

1.8 HashSet

HashSet은 Set인터페이스를 구현한 가장 대표적인 컬렉션이며, Set인터페이스의 특징대로 HashSet은 중복된 요소를 저장하지 않는다.

HashSet에 새로운 요소를 추가할 때는 add메서드나 addAll메서드를 사용하는데, 만일 HashSet에 이미 저장되어 있는 요소와 중복된 요소를 추가하고자 한다면 이 메서드들은 false를 반환함으로써 중복된 요소이기 때문에 추가에 실패했다는 것을 알린다.

이러한 HashSet의 특징을 이용하면, 컬렉션 내의 중복 요소들을 쉽게 제거할 수 있다.

ArrayList와 같이 List인터페이스를 구현한 컬렉션과 달리 HashSet은 저장순서를 유지하지 않으므로 저장순서를 유지하고자 한다면 LinkedHashSet을 사용해야한다.

| 참고 | HashSet은 내부적으로 HashMap을 이용해서 만들어졌으며, HashSet이란 이름은 해싱(hashing)을 이용해서 구현했기 때문에 붙여진 것이다. 해싱(hashing)에 대한 자세한 내용은 HashMap에서 설명한다.

생성자 또는 메서드	설명
HashSet()	HashSet객체를 생성한다.
HashSet(Collection c)	주어진 컬렉션을 포함하는 HashSet객체를 생성한다.
HashSet(int initialCapacity)	주어진 값을 초기용량으로하는 HashSet객체를 생성한다.
HashSet(int initialCapacity, float loadFactor)	초기용량과 load factor를 지정하는 생성자.
boolean add(Object o)	새로운 객체를 저장한다.
boolean addAll(Collection c)	주어진 컬렉션에 저장된 모든 객체들을 추가한다.(합집합)
void clear()	저장된 모든 객체를 삭제한다.
Object clone()	HashSet을 복제해서 반환한다.(얕은 복사)
boolean contains(Object o)	지정된 객체를 포함하고 있는지 알려준다.
boolean containsAll(Collection c)	주어진 컬렉션에 저장된 모든 객체들을 포함하고 있는지 알려준다.
boolean isEmpty()	HashSet이 비어있는지 알려준다.
Iterator iterator()	Iterator를 반환한다.
boolean remove(Object o)	지정된 객체를 HashSet에서 삭제한다.(성공하면 true, 실패하면 false)
boolean removeAll(Collection c)	주어진 컬렉션에 저장된 모든 객체와 동일한 것들을 HashSet에서 모두 삭제한다.(차집합)
boolean retainAll(Collection c)	주어진 컬렉션에 저장된 객체와 동일한 것만 남기고 삭제한다.(교집합)
int size()	저장된 객체의 개수를 반환한다.
Object[] toArray()	저장된 객체들을 객체배열의 형태로 반환한다.
Object[] toArray(Object[] a)	저장된 객체들을 주어진 객체배열(a)에 담는다.

▲ 표 11-15 HashSet의 메서드

| 참고 | load factor는 컬렉션 클래스에 저장공간이 가득 차기 전에 미리 용량을 확보하기 위한 것으로 이 값을 0.8로 지정하면, 저장공간의 80%가 채워졌을 때 용량이 두 배로 늘어난다. 기본값은 0.75, 즉 75%이다.

| 참고 | JDK1.8부터 추가된 스트림(Stream)과 관련된 메서드들이 추가되었으나 메서드 목록에 넣지 않았다. 관련 내용은 14장 [람다와 스트림](#)에서 다룬다.

▼ 예제 11-20/ch11/HashSetEx1.java

```

import java.util.*;

class HashSetEx1 {
    public static void main(String[] args) {
        Object[] objArr = {"1", new Integer(1), "2", "2", "3", "3", "4", "4", "4"};
        Set set = new HashSet();
        for(int i=0; i < objArr.length; i++) {
            set.add(objArr[i]);           // HashSet에 objArr의 요소들을 저장한다.
        }
        // HashSet에 저장된 요소들을 출력한다.
        System.out.println(set);
    }
}

```

▼ 실행결과
[1, 1, 2, 3, 4]

결과에서 알 수 있듯이 중복된 값은 저장되지 않았다. add메서드는 객체를 추가할 때 HashSet에 이미 같은 객체가 있으면 중복으로 간주하고 저장하지 않는다. 그리고는 작업이 실패했다는 의미로 false를 반환한다.

'1'이 두 번 출력되었는데, 둘 다 '1'로 보이기 때문에 구별이 안 되지만, 사실 하나는 String인스턴스이고 다른 하나는 Integer인스턴스로 서로 다른 객체이므로 중복으로 간주하지 않는다.

Set을 구현한 컬렉션 클래스는 List를 구현한 컬렉션 클래스와 달리 순서를 유지하지 않기 때문에 저장한 순서와 다를 수 있다.

만일 중복을 제거하는 동시에 저장한 순서를 유지하고자 한다면 HashSet대신 LinkedHashSet을 사용해야한다.

▼ 예제 11-21/ch11/HashSetLotto.java

```

import java.util.*;

class HashSetLotto {
    public static void main(String[] args) {
        Set set = new HashSet();

        for (int i = 0; set.size() < 6 ; i++) {
            int num = (int)(Math.random()*45) + 1;
            set.add(new Integer(num));
        }

        List list = new LinkedList(set); // LinkedList(Collection c)
        Collections.sort(list);        // Collections.sort(List list)
        System.out.println(list);
    }
}

```

▼ 실행결과
[7, 11, 17, 18, 24, 28]

중복된 값은 저장되지 않는 HashSet의 성질을 이용해서 로또번호를 만드는 예제이다. Math.random()을 사용했기 때문에 실행할 때마다 결과가 다른 것이다.

번호를 크기순으로 정렬하기 위해서 Collections클래스의 sort(List list)를 사용했다. 이 메서드는 인자로 List인터페이스 타입을 필요로 하기 때문에 LinkedList클래스의 생성자 LinkedList(Collection c)를 이용해서 HashSet에 저장된 객체들을 LinkedList에 담아서 처리했다.

실행결과의 정렬기준은, 컬렉션에 저장된 객체가 Integer이기 때문에 Integer클래스에 정의된 기본정렬이 사용되었다. 정렬기준을 변경하는 방법과 Collections클래스에 대해서는 이장의 끝부분에서 자세히 다룰 것이다.

| 참고 | Collection은 인터페이스고 Collections는 클래스임에 주의하자.

▼ 예제 11-22/ch11/Bingo.java

```
import java.util.*;

class Bingo {
    public static void main(String[] args) {
        Set set = new HashSet();
        // Set set = new LinkedHashSet();
        int[][] board = new int[5][5];

        for(int i=0; set.size() < 25; i++) {
            set.add((int)(Math.random()*50)+1+"");
        }

        Iterator it = set.iterator();

        for(int i=0; i < board.length; i++) {
            for(int j=0; j < board[i].length; j++) {
                board[i][j] = Integer.parseInt((String)it.next());
                System.out.print((board[i][j] < 10 ? " " : " ") + board[i][j]);
            }
            System.out.println();
        }
    } // main
}
```

▼ 실행결과					
28	3	29	14	6	
1	39	16	50	11	
4	23	12	34	45	
30	20	42	31	25	
32	13	26	44	19	

1~50사이의 숫자 중에서 25개를 골라서 '5×5'크기의 빙고판을 만드는 예제이다. next()는 반환값이 Object타입이므로 형변환해서 원래의 타입으로 되돌려 놓아야 한다. 이 예제 역시 random()을 사용했기 때문에 실행할 때마다 다른 결과를 얻을 것이다.

