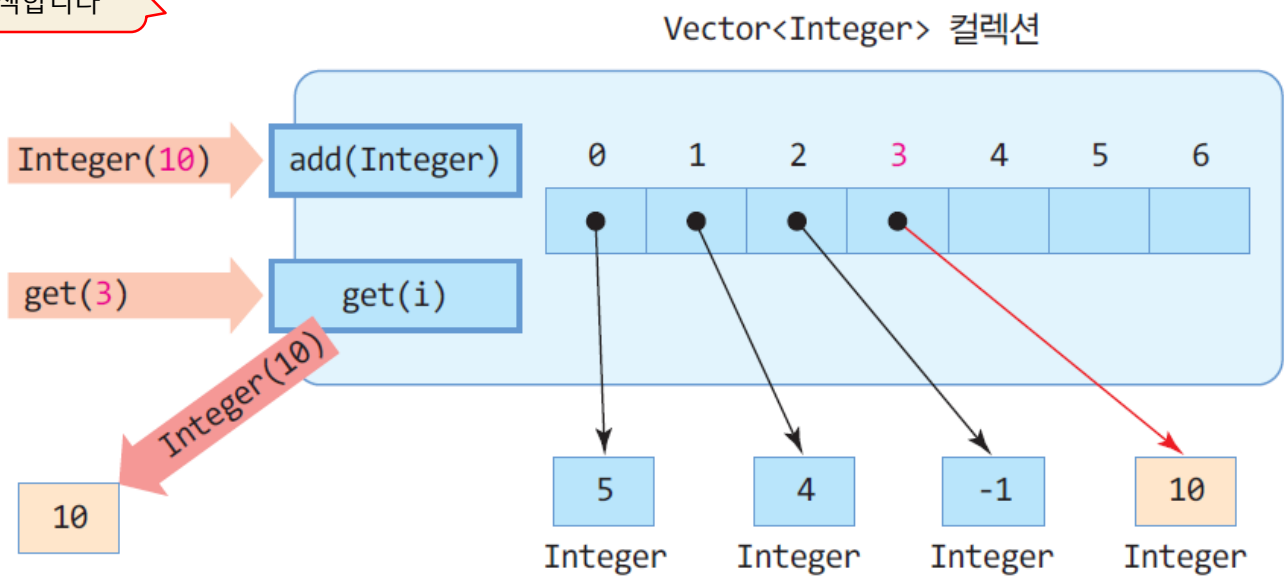
- 벡터 Vector<E>의 특성
  - <E>에 사용할 요소의 특정 타입으로 구체화
  - 배열을 가변 크기로 다룰 수 있게 하는 컨테이너
    - 배열의 길이 제한 극복
    - 요소의 개수가 넘치면 자동으로 길이 조절
  - 요소 객체들을 삽입, 삭제, 검색하는 컨테이너
    - 삽입, 삭제에 따라 자동으로 요소의 위치 조정
  - Vector에 삽입 가능한 것
    - 객체, null
    - 기본 타입의 값은 Wrapper 객체로 만들어 저장
  - Vector에 객체 삽입
    - 벡터의 맨 뒤, 중간에 객체 삽입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

