Power Java

제14장 제네릭과 컬렉션

이번 장에서 학습할 내용

- 제네릭 클래스
- 제네릭 메소드
- 컬렉션
- ArrayList
- LinkedList
- Set
- Queue
- Map
- Collections 클래스

제네릭이란?

- 제네릭 프로그래밍(generic programming)
  - 일반적인 코드를 작성하고 이 코드를 다양한 타입의 객체에 대하여 재사용하는 프로그래밍 기법
  - 제네릭은 컬렉션 라이브러리에 많이 사용

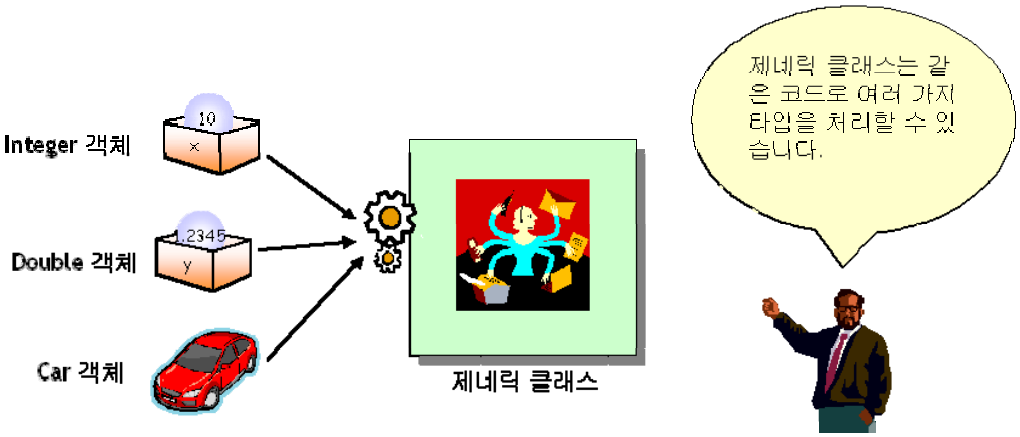
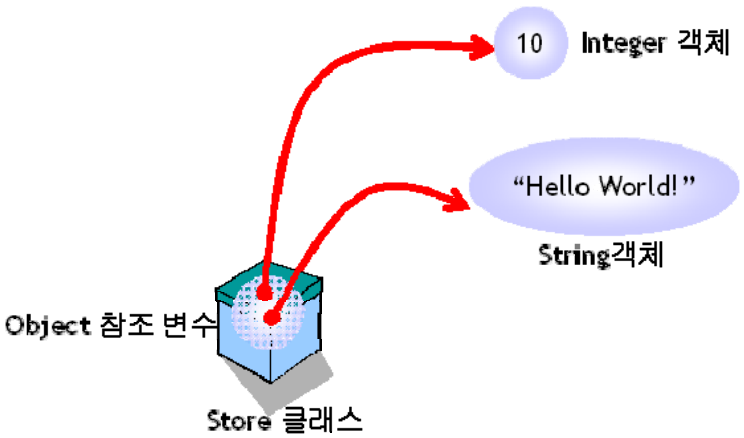


그림 14-1 제네릭 프로그래밍의 개념

기존의 방법

- 일반적인 객체를 처리하려면 Object 참조 변수를 사용
- Object 참조 변수는 어떤 객체이던지 참조할 수 있다.
- 예제로 하나의 데이터를 저장하는 Store 클래스를 살펴보자.



기존의 방법(코드)

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

```
Store s = new Store();
s.set(new Integer(10));        // ①정수 객체 저장
s.set("Hello World!");         // 정수 객체가 없어지고 문자열 객체를 저장
String s = (String)s.get();    // ②Object 타입을 String 타입으로 형변환
```

```
s.set("Hello World!");
Integer i = (Integer)s.get( );          // 오류! 문자열을 정수 객체로 형변환
```

문제점 발생!!

제네릭을 이용한 버전

```
class Store<T> {
    private T data;            // T 는 타입을 의미한다.
    public void set(T data)   {        this.data = data;       }
    public T get()            {        return data;            }
}
```

- 문자열을 저장하려면 다음과 같이 선언
  - Store<String> store = **new** Store<String>();
- 정수를 저장하려면 다음과 같이 선언
  - Store<Integer> store = **new** Store<Integer>();

제네릭 버전의 사용

- 문자열을 저장하는 DataStore 객체를 생성하여 사용하면 다음과 같다.
  - Store<String> store = new Store<String>();
  - store.set("Hello World!"); // 문자열 타입 저장
  - String s=(String)store.get();
  - System.*out*.println(s);

- 만약 Store<String>에 정수 타입을 추가하려고 하면 컴파일러가 컴파일 단계에서 오류를 감지할 수 있다. 따라서 더 안전하게 프로그래밍할 수 있다.
  - Store<String> store = new Store<String>();
  - store.set(new Integer(10)); // 정수 타입을 저장하려고 하면 컴파일 오류!