그런데 몇번 실행해보면 같은 숫자가 비슷한 위치에 나온다는 사실을 발견할 수 있을 것이다. 앞서 언급한 것과 같이 HashSet은 저장된 순서를 보장하지 않고 자체적인 저장방식에 따라 순서가 결정되기 때문이다. 이 경우에는 HashSet보다 LinkedHashSet이 더 나은 선택이다.

▼ 예제 11-23/ch11/HashSetEx3.java

```
import java.util.*;  
class HashSetEx3 {  
    public static void main(String[] args) {  
        HashSet set = new HashSet();  
  
        set.add("abc");  
        set.add("abc");  
        set.add(new Person("David", 10));  
        set.add(new Person("David", 10));  
  
        System.out.println(set);  
    }  
  
    class Person {  
        String name;  
        int age;  
  
        Person(String name, int age) {  
            this.name = name;  
            this.age = age;  
        }  
  
        public String toString() {  
            return name + ":" + age;  
        }  
    }  
}
```

▼ 실행결과
[abc, David:10, David:10]

Person 클래스는 name과 age를 멤버 변수로 갖는다. 이름(name)과 나이(age)가 같으면 같은 사람으로 인식하도록 하려는 의도로 작성하였다. 하지만 실행 결과를 보면 두 인스턴스의 name과 age의 값이 같음에도 불구하고 서로 다른 것으로 인식하여 'David:10'이 두 번 출력되었다.

클래스의 작성 의도대로 이 두 인스턴스를 같은 것으로 인식하게 하려면 어떻게 해야 하는 걸까? 이에 대한 답은 다음 예제에 나와 있다.

▼ 예제 11-24/ch11/HashSetEx4.java

```
import java.util.*;  
class HashSetEx4 {  
    public static void main(String[] args) {  
        HashSet set = new HashSet();  
  
        set.add(new String("abc"));  
        set.add(new String("abc"));  
        set.add(new Person2("David", 10));  
        set.add(new Person2("David", 10));  
  
        System.out.println(set);  
    }  
}
```

```

class Person2 {
    String name;
    int age;

    Person2(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public boolean equals(Object obj) {
        if(obj instanceof Person2) {
            Person2 tmp = (Person2)obj;
            return name.equals(tmp.name) && age==tmp.age;
        }

        return false;
    }

    public int hashCode() {
        return (name+age).hashCode();
    }

    public String toString() {
        return name + ":" + age;
    }
}

```

▼ 실행결과 [abc, David:10]

HashSet의 add메서드는 새로운 요소를 추가하기 전에 기존에 저장된 요소와 같은 것인지 판별하기 위해 추가하려는 요소의 equals()와 hashCode()를 호출하기 때문에 equals()와 hashCode()를 목적에 맞게 오버라이딩해야 한다.

그래서 String클래스에서 같은 내용의 문자열에 대한 equals()의 호출결과가 true인 것과 같이, Person2클래스에서도 두 인스턴스의 name과 age가 서로 같으면 true를 반환하도록 equals()를 오버라이딩했다. 그리고 hashCode()는 String클래스의 hashCode()를 이용해서 구현했다. String클래스의 hashCode()는 잘 구현되어 있기 때문에 이를 활용하면 간단히 처리할 수 있다.

```

public int hashCode() {
    return (name+age).hashCode();
}

```

위의 코드를 JDK1.8부터 추가된 java.util.Objects클래스의 hash()를 이용해서 작성하면 아래와 같다. 이 메서드의 괄호 안에 클래스의 인스턴스 변수들을 넣으면 된다. 이전의 코드와 별반 다르지 않지만, 가능하면 아래의 코드를 쓰자.

```

public int hashCode() {
    return Objects.hash(name, age); // int hash(Object... values)
}

```

오버라이딩을 통해 작성된 hashCode()는 다음의 세 가지 조건을 만족 시켜야 한다.

1. 실행 중인 애플리케이션 내의 동일한 객체에 대해서 여러 번 hashCode()를 호출해도 동일한 int값을 반환해야한다. 하지만, 실행시마다 동일한 int값을 반환할 필요는 없다. (단, equals메서드의 구현에 사용된 멤버변수의 값이 바뀌지 않았다고 가정한다.)

예를 들어 Person2클래스의 equals메서드에 사용된 멤버변수 name과 age의 값이 바뀌지 않는 한, 하나의 Person2인스턴스에 대해 hashCode()를 여러 번 호출했을 때 항상 같은 int값을 얻어야 한다.

```
Person2 p = new Person2("David", 10);
int hashCode1 = p.hashCode();
int hashCode2 = p.hashCode();

p.age = 20;
int hashCode3 = p.hashCode();
```

위의 코드에서 hashCode1의 값과 hashCode2의 값은 항상 일치해야하지만 이 두 값이 매번 실행할 때마다 반드시 같은 값일 필요는 없다. hashCode3은 equals메서드에 사용 된 멤버변수 age를 변경한 후에 hashCode메서드를 호출한 결과이므로 hashCode1이나 hashCode2와 달라도 된다.

| 참고 | String클래스는 문자열의 내용으로 해시코드를 만들어 내기 때문에 내용이 같은 문자열에 대한 hashCode()호출은 항상 동일한 해시코드를 반환한다. 반면에 Object클래스는 객체의 주소로 해시코드를 만들어 내기 때문에 실행할 때마다 해시코드값이 달라질 수 있다.

2. equals메서드를 이용한 비교에 의해서 true를 얻은 두 객체에 대해 각각 hashCode()를 호출해서 얻은 결과는 반드시 같아야 한다.

인스턴스 p1과 p2에 대해서 equals메서드를 이용한 비교의 결과인 변수 b의 값이 true라면, hashCode1과 hashCode2의 값은 같아야 한다는 뜻이다.

```
Person2 p1 = new Person2("David", 10);
Person2 p2 = new Person2("David", 10);

boolean b = p1.equals(p2);

int hashCode1 = p1.hashCode();
int hashCode2 = p2.hashCode();
```

3. equals메서드를 호출했을 때 false를 반환하는 두 객체는 hashCode() 호출에 대해 같은 int값 을 반환하는 경우가 있어도 괜찮지만, 해싱(hashing)을 사용하는 컬렉션의 성능을 향상시키기 위해서는 다른 int값을 반환하는 것이 좋다.

위의 코드에서 변수 b의 값이 false일지라도 hashCode1과 hashCode2의 값이 같은 경우 발생하는 것을 허용한다. 하지만, 해시코드를 사용하는 Hashtable이나 HashMap과 같은 컬렉션의 성능을 높이기 위해서는 가능한 한 서로 다른 값을 반환하도록 hashCode()를 잘 작성해야 한다는 뜻이다.

록 해싱을 사용하는 Hashtable, HashMap과 같은 컬렉션의 검색속도가 떨어진다.
| 참고 | 해시코드와 해싱에 대한 보다 자세한 내용은 Hashtable에서 설명한 것이다.

두 객체에 대해 equals메서드를 호출한 결과가 true이면, 두 객체의 해시코드는 반드시 같아야하지만, 두 객체의 해시코드가 같다고 해서 equals메서드의 호출결과가 반드시 true이어야 하는 것은 아니다.

사용자정의 클래스를 작성할 때 equals메서드를 오버라이딩해야 한다면 hashCode()도 클래스의 작성의도에 맞게 오버라이딩하는 것이 원칙이지만, 경우에 따라 위의 예제에서와 같이 간단히 구현하거나 생략해도 별 문제가 되지 않으므로 hashCode()를 구현하는데 너무 부담 갖지 않았으면 한다.

마지막으로 예제 하나만 더 보고 HashSet에 대한 설명을 마무리하겠다.