# Vector<Integer> 벡터 컬렉션 내부



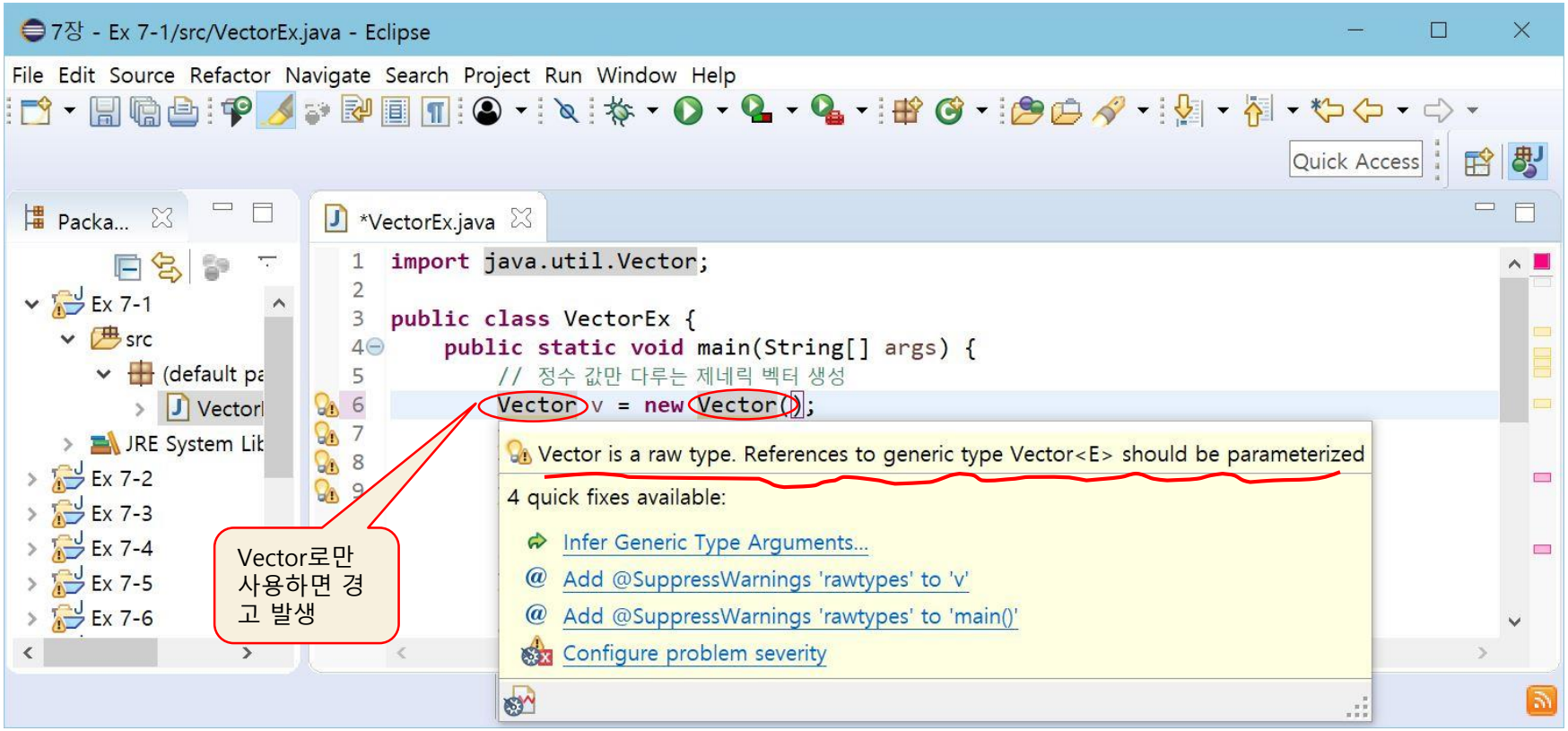
```
Vector<Integer> v = new Vector<Integer>();
```

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



# 타입 매개 변수 사용하지 않는 경우 경고 발생






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

# Vector<E> 클래스의 주요 메소드

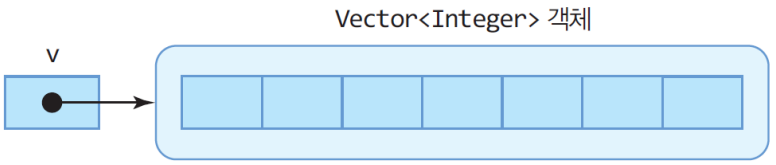


| 메소드 | 설명 |
|---|---|
| boolean add(E element) | 벡터의 맨 뒤에 element 추가 |
| void add(int index, E element) | 인덱스 index에 element를 삽입 |
| int capacity() | 벡터의 현재 용량 리턴 |
| boolean addAll(Collection<? extends E> c) | 컬렉션 c의 모든 요소를 벡터의 맨 뒤에 추가 |
| void clear() | 벡터의 모든 요소 삭제 |
| boolean contains(Object o)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