중간 점검 문제

1. 왜 데이터를 Object 참조형 변수에 저장하는 것이 위험할 수 있는가?
2. Store 객체에 Rectangle 객체를 저장하도록 제네릭을 이용하여 생성하여 보라.
3. 타입 매개 변수 T를 가지는 Point 클래스를 정의하여 보라. Point 클래스는 2차원 공간에서 점을 나타낸다.

제네릭 메소드

- 메소드에서도 타입 매개 변수를 사용하여서 제네릭 메소드를 정의할 수 있다.
- 타입 매개 변수의 범위가 메소드 내부로 제한된다.

```
public class Array
{
    public static <T> T getLast(T[] a)
    {
        return a[a.length-1];
    }
}
```

배열의 마지막 원소를 반환하는 메소드

제네릭 메소드의 사용

- String[] language= { "C++", "C#", "JAVA" };
- String last = Array.<String>getLast(language); // last는 "JAVA"

- 또는
- String last = Array.getLast(language); // last는 "JAVA"

중간 점검 문제

1. 제네릭 메소드 sub()에서 매개 변수 d를 타입 매개 변수를 이용하여서 정의하여 보라.

2. displayArray()라는 메소드는 배열을 매개 변수로 받아서 반복 루프를 사용하여서 배열의 원소를 화면에 출력한다. 어떤 타입의 배열도 처리할 수 있도록 제네릭 메소드로 정의하여 보라.

컬렉션

- 컬렉션(collection)은 자바에서 자료 구조를 구현한 클래스
- 자료 구조로는 리스트(list), 스택(stack), 큐(queue), 집합(set), 해쉬 테이블(hash table) 등이 있다.

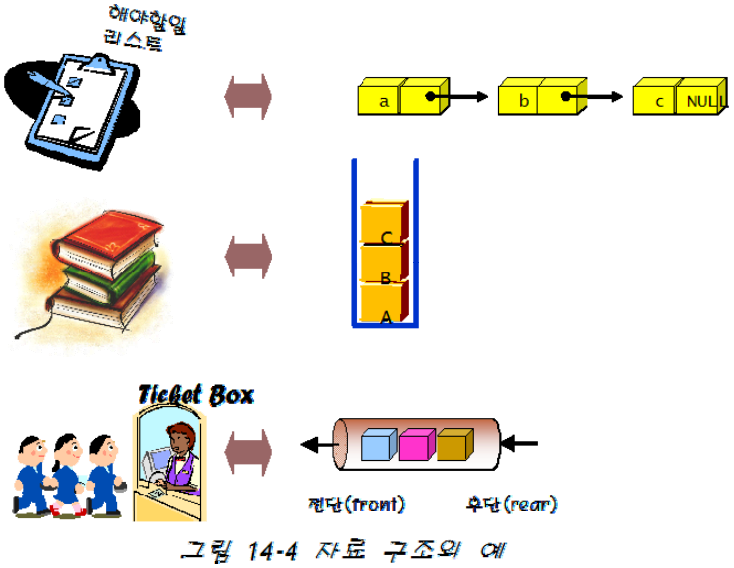
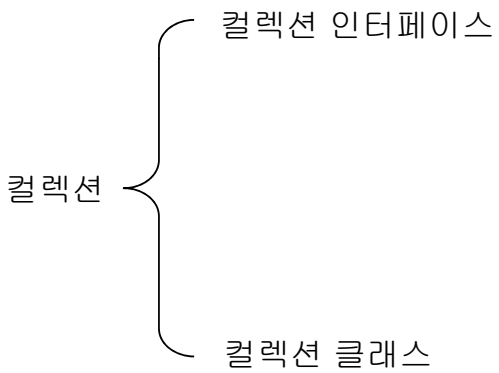


그림 14-4 자료 구조의 예

자바에서의 컬렉션



컬렉션 인터페이스

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

컬렉션의 역사

- 초기 버전: Vector, Stack, HashTable, Bitset, Enumeration이 그것이다.
- 버전 1.2부터는 풍부한 컬렉션 라이브러리가 제공
  - 인터페이스와 구현을 분리
  - (예) List 인터페이스를 ArrayList와 LinkedList 클래스가 구현