▼ 예제 11-25/ch11/HashSetEx5.java

```
import java.util.*;
class HashSetEx5 {
    public static void main(String args[]) {
        HashSet setA = new HashSet();
        HashSet setB = new HashSet();
        HashSet setHab = new HashSet();
        HashSet setKyo = new HashSet();
        HashSet setCha = new HashSet();

        setA.add("1"); setA.add("2"); setA.add("3");
        setA.add("4"); setA.add("5");
        System.out.println("A = " + setA);

        setB.add("4"); setB.add("5"); setB.add("6");
        setB.add("7"); setB.add("8");
        System.out.println("B = " + setB);

        Iterator it = setB.iterator();
        while(it.hasNext()) {
            Object tmp = it.next();
            if(setA.contains(tmp))
                setKyo.add(tmp);
        }

        it = setA.iterator();
        while(it.hasNext()) {
            Object tmp = it.next();
            if(!setB.contains(tmp))
                setCha.add(tmp);
        }

        it = setA.iterator();
        while(it.hasNext())
            setHab.add(it.next());
        it = setB.iterator();
        while(it.hasNext())
            setHab.add(it.next());

        System.out.println("A ∩ B = " + setKyo); // 한글 ↵을 누르고 한자키
        System.out.println("A ∪ B = " + setHab); // 한글 ↵을 누르고 한자키
        System.out.println("A - B = " + setCha);
    }
}
```

▼ 실행결과

```
A = [1, 2, 3, 4, 5]
B = [4, 5, 6, 7, 8]
A ∩ B = [4, 5]
A ∪ B = [1, 2, 3, 4, 5, 6, 7, 8]
A - B = [1, 2, 3]
```

이 예제는 두 개의 HashSet에 저장된 객체들을 비교해서 합집합, 교집합, 차집합을 구하는 방법을 보여준다. 사실 Set은 중복을 허용하지 않으므로 HashSet의 메서드를 호출하는 것만으로도 간단하게 합집합(addAll), 교집합(retainAll), 차집합(removeAll)을 구할 수 있다. 그래도 직접 작성해 보는 것이 더 도움이 될 것 같아서 만들어 보았다. contains()와 add()만을 이용한 간단한 것이기 때문에 별도의 설명은 필요없을 것이다.

1.9 TreeSet

TreeSet은 이진 검색 트리(binary search tree)라는 자료구조의 형태로 데이터를 저장하는 컬렉션 클래스이다. 이진 검색 트리는 정렬, 검색, 범위검색(range search)에 높은 성능을 보이는 자료구조이며 TreeSet은 이진 검색 트리의 성능을 향상시킨 '레드-블랙 트리(Red-Black tree)'로 구현되어 있다.

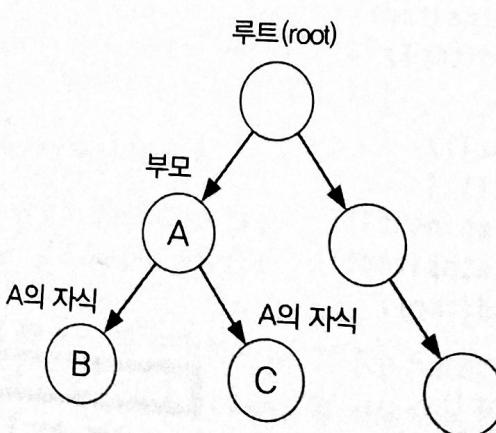
그리고 Set인터페이스를 구현했으므로 중복된 데이터의 저장을 허용하지 않으며 정렬된 위치에 저장하므로 저장순서를 유지하지도 않는다.

이진 트리(binary tree)는 링크드리스트처럼 여러 개의 노드(node)가 서로 연결된 구조로, 각 노드에 최대 2개의 노드를 연결할 수 있으며 '루트(root)'라고 불리는 하나의 노드에서부터 시작해서 계속 확장해 나갈 수 있다.

위 아래로 연결된 두 노드를 '부모-자식관계'에 있다고 하며 위의 노드를 부모 노드, 아래의 노드를 자식 노드라 한다. 부모-자식관계는 상대적인 것이며 하나의 부모 노드는 최대 두 개의 자식 노드와 연결될 수 있다.

아래의 그림에서 A는 B와 C의 부모 노드이고, B와 C는 A의 자식 노드이다.

| 참고 | 트리(tree)는 각 노드간의 연결된 모양이 나무와 같다고 해서 붙여진 이름이다.



▲ 그림 11-19 이진 트리(binary tree)의 예

이진 트리의 노드를 코드로 표현하면 다음과 같다.

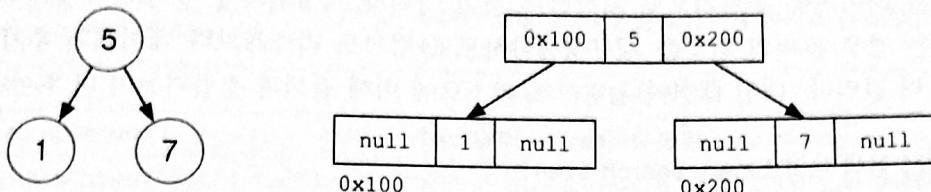
```

class TreeNode {
    TreeNode left;
    Object element;           // 왼쪽 자식노드
    TreeNode right;           // 객체를 저장하기 위한 참조변수
                            // 오른쪽 자식노드
}
  
```

데이터를 저장하기 위한 Object타입의 참조변수 하나와 두 개의 노드를 참조하기 위한 두 개의 참조변수를 선언했다.

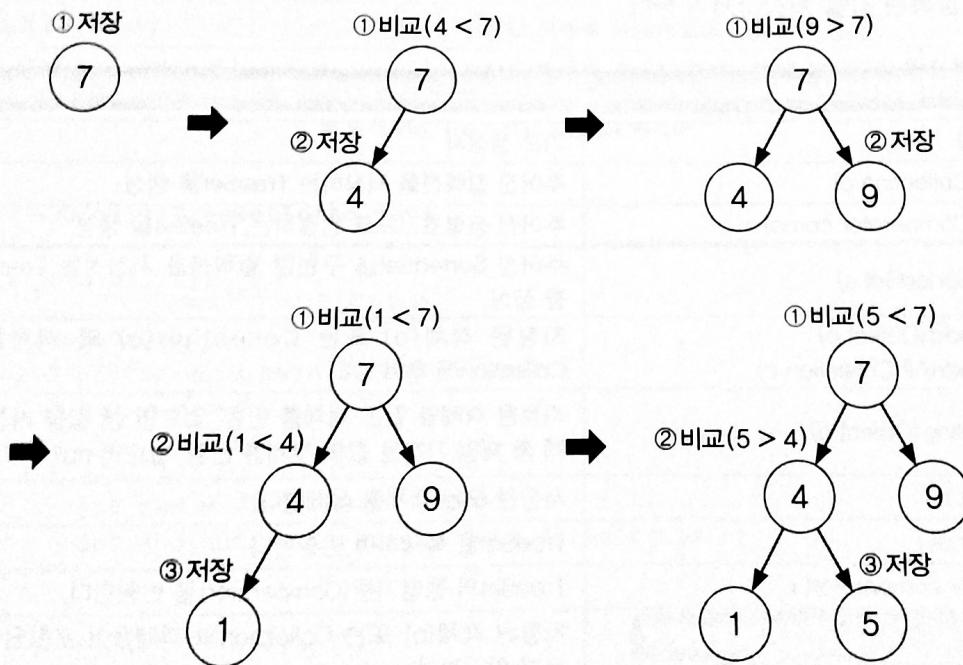
이진 검색 트리(binary search tree)는 부모노드의 왼쪽에는 부모노드의 값보다 작은 값의 자식노드를 오른쪽에는 큰 값의 자식노드를 저장하는 이진 트리이다.