# Vector<Integer> 컬렉션 활용 사례

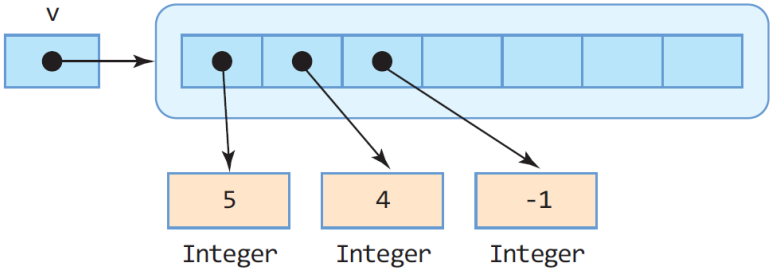
벡터 생성

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




요소 삽입

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




요소 개수 n
벡터의 용량 c

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

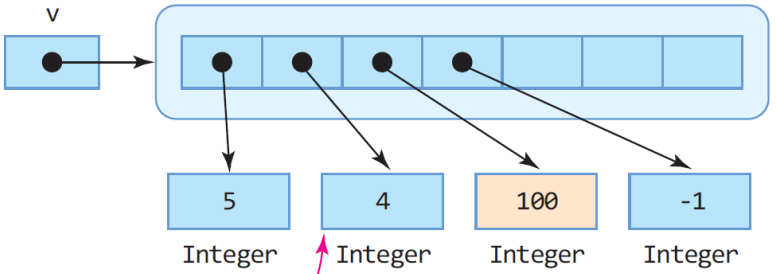
```
n = 3
c = 7
```

요소 중간 삽입

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

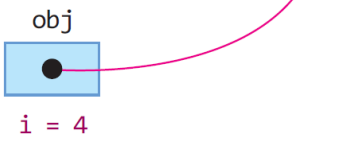
오류

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




요소 얻어내기

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



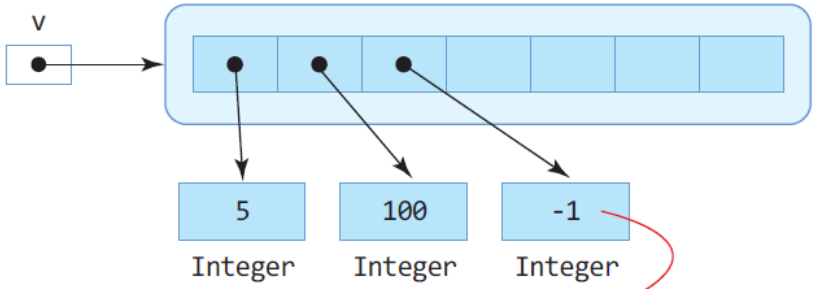
# Vector<Integer> 컬렉션 활용 사례(계속)

요소 삭제

```
v.remove(1);
```

오류

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

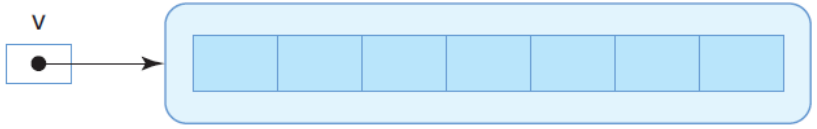



마지막 요소

```
int last = v.lastElement();
```


모든 요소 삭제

```
v.removeAllElements();
```



# 컬렉션과 자동 박싱/언박싱



- JDK 1.5 이전
  - 기본 타입 데이터를 Wrapper 객체로 만들어 삽입

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

오류

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

# 컬렉션 생성문의 진화 : Java 7, Java 10



- Java 7 이전

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

- Java 7 이후
  - 컴파일러의 타입 추론 기능 추가
  - <>(다이어몬드 연산자)에 타입 매개변수 생략

```
Vector<Integer> v = new Vector<>(); // Java 7부터 추가, 가능
```

- Java 10 이후
  - var 키워드 도입, 컴파일러의 지역 변수 타입 추론 가능

```
var v = new Vector<Integer>(); // Java 10부터 추가, 가능
```