컬렉션 인터페이스

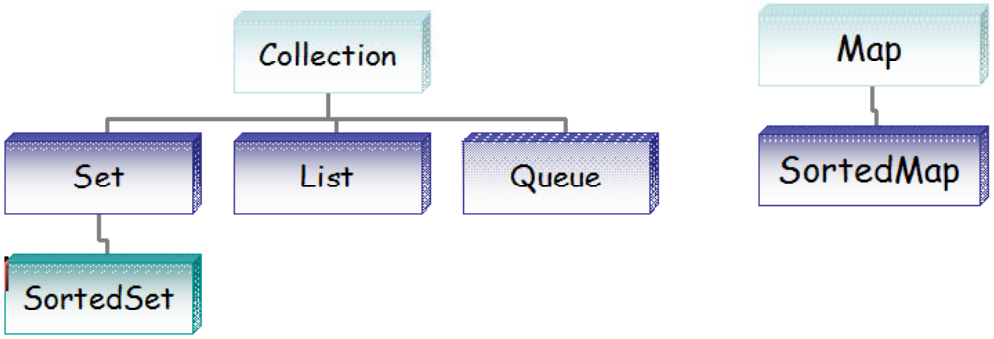
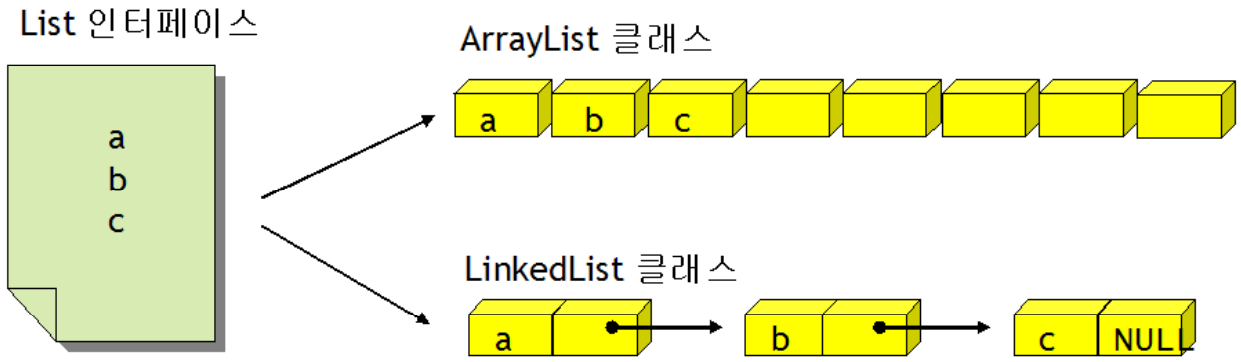


그림 14-5 인터페이스들의 계층 구조

인터페이스가 제공하는 메소드

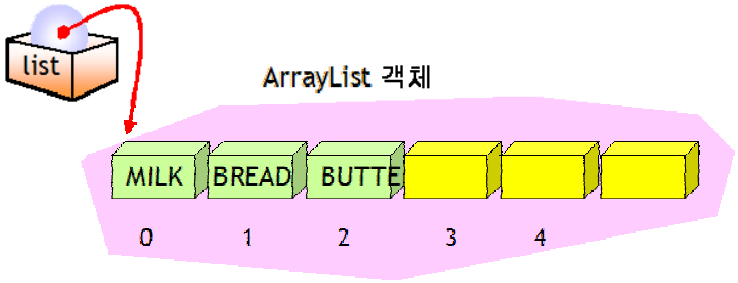
분류	메소드	설명
기본 연산	int size()	원소의 개수 반환
	boolean isEmpty()	공백 상태이면 true 반환
	boolean contains(Object obj)	obj를 포함하고 있으면 true 반환
	boolean add(E element);	원소 추가
	boolean remove(Object obj)	원소 삭제
	Iterator<E> iterator();	원소 방문
벌크 연산	boolean addAll(Collection<? extends E> from)	c에 있는 모든 원소 추가
	boolean containsAll(Collection<?> c)	c에 있는 모든 원소가 포함되어 있으면 true
	boolean removeAll(Collection<?> c)	c에 있는 모든 원소 삭제
	void clear()	모든 원소 삭제
배열 연산	Object[] toArray()	컬렉션을 배열로 변환
	<T> T[] toArray(T[] a);	컬렉션을 배열로 변환