예를 들어 데이터를 5, 1, 7의 순서로 저장한 이진 트리의 구조는 아래와 같이 표현할 수 있다. 실제로는 오른쪽 그림과 같이 표현해야하나 앞으로 간단히 왼쪽과 같이 표현하여 설명하겠다.



▲ 그림11-20 이진 검색 트리

예를 들어 이진검색트리에 7, 4, 9, 1, 5의 순서로 값을 저장한다고 가정하면 다음과 같은 순서로 진행된다.



▲ 그림11-21 이진 검색 트리의 저장과정

첫 번째로 저장되는 값은 루트가 되고, 두 번째 값은 트리의 루트부터 시작해서 값의 크기를 비교하면서 트리를 따라 내려간다. 작은 값은 왼쪽에 큰 값은 오른쪽에 저장한다. 이렇게 트리를 구성하면, 왼쪽 마지막 레벨이 제일 작은 값이 되고 오른쪽 마지막 레벨의 값이 제일 큰 값이 된다. 앞서 살펴본 것처럼, 컴퓨터는 알아서 값을 비교하지 못한다.

TreeSet에 저장되는 객체가 Comparable을 구현하던가 아니면, Comparator를 제공해서 두 객체를 비교할 방법을 알려줘야 한다. 그렇지 않으면, TreeSet에 객체를 저장할 때 예외가 발생한다.

원쪽 마지막 값에서부터 오른쪽 값까지 값을 '원쪽 노드 → 부모 노드 → 오른쪽 노드' 순으로 읽어오면 오름차순으로 정렬된 순서를 얻을 수 있다. TreeSet은 이처럼 정렬된 상태를 유지하기 때문에 단일 값 검색과 범위검색(range search), 예를 들면 3과 7사이의 범위에 있는 값을 검색이 매우 빠르다.

저장된 값의 개수에 비례해서 검색시간이 증가하긴 하지만 값의 개수가 10배 증가해도 특정 값을 찾는데 필요한 비교횟수가 3~4번만 증가할 정도로 검색효율이 뛰어난 자료구조이다.

트리는 데이터를 순차적으로 저장하는 것이 아니라 저장위치를 찾아서 저장해야하고, 삭제하는 경우 트리의 일부를 재구성해야하므로 링크드 리스트보다 데이터의 추가/삭제 시간은 더 걸린다. 대신 배열이나 링크드 리스트에 비해 검색과 정렬기능이 더 뛰어나다.

이진 검색 트리(binary search tree)는

- 모든 노드는 최대 두 개의 자식노드를 가질 수 있다.
- 원쪽 자식노드의 값은 부모노드의 값보다 작고 오른쪽자식노드의 값은 부모노드의 값보다 커야한다.
- 노드의 추가 삭제에 시간이 걸린다.(순차적으로 저장하지 않으므로)
- 검색(범위검색)과 정렬에 유리하다.
- 중복된 값을 저장하지 못한다.

생성자 또는 메서드	설명
<code>TreeSet()</code>	기본 생성자
<code>TreeSet(Collection c)</code>	주어진 컬렉션을 저장하는 TreeSet을 생성
<code>TreeSet(Comparator comp)</code>	주어진 정렬조건으로 정렬하는 TreeSet을 생성
<code>TreeSet(SortedSet s)</code>	주어진 SortedSet을 구현한 컬렉션을 저장하는 TreeSet을 생성
<code>boolean add(Object o)</code> <code>boolean addAll(Collection c)</code>	지정된 객체(o) 또는 Collection(c)의 객체들을 Collection에 추가
<code>Object ceiling(Object o)</code>	지정된 객체와 같은 객체를 반환. 없으면 큰 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
<code>void clear()</code>	저장된 모든 객체를 삭제한다.
<code>Object clone()</code>	TreeSet을 복제하여 반환한다.
<code>Comparator comparator()</code>	TreeSet의 정렬기준(Comparator)을 반환한다.
<code>boolean contains(Object o)</code> <code>boolean containsAll(Collection c)</code>	지정된 객체(o) 또는 Collection의 객체들이 포함되어 있는지 확인한다.
<code>NavigableSet descendingSet()</code>	TreeSet에 저장된 요소들을 역순으로 정렬해서 반환
<code>Object first()</code>	정렬된 순서에서 첫 번째 객체를 반환한다.
<code>Object floor(Object o)</code>	지정된 객체와 같은 객체를 반환. 없으면 작은 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
<code>SortedSet headSet(Object toElement)</code> <code>NavigableSet headSet(Object toElement, boolean inclusive)</code>	지정된 객체보다 작은 값의 객체들을 반환
<code>Object higher(Object o)</code>	지정된 객체보다 같은 값의 객체도 포함 inclusive가 true이면, 같은 값의 객체도 포함 지정된 객체보다 큰 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null

boolean isEmpty()	TreeSet이 비어있는지 확인한다.
Iterator iterator()	TreeSet의 Iterator를 반환한다.
Object last()	정렬된 순서에서 마지막 객체를 반환한다.
Object lower(Object o)	지정된 객체보다 작은 값을 가진 객체 중 제일 가까운 값의 객체를 반환. 없으면 null
Object pollFirst()	TreeSet의 첫번째 요소(제일 작은 값의 객체)를 반환.
Object pollLast()	TreeSet의 마지막 번째 요소(제일 큰 값의 객체)를 반환.
boolean remove(Object o)	지정된 객체를 삭제한다.
boolean retainAll(Collection c)	주어진 컬렉션과 공통된 요소만을 남기고 삭제한다.(교집합)
int size()	저장된 객체의 개수를 반환한다.
Spliterator spliterator()	TreeSet의 spliterator를 반환
SortedSet subSet(Object fromElement, Object toElement)	범위 검색(fromElement와 toElement사이)의 결과를 반환한다.(골 범위인 toElement는 범위에 포함되지 않음)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	범위 검색(fromElement와 toElement사이)의 결과를 반환한다.(fromInclusive가 true면 시작값이 포함되고, toInclusive가 true면 끝값이 포함된다.)
SortedSet tailSet(Object fromElement)	지정된 객체보다 큰 값의 객체들을 반환한다.
Object[] toArray()	저장된 객체를 객체배열로 반환한다.
Object[] toArray(Object[] a)	저장된 객체를 주어진 객체배열에 저장하여 반환한다.

▲ 표 11-16 TreeSet의 생성자와 메서드

▼ 예제 11-26/ch11/TreeSetLotto.java

```
import java.util.*;

class TreeSetLotto {
    public static void main(String[] args) {
        Set set = new TreeSet();

        for (int i = 0; set.size() < 6 ; i++) {
            int num = (int)(Math.random()*45) + 1;
            set.add(num); // set.add(new Integer(num));
        }

        System.out.println(set);
    }
}
```

▼ 실행결과
[5, 12, 24, 26, 33, 45]

이전의 예제 11-21 HashSetLotto.java를 TreeSet을 사용해서 바꾸었다. 이전 예제와는 달리 정렬하는 코드가 빠져 있는데, TreeSet은 저장할 때 이미 정렬하기 때문에 읽어올 때 따로 정렬할 필요가 없기 때문이다.

| 참고 | Math.random()을 이용했기 때문에 실행할 때마다 결과가 달라진다.