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



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



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 + ")";
    }
}
```

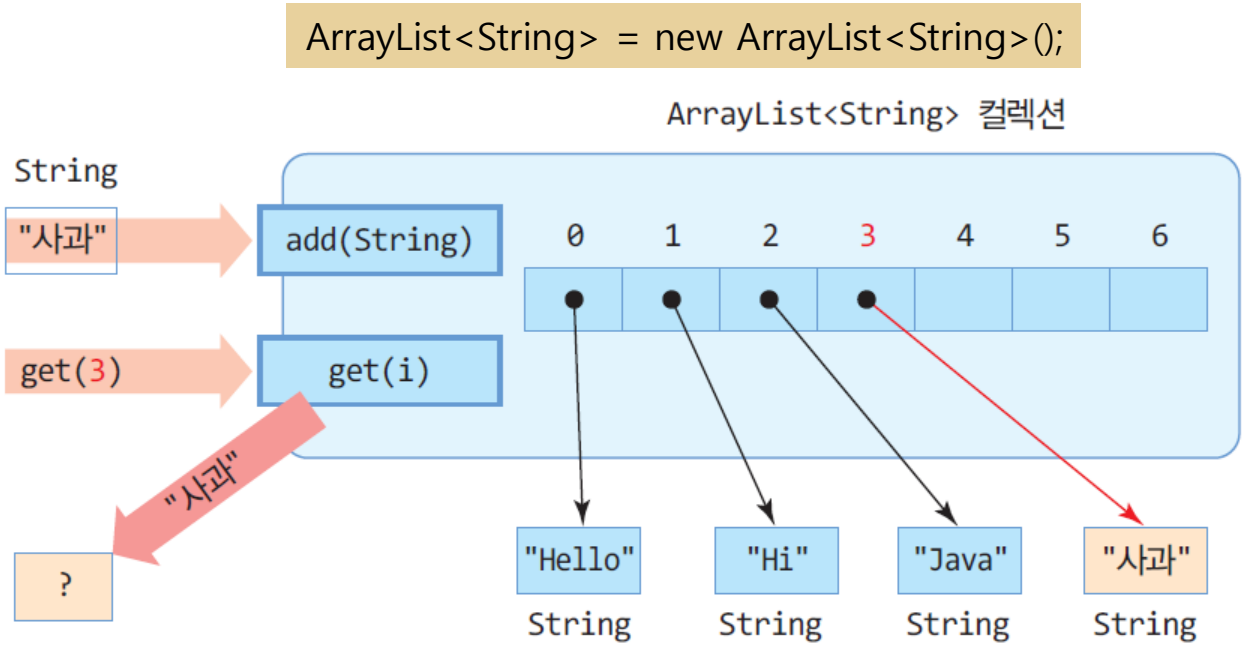
```
public class PointVectorEx {
    public static void main(String[] args) {
        Vector<Point> v = new Vector<Point>();

        // 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)
```

# ArrayList<E>



- 가변 크기 배열을 구현한 클래스
  - <E>에 요소로 사용할 특정 타입으로 구체화
- 벡터와 거의 동일
  - 요소 삽입, 삭제, 검색 등 벡터 기능과 거의 동일
  - 벡터와 달리 스레드 동기화 기능 없음
    - 다수 스레드가 동시에 *ArrayList*에 접근할 때 동기화되지 않음. 개발자가 스레드 동기화 코드 작성



# ArrayList<E> 클래스의 주요 메소드

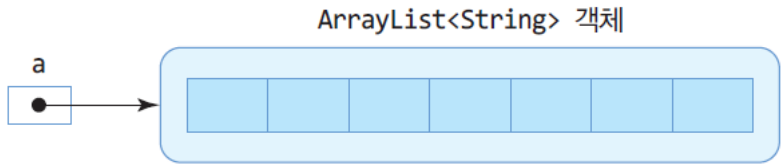


| 메소드 | 설명 |
|---|---|
| boolean add(E element) | ArrayList의 맨 뒤에 element 추가 |
| void add(int index, E element) | 인덱스 index에 지정된 element 삽입 |
| boolean addAll(Collection<? extends E> c) | 컬렉션 c의 모든 요소를 ArrayList의 맨 뒤에 추가 |
| void clear() | ArrayList의 모든 요소 삭제 |
| boolean contains(Object o) | ArrayList가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

# ArrayList<String> 컬렉션 활용 사례

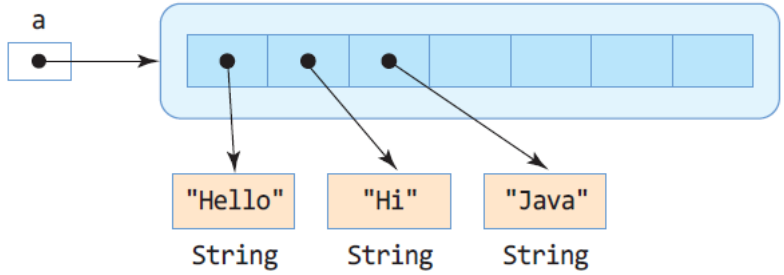
ArrayList 생성

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




요소 삽입

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




요소 개수 n
벡터의 용량 c