List 인터페이스

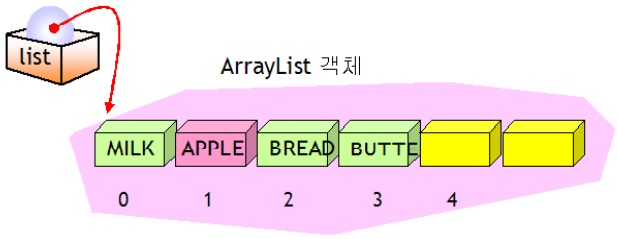


ArrayList

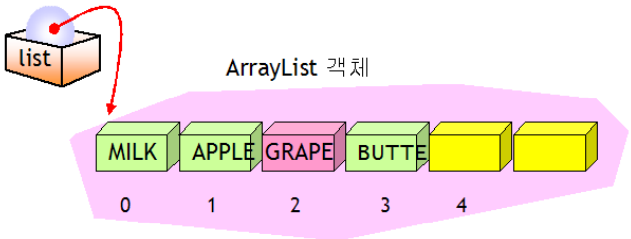
- ArrayList를 배열(Array)의 향상된 버전 또는 가변 크기의 배열이라고 생각하면 된다.
- ArrayList의 생성
  - ArrayList<String> list = **new** ArrayList<String>();
- 원소 추가
  - list.add("MILK");
  - list.add("BREAD");
  - list.add("BUTTER");



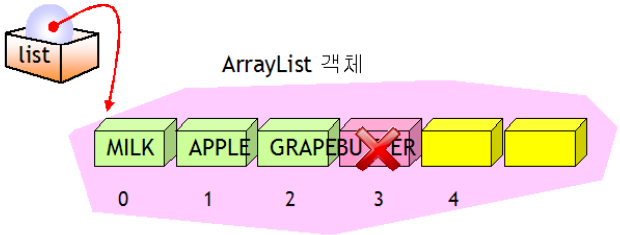
```
list.add( 1, "APPLE" );        // 인덱스 1에 "APPLE"을 삽입
```




```
list.set( 2, "GRAPE" );        // 인덱스 2의 원소를 "GRAPE"로 대체
```




```
list.remove( 3 );        // 인덱스 3의 원소를 삭제한다.
```



예제

ArrayListTest.java

```
import java.util.*;

public class ArrayListTest {
        public static void main(String args[]) {
                ArrayList<String> list = new ArrayList<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));
        }
}
```

실행결과

```
MILK
APPLE
GRAPE
```


반복자

- 반복자(iterator): 반복자는 컬렉션의 원소들을 하나씩 처리하는데 사용

```
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);
}
```

LinkedList

- 빈번하게 삽입과 삭제가 일어나는 경우에 사용

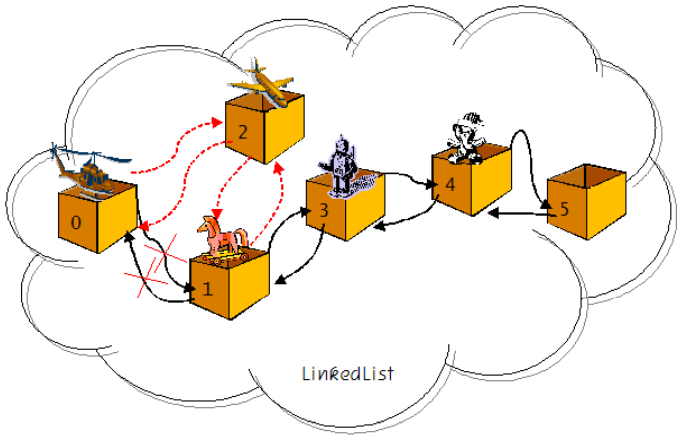


그림 14-8 연결 리스트 중간에 삽입하려면 링크만 수정하면 된다.

예제

LinkedListTest.java

```
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));
        }
}
```

배열을 리스트로 변환하기

- List<String> list = Arrays.asList(new String[size]);
  - 일반적인 배열을 리스트로 변환한다.

중간 점검 문제

1. ArrayList와 LinkedList의 차이점은 무엇인가?
2. 어떤 경우에 LinkedList를 사용하여야 하는가?

Set

- 집합(Set)은 원소의 중복을 허용하지 않는다.