▼ 예제 11-27/ch11/TreeSetEx1.java

```

import java.util.*;
class TreeSetEx1 {
    public static void main(String[] args) {
        TreeSet set = new TreeSet();

        String from = "b";
        String to = "d";
        set.add("abc");      set.add("alien");      set.add("bat");
        set.add("car");      set.add("Car");       set.add("disc");
        set.add("dance");    set.add("dZZZZ");     set.add("dzzzz");
        set.add("elephant"); set.add("elevator"); set.add("fan");
        set.add("flower");

        System.out.println(set);
        System.out.println("range search : from " + from + " to " + to);
        System.out.println("result1 : " + set.subSet(from, to));
        System.out.println("result2 : " + set.subSet(from, to + "zzz"));
    }
}

```

▼ 실행결과

```

[Car, abc, alien, bat, car, dZZZZ, dance, disc, dzzzz, elephant, elevator,
fan, flower]
range search : from b to d
result1 : [bat, car]
result2 : [bat, car, dZZZZ, dance, disc]

```

subSet()을 이용해서 범위검색(range search)할 때 시작범위는 포함되지만 끝 범위는 포함되지 않으므로 result1에는 c로 시작하는 단어까지만 검색결과에 포함되어 있다.

만일 끝 범위인 d로 시작하는 단어까지 포함시키고자 한다면, 아래와 같이 끝 범위에 'zzz'와 같은 문자열을 붙이면 된다.

```
System.out.println("result2 : " + set.subSet(from, to + "zzz"));
```

d로 시작하는 단어 중에서 'dzzz' 다음에 오는 단어는 없을 것이기 때문에 d로 시작하는 모든 단어들이 포함될 것이다.

결과를 보면 'abc'보다 'Car'가 앞에 있고 'dZZZZ'가 'dance' 보다 앞에 정렬되어 있는 것을 알 수 있다. 대문자가 소문자보다 우선하기 때문에 대소문자가 섞여 있는 경우 의도한 결과는 다른 범위검색결과를 얻을 수 있다.

그래서 가능하면 대문자 또는 소문자로 통일해서 저장하는 것이 좋다.

참고 반드시 대소문자가 섞여 있어야 하거나 다른 방식으로 정렬해야하는 경우 Comparator를 이용하면 된다.

문자열의 경우 정렬순서는 문자의 코드값이 기준이 되므로, 오름차순 정렬의 경우 코드값의 크기가 작은 순서에서 큰 순서, 즉 공백, 숫자, 대문자, 소문자 순으로 정렬되고 내림차순의 경우 그 반대가 된다. 다음의 예제를 통해서 이를 확인하고 잘 기억해 두자.

▼ 예제 11-28/ch11/AsciiPrint.java

```
class AsciiPrint {
    public static void main(String[] args) {
        char ch = ' ';
        // 공백(' ') 이후의 문자들을 출력한다.
        for(int i=0; i < 95; i++)
            System.out.print(ch++);
    }
}
```

▼ 실행결과

```
!"#%&'()*+,.-./0123456789:;<=>?@ABC
DEFGHIJKLMNOPQRSTUVWXYZ[\]^_ abcdefghi
jk lmnopqrstuvwxyz{|}~
```

! 참고! 출력된 실행결과의 첫 번째 문자는 공백문자이다.

▼ 예제 11-29/ch11/TreeSetEx2.java

```
import java.util.*;
class TreeSetEx2 {
    public static void main(String[] args) {
        TreeSet set = new TreeSet();
        int[] score = {80, 95, 50, 35, 45, 65, 10, 100};

        for(int i=0; i < score.length; i++)
            set.add(new Integer(score[i]));

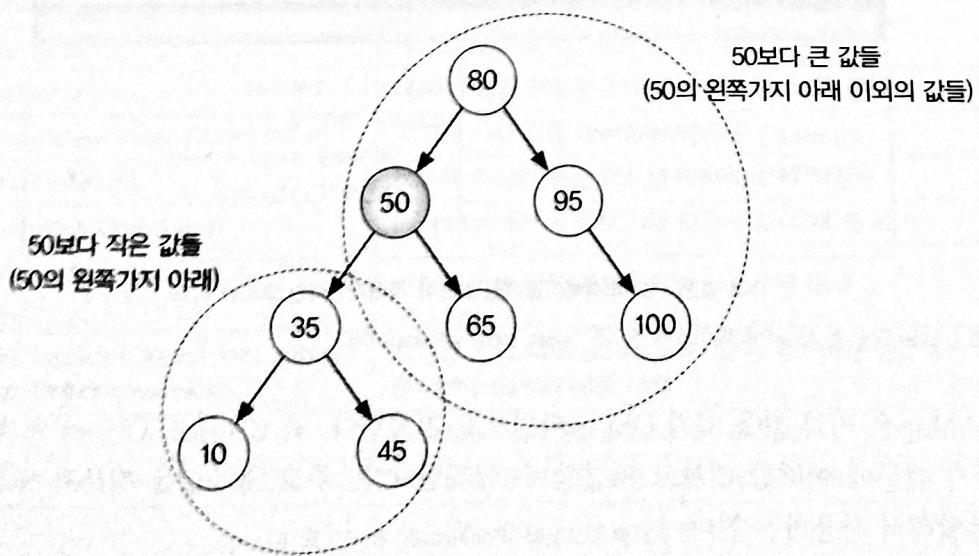
        System.out.println("50보다 작은 값 :" + set.headSet(new Integer(50)));
        System.out.println("50보다 큰 값 :" + set.tailSet(new Integer(50)));
    }
}
```

▼ 실행결과

```
50보다 작은 값 :[10, 35, 45]
50보다 큰 값 :[50, 65, 80, 95, 100]
```

headSet메서드와 tailSet메서드를 이용하면, TreeSet에 저장된 객체 중 지정된 기준 값보다 큰 값의 객체들과 작은 값의 객체들을 얻을 수 있다.

예제에 사용된 값들로 이진 검색 트리를 구성해 보면 다음 그림과 같다.



▲ 그림 11-22 예제 11-29에 사용된 데이터로 구성한 이진 검색 트리

위의 그림을 보면 50이 저장된 노드의 왼쪽노드와 그 아래 연결된 모든 노드의 값은 50보다 작고, 나머지 다른 노드의 값들은 50보다 같거나 크다는 것을 알 수 있다.

▼ 예제 11-28/ch11/AsciiPrint.java

```
class Asciprint{
    public static void main(String[] args) {
        char ch = ' ';
        // 공백(' ') 이후의 문자들을 출력한다.
        for(int i=0; i < 95; i++)
            System.out.print(ch++);
    }
}
```

▼ 실행결과

```
!"#$%&'()*+,-./0123456789:;<=>?@ABC
DEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
jk lmnopqrstuvwxyz{|}|~
```

| 참고 | 출력된 실행결과의 첫 번째 문자는 공백문자이다.