```
int n = a.size();  // n은 3
int c = a.capacity();  // capacity() 메소드 없음
```

오류

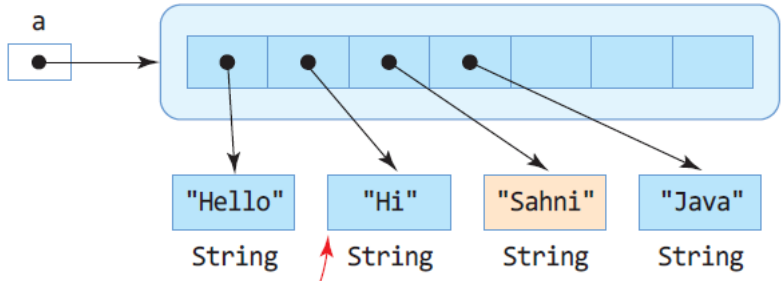
n = 3

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);
```

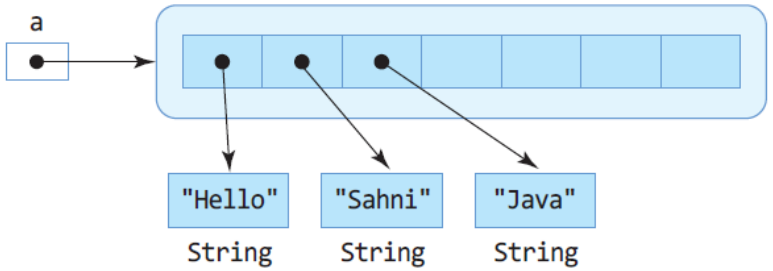
# ArrayList<String> 컬렉션 활용 사례(계속)

요소 삭제

```
a.remove(1);
```

오류

```
a.remove(4); // 오류
```




모든 요소 삭제

```
a.clear();
```

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



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
```

# 컬렉션의 순차 검색을 위한 Iterator



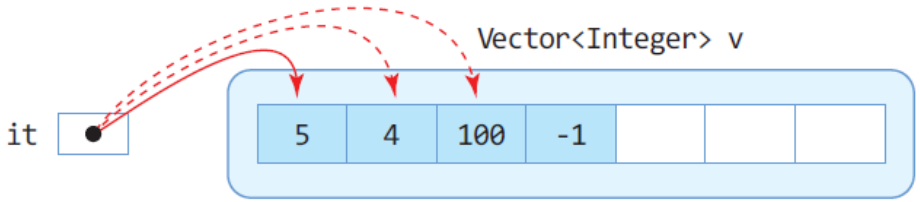
- Iterator<E> 인터페이스
  - 리스트 구조의 컬렉션에서 요소의 순차 검색을 위한 인터페이스
    - Vector<E>, ArrayList<E>, LinkedList<E>가 상속받는 인터페이스

- 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



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
```