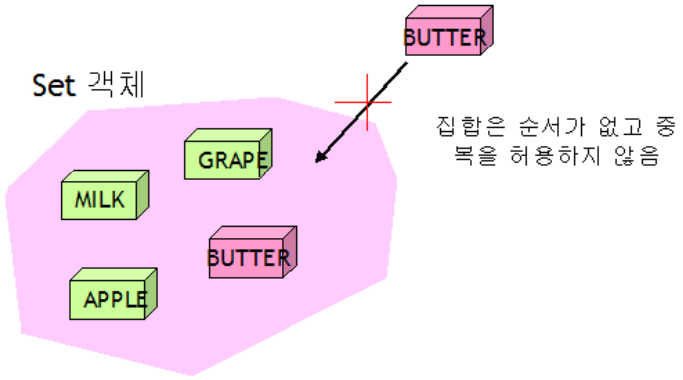


그림 14-9 집합

Set 인터페이스를 구현하는 방법

- 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.

예제

SetTest.java

```
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);
  }
}
```

[Bread, Milk, Butter, Ham, Cheese]

대량 연산 메소드

- s1.containsAll(s2) – 만약 s2가 s1의 부분 집합이면 참이다.
- s1.addAll(s2) – s1을 s1과 s2의 합집합으로 만든다.
- s1.retainAll(s2) – s1을 s1과 s2의 교집합으로 만든다.
- s1.removeAll(s2) – s1을 s1과 s2의 차집합으로 만든다.

예제

FindDupplication.java

```
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);
        }
}
```

실행결과

```
중복된 단어 중복
3 중복되지 않은 단어: [중복, 구절, 단어]
```


```
import java.util.*;
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);
    }
}
```

실행결과

```
합집합 [D, A, B, C]
교집합 [A]
```

Queue

- 큐는 먼저 들어온 데이터가 먼저 나가는 자료 구조
- FIFO(First-In First-Out)

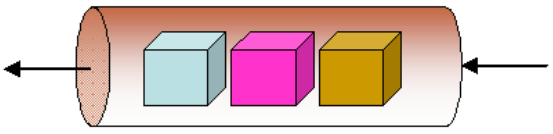


그림 14-11 큐

Queue 인터페이스

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

예제

QueueTest.java

```
import java.util.*;

public class QueueTest {
    public static void main(String[] args) throws InterruptedException {
        int time = 10;
        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--)
            queue.add(i);
        while (!queue.isEmpty()) {
            System.out.print(queue.remove()+" ");
            Thread.sleep(1000);        // 현재의 스레드를 1초간 재운다.
        }
    }
}
```

실행결과

```
10 9 8 7 6 5 4 3 2 1 0
```

우선 순위큐

- 우선순위큐(priority queue): 우선순위를 가진 항목들을 저장하는 큐
- FIFO 순서가 아니라 우선 순위가 높은 데이터가 먼저 나가게 된다.
- 가장 일반적인 큐: 스택이나 FIFO 큐를 우선순위큐로 구현할 수 있다.

예제

PriorityQueueTest.java

```
import java.util.*;

public class PriorityQueueTest {
        public static void main(String[] args) {
                PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
                pq.add(30);
                pq.add(80);
                pq.add(20);

                for (Integer o : pq)
                        System.out.println(o);
                System.out.println("원소 삭제");
                while (!pq.isEmpty())
                        System.out.println(pq.remove());
        }
}
```

실행결과

```
20
80
30
원소 삭제
20
30
80
```


Map

- 사전과 같은 자료 구조
- 키(key)에 값(value)이 매핑된다.

Map

그림 14-12 Map의 개념

예제

MapTest.java

```
import java.util.*;

class Student {
        int number;
        String name;

        public Student(int number, String name) {
                this.number = number;
                this.name = name;
        }

        public String toString() {
                return name;
        }
}
```


```
public class MapTest {
        public static void main(String[] args) {
                Map<String, Student> st = new HashMap<String, Student>();
                st.put("20090001", new Student(20090001, "구준표"));
                st.put("20090002", new Student(20090002, "금잔디"));
                st.put("20090003", new Student(20090003, "윤지후"));

                // 모든 항목을 출력한다.
                System.out.println(st);

                // 하나의 항목을 삭제한다.
                st.remove("20090002");
                // 하나의 항목을 대치한다.
                st.put("20090003", new Student(20090003, "소이정"));
                // 값을 참조한다.
                System.out.println(st.get("20090003"));
                // 모든 항목을 방문한다.
                for (Map.Entry<String, Student> s : st.entrySet()) {
                        String key = s.getKey();
                        Student value = s.getValue();
                        System.out.println("key=" + key + ", value=" + value);
                }
        }
}
```

실행결과

```
{20090001=구준표, 20090002=금잔디, 20090003=윤지후}
소이정
key=20090001, value=구준표
key=20090003, value=소이정
```

Q & A