▼ 예제 11-29/ch11/TreeSetEx2.java

```
import java.util.*;
class TreeSetEx2 {
    public static void main(String[] args) {
        TreeSet set = new TreeSet();
        int[] score = {80, 95, 50, 35, 45, 65, 10, 100};

        for(int i=0; i < score.length; i++)
            set.add(new Integer(score[i]));

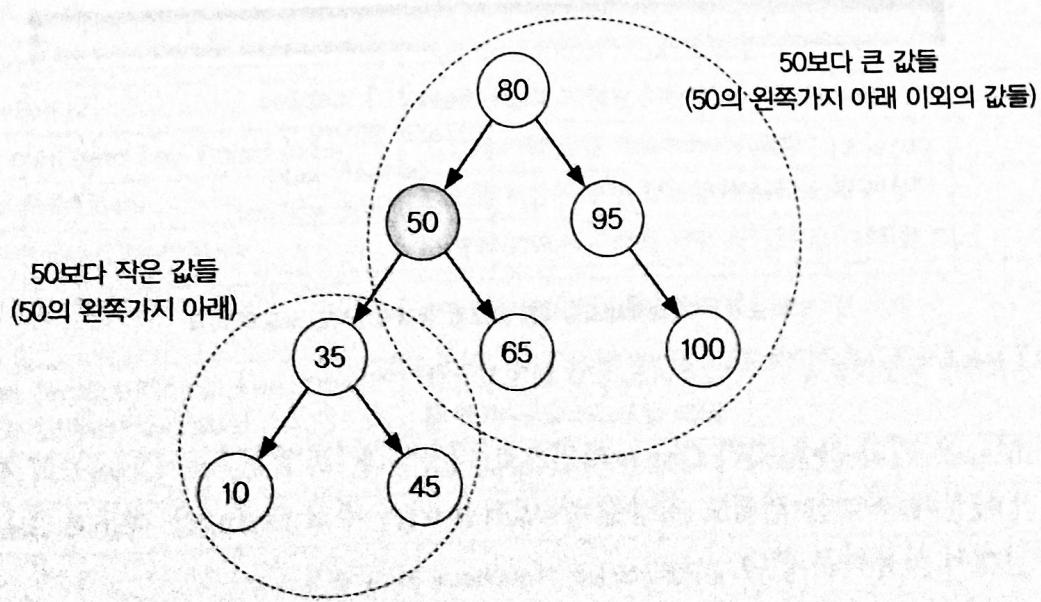
        System.out.println("50보다 작은 값 :" + set.headSet(new Integer(50)));
        System.out.println("50보다 큰 값 :" + set.tailSet(new Integer(50)));
    }
}
```

▼ 실행결과

```
50보다 작은 값 :[10, 35, 45]
50보다 큰 값 :[50, 65, 80, 95, 100]
```

headSet메서드와 tailSet메서드를 이용하면, TreeSet에 저장된 객체 중 지정된 기준 값보다 큰 값의 객체들과 작은 값의 객체들을 얻을 수 있다.

예제에 사용된 값들로 이진 검색 트리를 구성해 보면 다음과 같다.



▲ 그림 11-22 예제 11-29에 사용된 데이터로 구성한 이진 검색 트리

위의 그림을 보면 50이 저장된 노드의 왼쪽노드와 그 아래 연결된 모든 노드의 값은 50보다 작고, 나머지 다른 노드의 값들은 50보다 같거나 크다는 것을 알 수 있다.

▼ 예제 11-28/ch11/AsciiPrint.java

```
class AsciiPrint{
    public static void main(String[] args) {
        char ch = ' ';
        // 공백(' ') 이후의 문자들을 출력한다.
        for(int i=0; i < 95; i++)
            System.out.print(ch++);
    }
}
```

▼ 실행결과

```
!"#$%&'()*+,-./0123456789:;<=>?@ABC
DEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnoqrstuvwxyz{|}~
```

| 참고 | 출력된 실행결과의 첫 번째 문자는 공백문자이다.

▼ 예제 11-29/ch11/TreeSetEx2.java

```
import java.util.*;
class TreeSetEx2 {
    public static void main(String[] args) {
        TreeSet set = new TreeSet();
        int[] score = {80, 95, 50, 35, 45, 65, 10, 100};

        for(int i=0; i < score.length; i++)
            set.add(new Integer(score[i]));

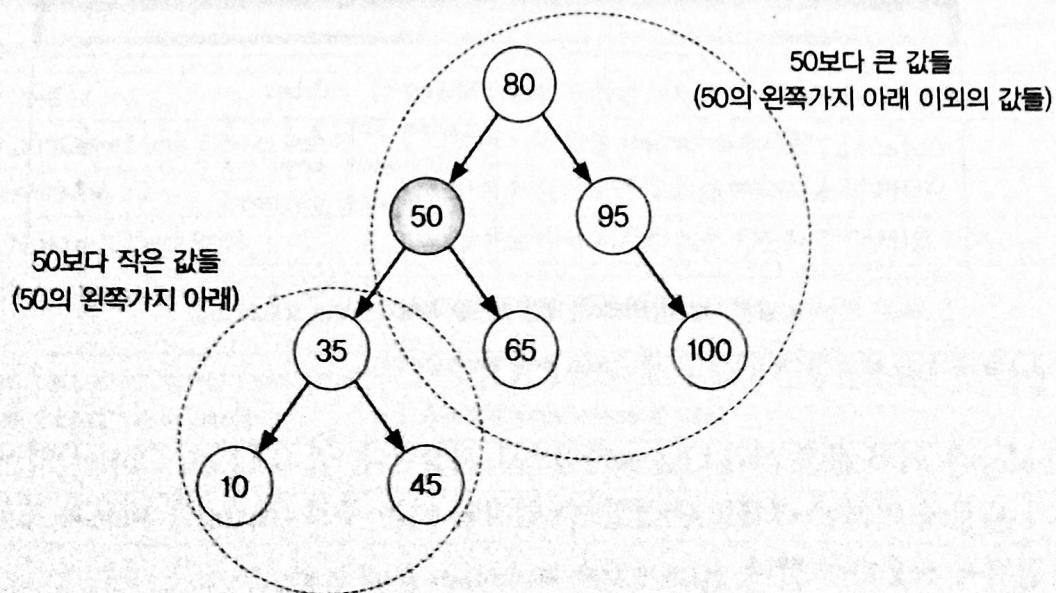
        System.out.println("50보다 작은 값 :" + set.headSet(new Integer(50)));
        System.out.println("50보다 큰 값 :" + set.tailSet(new Integer(50)));
    }
}
```

▼ 실행결과

```
50보다 작은 값 :[10, 35, 45]
50보다 큰 값 :[50, 65, 80, 95, 100]
```

headSet메서드와 tailSet메서드를 이용하면, TreeSet에 저장된 객체 중 지정된 기준 값보다 큰 값의 객체들과 작은 값의 객체들을 얻을 수 있다.

예제에 사용된 값들로 이진 검색 트리를 구성해 보면 다음 그림과 같다.



▲ 그림 11-22 예제11-29에 사용된 데이터로 구성한 이진 검색 트리

위의 그림을 보면 50이 저장된 노드의 왼쪽노드와 그 아래 연결된 모든 노드의 값은 50보다 작고, 나머지 다른 노드의 값들은 50보다 같거나 크다는 것을 알 수 있다.

1.10 HashMap과 Hashtable

Hashtable과 HashMap의 관계는 Vector와 ArrayList의 관계와 같아서 Hashtable보다는 새로운 버전인 HashMap을 사용할 것을 권한다. 여기서는 HashMap 대해서만 설명하도록 하겠다.

HashMap은 Map을 구현했으므로 앞에서 살펴본 Map의 특징, 키(key)와 값(value)을 묶어서 하나의 데이터(entry)로 저장한다는 특징을 갖는다. 그리고 해싱(hashing)을 사용하기 때문에 많은 양의 데이터를 검색하는데 있어서 뛰어난 성능을 보인다.

HashMap이 데이터를 어떻게 저장하는지 확인하기 위해 실제 소스의 일부를 발췌하였다.

```
public class HashMap extends AbstractMap implements Map, Cloneable,  
    Serializable  
{  
    transient Entry[] table;  
    ...  
    static class Entry implements Map.Entry {  
        final Object key;  

```

HashMap은 Entry라는 내부 클래스를 정의하고, 다시 Entry 타입의 배열을 선언하고 있다. 키(key)와 값(value)은 별개의 값이 아니라 서로 관련된 값이기 때문에 각각의 배열로 선언하기보다는 하나의 클래스로 정의해서 하나의 배열로 다루는 것이 데이터의 무결성(integrity)적인 측면에서 더 바람직하기 때문이다.

비객체지향적인 코드	객체지향적인 코드
Object[] key; Object[] value;	Entry[] table; class Entry { Object key; Object value; }

▲ 표 11-17 비객체지향적인 코드와 객체지향적인 코드의 비교

| 참고 | Map.Entry는 Map 인터페이스에 정의된 'static inner interface'이다.