# HashMap<K,V>



- 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

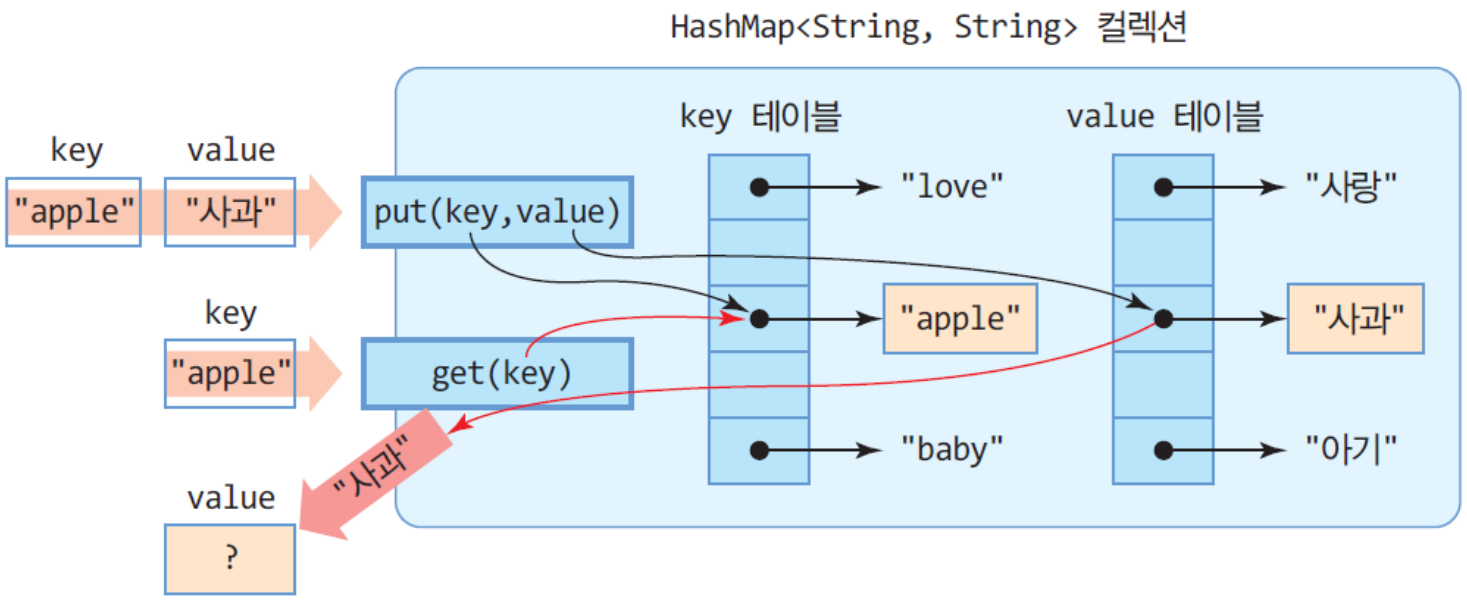
- 예) HashMap<String, String> 생성, 요소 삽입, 요소 검색

```
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



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



# HashMap<K,V>의 주요 메소드

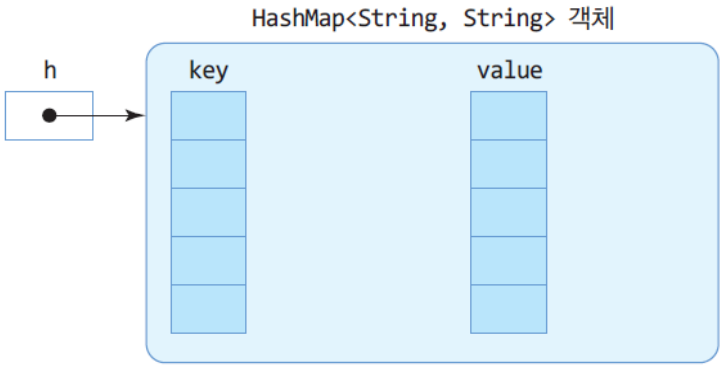


| 메소드 | 설명 |
|---|---|
| void clear() | HashMap의 모든 요소 삭제 |
| boolean containsKey(Object key) | 지정된 키(key)를 포함하고 있으면 true 리턴 |
| boolean containsValue(Object value) | 하나 이상의 키를 지정된 값(value)에 매핑시킬 수 있으면 true 리턴 |
| V get(Object key) | 지정된 키(key)에 매핑되는 값 리턴. 키에 매핑되는 어떤 값도 없으면 null 리턴 |
| boolean isEmpty()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

# HashMap<String, String> 컬렉션 활용 사례

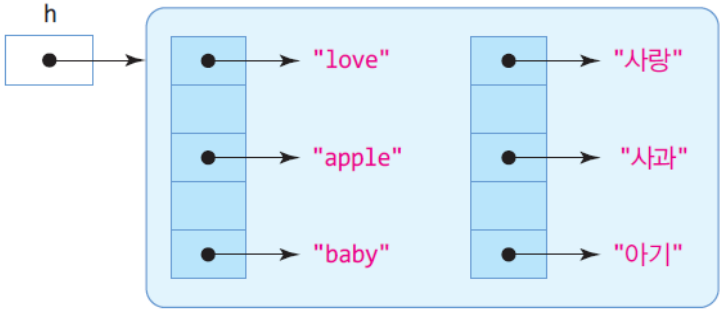
해시맵 생성

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




(키, 값) 삽입

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



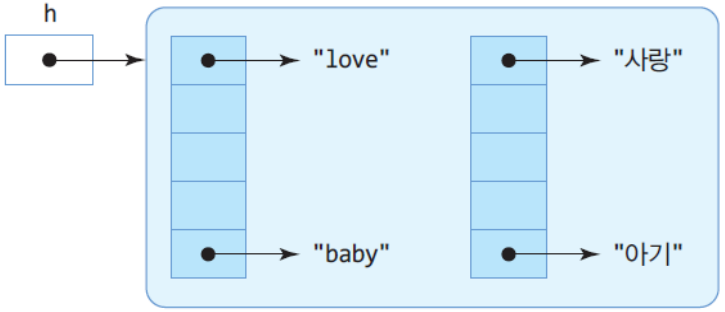

키로 값 읽기