HashMap은 키와 값을 각각 Object 타입으로 저장한다. 즉 (Object, Object)의 형태로 저장하기 때문에 어떠한 객체도 저장할 수 있지만 키는 주로 String을 대문자 또는 소문자로 통일해서 사용하곤 한다.

키(key)

컬렉션 내의 키(key) 중에서 유일해야 한다.

값(value)

키(key)와 달리 데이터의 중복을 허용한다.

키는 저장된 값을 찾는데 사용되는 것이기 때문에 컬렉션 내에서 유일(unique)해야 한다. 즉, HashMap에 저장된 데이터를 하나의 키로 검색했을 때 결과가 단 하나이어야 함을 뜻한다. 만일 하나의 키에 대해 여러 검색결과 값을 얻는다면 원하는 값이 어떤 것인지 알 수 없기 때문이다.

예를 들어 사용자ID가 키(key)로, 비밀번호가 값(value)으로 연결되어 저장된 데이터집합이 있다고 가정하자. 로그인 시에 비밀번호를 확인하기 위해서 입력된 사용자ID에 대한 비밀번호를 검색했을 때, 단 하나의 결과를 얻어야만 올바른 비밀번호를 입력했는지 확인이 가능할 것이다. 만일 하나의 사용자ID에 대해서 두 개 이상의 비밀번호를 얻는다면 어떤 비밀번호가 맞는 것인지 알 수 없다.

생성자 / 메서드	설명
HashMap()	HashMap 객체를 생성
HashMap(int initialCapacity)	지정된 값을 초기용량으로 하는 HashMap 객체를 생성
HashMap(int initialCapacity, float loadFactor)	지정된 초기용량과 load factor의 HashMap 객체를 생성
HashMap(Map m)	지정된 Map의 모든 요소를 포함하는 HashMap을 생성
void clear()	HashMap에 저장된 모든 객체를 제거
Object clone()	현재 HashMap을 복제해서 반환
boolean containsKey(Object key)	HashMap에 지정된 키(key)가 포함되어있는지 알려준다.(포함되어 있으면 true)
boolean containsValue(Object value)	HashMap에 지정된 값(value)가 포함되어있는지 알려준다.(포함되어 있으면 true)
Set entrySet()	HashMap에 저장된 키와 값을 엔트리(키와 값의 결합)의 형태로 Set에 저장해서 반환
Object get(Object key)	지정된 키(key)의 값(객체)을 반환. 못찾으면 null 반환
Object getOrDefault(Object key, Object defaultValue)	지정된 키(key)의 값(객체)을 반환한다. 키를 못찾으면, 기본값(defaultValue)로 지정된 객체를 반환
boolean isEmpty()	HashMap이 비어있는지 알려준다.
Set keySet()	HashMap에 저장된 모든 키가 저장된 Set을 반환
Object put(Object key, Object value)	지정된 키와 값을 HashMap에 저장
void putAll(Map m)	Map에 저장된 모든 요소를 HashMap에 저장
Object remove(Object key)	HashMap에서 지정된 키로 저장된 값(객체)을 제거
Object replace(Object key, Object value)	지정된 키의 값을 지정된 객체(value)로 대체
boolean replace(Object key, Object oldValue, Object newValue)	지정된 키와 객체(oldValue)가 모두 일치하는 경우에만 새로운 객체(newValue)로 대체
int size()	HashMap에 저장된 요소의 개수를 반환
Collection values()	HashMap에 저장된 모든 값을 컬렉션의 형태로 반환

▲ 표 11-18 HashMap의 생성자와 메서드

† 참고 1 JDK1.8부터 새로 추가된 메서드 중 람다와 스트림에 관련된 것들은 표 11-18에서 제외됨.

▼ 예제 11-30/ch11/HashMapEx1.java

```

import java.util.*;
class HashMapEx1 {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put("myId", "1234");
        map.put("asdf", "1111");
        map.put("asdf", "1234");
        Scanner s = new Scanner(System.in); // 키면으로부터 리인딩위로 입력받는다.

        while(true) {
            System.out.println("id와 password를 입력해주세요.");
            System.out.print("id :");
            String id = s.nextLine().trim();
            System.out.print("password :");
            String password = s.nextLine().trim();
            System.out.println();

            if(!map.containsKey(id)) {
                System.out.println("입력하신 id는 존재하지 않습니다."
                    + " 다시 입력해주세요.");
                continue;
            } else {
                if(!(map.get(id)).equals(password)) {
                    System.out.println("비밀번호가 일치하지 않습니다. 다시 입력해주세요.");
                } else {
                    System.out.println("id와 비밀번호가 일치합니다.");
                    break;
                }
            }
        } // while
    } // main의 끝
}

```

▼ 실행결과

```

c:\jdk1.8\work\ch11>java HashMapEx1
id와 password를 입력해주세요.
id :asdf
password :1111
비밀번호가 일치하지 않습니다. 다시 입력해주세요.
id와 password를 입력해주세요.
id :asdf
password :1234
id와 비밀번호가 일치합니다.
c:\jdk1.8\work\ch11>

```

HashMap을 생성하고 사용자ID와 비밀번호를 키와 값의 쌍(pair)으로 저장한 다음, 입력된 사용자ID를 키로 HashMap에서 검색해서 얻은 값(비밀번호)을 입력된 비밀번호와 비교하는 예제이다.

```
HashMap map = new HashMap();
map.put("myId", "1234");
map.put("asdf", "1111");
map.put("asdf", "1234");
```

위의 코드는 HashMap을 생성하고 데이터를 저장하는 부분인데 이 코드가 실행되고 나면 HashMap에는 아래와 같은 형태로 데이터가 저장된다.

키(key)	값(value)
myId	1234
asdf	1234

3개의 데이터 쌍을 저장했지만 실제로는 2개 밖에 저장되지 않은 이유는 중복된 키가 있기 때문이다. 세 번째로 저장한 데이터의 키인 'asdf'는 이미 존재하기 때문에 새로 추가되는 대신 기존의 값을 덮어썼다. 그래서 키 'asdf'에 연결된 값은 '1234'가 된다.

Map은 값은 중복을 허용하지만 키는 중복을 허용하지 않기 때문에 저장하려는 두 데이터 중에서 어느 쪽을 키로 할 것인지를 잘 결정해야한다.

| 참고 | Hashtable은 키(key)나 값(value)으로 null을 허용하지 않지만, HashMap은 허용한다. 그래서 'map.put(null, null);'이나 'map.get(null);'과 같이 할 수 있다.

▼ 예제 11-31/ch11/HashMapEx2.java

```
import java.util.*;

class HashMapEx2 {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put("김자바", new Integer(90));
        map.put("김자바", new Integer(100));
        map.put("이자바", new Integer(100));
        map.put("강자바", new Integer(80));
        map.put("안자바", new Integer(90));

        Set set = map.entrySet();
        Iterator it = set.iterator();

        while(it.hasNext()) {
            Map.Entry e = (Map.Entry)it.next();
            System.out.println("이름 : " + e.getKey()
                + ", 점수 : " + e.getValue());
        }

        set = map.keySet();
        System.out.println("참가자 명단 : " + set);

        Collection values = map.values();
        it = values.iterator();

        int total = 0;
```

```
HashMap map = new HashMap();
map.put("myId", "1234");
map.put("asdf", "1111");
map.put("asdf", "1234");
```

위의 코드는 HashMap을 생성하고 데이터를 저장하는 부분인데 이 코드가 실행되고 나면 HashMap에는 아래와 같은 형태로 데이터가 저장된다.