```
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



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
```

# 제네릭 만들기



- 제네릭 클래스 작성
  - 클래스 이름 옆에 일반화된 타입 매개 변수 추가

```
public class MyClass<T> {
   T val;
   void set(T a) {
      val = a;
   }
   T get() {
      return val;
   }
}
```

val의 타입은 T

제네릭 클래스 MyClass 선언, 타입 매개 변수 T

T 타입의 값 a를 val에 지정

T 타입의 값 val 리턴

  - 제네릭 객체 생성 및 활용
    - 제네릭 타입에 구체적인 타입을 지정하여 객체를 생성하는 것을 **구체화**라고 함

```
MyClass<String> s = new MyClass<String>(); // T를 String으로 구체화
s.set("hello");
System.out.println(s.get()); // "hello" 출력

MyClass<Integer> n = new MyClass<Integer>(); // T를 Integer로 구체화
n.set(5);
System.out.println(n.get()); // 숫자 5 출력
```

# 예제 7-6 : 제네릭 스택 만들기



스택을 제네릭 클래스로 작성하고, String과 Integer형 스택을 사용하는 예를 보여라.

```
class GStack<T> {
    int tos;
    Object [] stck;
    public GStack() {
        tos = 0;
        stck = new Object [10];
    }
    public void push(T item) {
        if(tos == 10)
            return;
        stck[tos] = item;
        tos++;
    }
    public T pop() {
        if(tos == 0)
            return null;
        tos--;
        return (T)stck[tos];
    }
}
```

```
public class MyStack {
    public static void main(String[] args) {
        GStack<String> stringStack = new GStack<String>();
        stringStack.push("seoul");
        stringStack.push("busan");
        stringStack.push("LA");

        for(int n=0; n<3; n++)
            System.out.println(stringStack.pop());

        GStack<Integer> intStack = new GStack<Integer>();
        intStack.push(1);
        intStack.push(3);
        intStack.push(5);

        for(int n=0; n<3; n++)
            System.out.println(intStack.pop());
    }
}
```

```
LA
busan
seoul
5
3
1
```

# Generic type의 배열생성



- Generic Type으로 배열 생성
  - T 타입의 배열 생성은 근본적으로 불가능하지만…
    - 필요시 Object 타입의 배열 생성은 가능
    - T[] arr = new T[size]; // 불가능
  - T[] arr = (T[])(new Object[size]); // 가능

```
class MyClass<T>
{
    private T[] array;

    public MyClass(int size)
    {
        array = (T[])(new Object[size]);
    }
}
```

# Generic Type 파라메터



- Bounded type : 가능한 type의 제한
  - <T extends TYPE>
    - 지정된 TYPE 또는 그 하위 TYPE만 가능
- Wildcard type : 구체적 type지정 없음
  - <?>
    - 제한 없음(모든 타입)
  - <? extends TYPE >
    - 지정된 TYPE 또는 그 하위 TYPE만 가능
  - <? super TYPE >
    - 지정된 TYPE 또는 그 상위 TYPE만 가능

# Bounded type의 예



- 문자열은 String 클래스, 그리고
- 숫자 연산의 경우, Number 이하의 type만을 가져야 함

```
class MyUtil
{
   public static <T extends String> int compare(T s1, T s2)
   {
      return s1.compareTo(s2);
   }

   public static <T extends Number> int compare(T n1, T n2)
   {
      return Double.compare(n1.doubleValue(),  n2.doubleValue());
   }
}
```

숫자형 Wrapper 클래스들의 부모 클래스

```
public class GenericEx
{
   public static void main(String[] args)
   {
      int result1 = MyUtil.compare("abcde", 123); // 오류
      int result2 = MyUtil.compare(10, 20); // Integer
      int result3 = MyUtil.compare(5.14, 4.5); // Double
      int result4 = MyUtil.compare(20, 20.5); // Double:auto-boxing
      int result5 = MyUtil.compare(30L, 30L); // Long
      System.out.println("result1 = " + result1);
      System.out.println("result2 = " + result2);
      System.out.println("result3 = " + result3);
      System.out.println("result4 = " + result4);
      System.out.println("result5 = " + result5);
   }
}
```

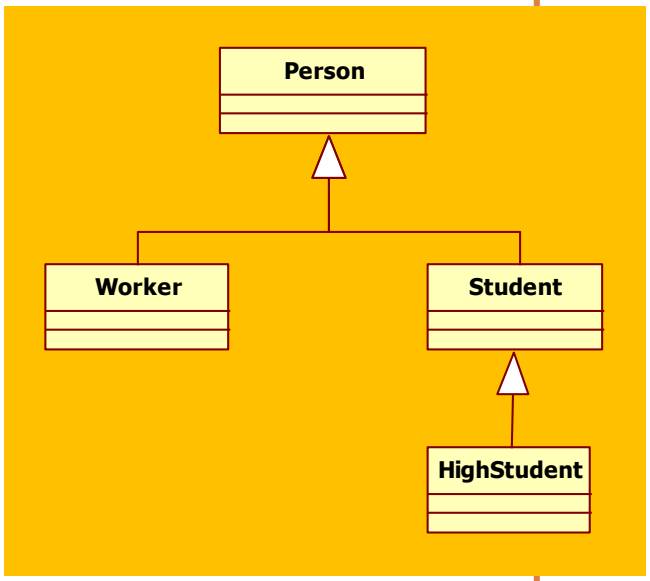
```
result1 = 2
result2 = -1
result3 = 1
result4 = -1
result5 = 0
```

# Wildcard type의 예



```
class Course<T>
{
    private String name;
    private T[] students;

    public Course(String name, int num)
    {
        this.name = name;
        students = (T[])(new Object[num]);
    }
    public void add(T student) {
        for(int i=0; i<students.length; i++) {
            if(students[i] == null){
                students[i] = student;
                break;
            }
        }
    }
    public String getName() { return name; }
    public T[] getStudents() { return students; }
}
```

```
class Person {}
class Worker extends Person {}
class Student extends Person {}
class HighStudent extends Student {}
```



```
public class GenericEx
{
	public static void registerCourse(Course<?> course){}
	public static void registerStudentCourse(Course<? extends Student> course) {}
	public static void registerWorkerCourse(Course<? super Worker> course) {}

	public static void main(String[] args) {
		Course<Person> course_P = new Course<Person>("일반인과정", 5);
		Course<Worker> course_W = new Course<Worker>("직장인과정", 5);
		Course<Student> course_S = new Course<Student>("대학생과정", 5);
		Course<HighStudent> course_H = new Course<HighStudent>("고등학생과정", 5);

		// 모든코스 등록 가능
		registerCourse(course_P);
		registerCourse(course_W);
		registerCourse(course_S);
		registerCourse(course_H);
		// 학생코스만 등록 가능(Student, HighStudent)
	//	registerStudentCourse(course_P);
	//	registerStudentCourse(course_W);
		registerStudentCourse(course_S);
		registerStudentCourse(course_H);
		// 일반직장인코스만 등록 가능(Person, Worker)
		registerWorkerCourse(course_P);
		registerWorkerCourse(course_W);
	//	registerWorkerCourse(course_S);
	//	registerWorkerCourse(course_H);
	}
}
```