키(key)	값(value)
myId	1234
asdf	1234

3개의 데이터 쌍을 저장했지만 실제로는 2개 밖에 저장되지 않은 이유는 중복된 키가 있기 때문이다. 세 번째로 저장한 데이터의 키인 'asdf'는 이미 존재하기 때문에 새로 추가되는 대신 기존의 값을 덮어썼다. 그래서 키 'asdf'에 연결된 값은 '1234'가 된다.

Map은 값은 중복을 허용하지만 키는 중복을 허용하지 않기 때문에 저장하려는 두 데이터 중에서 어느 쪽을 키로 할 것인지를 잘 결정해야한다.

| 참고 | Hashtable은 키(key)나 값(value)으로 null을 허용하지 않지만, HashMap은 허용한다. 그래서 'map.put(null, null);'이나 'map.get(null);'과 같이 할 수 있다.

▼ 예제 11-31/ch11/HashMapEx2.java

```
import java.util.*;

class HashMapEx2 {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put("김자바", new Integer(90));
        map.put("김자바", new Integer(100));
        map.put("이자바", new Integer(100));
        map.put("강자바", new Integer(80));
        map.put("안자바", new Integer(90));

        Set set = map.entrySet();
        Iterator it = set.iterator();

        while(it.hasNext()) {
            Map.Entry e = (Map.Entry)it.next();
            System.out.println("이름 : " + e.getKey()
                + ", 점수 : " + e.getValue());
        }

        set = map.keySet();
        System.out.println("참가자 명단 : " + set);

        Collection values = map.values();
        it = values.iterator();

        int total = 0;
```

```

        while(it.hasNext()) {
            Integer i = (Integer)it.next();
            total += i.intValue();
        }

        System.out.println("총점 : " + total);
        System.out.println("평균 : " + (float)total/set.size());
        System.out.println("최고점수 : " + Collections.max(values));
        System.out.println("최저점수 : " + Collections.min(values));
    }
}

```

▼ 실행결과

```

이름 : 인자바, 점수 : 90
이름 : 김자비, 점수 : 100
이름 : 김자바, 점수 : 80
이름 : 이자바, 점수 : 100
참가자 명단 : [인자바, 김자바, 강자바, 이자바]
총점 : 370
평균 : 92.5
최고점수 : 100
최저점수 : 80

```

HashMap의 기본적인 메서드를 이용해서 데이터를 저장하고 읽어오는 예제이다. entrySet()을 이용해서 키와 값을 함께 읽어 올 수도 있고 keySet()이나 values()를 이용해서 키와 값을 따로 읽어 올 수 있다. 간단한 예제이니 더 이상의 설명은 필요 없을 것 같다.

▼ 예제 11-32/ch11/HashMapEx3.java

```

import java.util.*;

class HashMapEx3 {
    static HashMap phoneBook = new HashMap();

    public static void main(String[] args) {
        addPhoneNo("친구", "이자바", "010-111-1111");
        addPhoneNo("친구", "김자바", "010-222-2222");
        addPhoneNo("친구", "김자바", "010-333-3333");
        addPhoneNo("회사", "김대리", "010-444-4444");
        addPhoneNo("회사", "김대리", "010-555-5555");
        addPhoneNo("회사", "박대리", "010-666-6666");
        addPhoneNo("회사", "이과장", "010-777-7777");
        addPhoneNo("세탁", "010-888-8888");

        printList();
    } // main

    // 그룹에 전화번호를 추가하는 메서드
    static void addPhoneNo(String groupName, String name, String tel) {
        addGroup(groupName);
        HashMap group = (HashMap)phoneBook.get(groupName);
        group.put(tel, name); // 이름은 중복될 수 있으니 전화번호를 key로 저장한다.
    }
}

```

```

// 그룹을 추가하는 메서드
static void addGroup(String groupName) {
    if(!phoneBook.containsKey(groupName))
        phoneBook.put(groupName, new HashMap());
}

static void addPhoneNo(String name, String tel) {
    addPhoneNo("기타", name, tel);
}

// 전화번호부 전체를 출력하는 메서드
static void printList() {
    Set set = phoneBook.entrySet();
    Iterator it = set.iterator();

    while(it.hasNext()) {
        Map.Entry e = (Map.Entry)it.next();

        Set subSet = ((HashMap)e.getValue()).entrySet();
        Iterator subIt = subSet.iterator();

        System.out.println(" * "+e.getKey()+"["+subSet.size()+"]");
        while(subIt.hasNext()) {
            Map.Entry subE = (Map.Entry)subIt.next();
            String telNo = (String)subE.getKey();
            String name = (String)subE.getValue();
            System.out.println(name + " " + telNo );
        }
        System.out.println();
    }
} // printList()
} // class

```

▼ 실행결과

* 기타[1]
세탁 010-888-8888

* 친구[3]
이자바 010-111-1111
김자바 010-222-2222
김자바 010-333-3333

* 회사[4]
이과장 010-777-7777
김대리 010-444-4444
김대리 010-555-5555
박대리 010-666-6666

HashMap은 데이터를 키와 값을 모두 Object타입으로 저장하기 때문에 HashMap의 값(value)으로 HashMap을 다시 저장할 수 있다. 이렇게 함으로써 하나의 키에 다시 복수의 데이터를 저장할 수 있다.

먼저 전화번호를 저장할 그룹을 만들고 그룹 안에 다시 이름과 전화번호를 저장하도록 했다. 이때 이름대신 전화번호를 키로 사용했다는 것을 확인하자. 이름은 동명이인이 있을 수 있지만 전화번호는 유일하기 때문이다.

|참고| 배열의 배열인 이차원 배열과 비교해보면 이해하기 쉬울 것이다.

▼ 예제 11-33/ch11/HashMapEx4.java

```
import java.util.*;  
  
class HashMapEx4 {  
    public static void main(String[] args) {  
        String[] data = { "A", "K", "A", "K", "D", "K", "A", "K", "Z", "D" };  
        HashMap map = new HashMap();  
        for(int i=0; i < data.length; i++) {  
            if(map.containsKey(data[i])) {  
                Integer value = (Integer)map.get(data[i]);  
                map.put(data[i], new Integer(value.intValue() + 1));  
            } else {  
                map.put(data[i], new Integer(1));  
            }  
        }  
        Iterator it = map.entrySet().iterator();  
        while(it.hasNext()) {  
            Map.Entry entry = (Map.Entry)it.next();  
            int value = ((Integer)entry.getValue()).intValue();  
            System.out.println(entry.getKey() + " : "  
                + printBar('#', value) + " " + value );  
        }  
    } // main  
  
    public static String printBar(char ch, int value) {  
        char[] bar = new char[value];  
        for(int i=0; i < bar.length; i++)  
            bar[i] = ch;  
        return new String(bar); // String(char[] chArr)  
    }  
}
```

▼ 실행결과

A : #### 3
D : ## 2
Z : # 1
K : ##### 6

문자열 배열에 담긴 문자열을 하나씩 읽어서 HashMap에 키로 저장하고 값으로 1을 저장한다. HashMap에 같은 문자열이 키로 저장되어 있는지 containsKey()로 확인하여 이미 저장되어 있는 문자열이면 값을 1증가시킨다.

그리고 그 결과를 printBar()를 이용해서 그래프로 표현했다. 이렇게 하면 문자열 배열에 담긴 문자열들의 빈도수를 구할 수 있다.

한정된 범위 내에 있는 순차적인 값들의 빈도수는 배열을 이용하지만, 이처럼 한정되지 않은 범위의 비순차적인 값들의 빈도수는 HashMap을 이용해서 구할 수 있다.

| 참고 | 결과를 통해 HashMap과 같이 해싱을 구현한 컬렉션 클래스들은 저장순서를 유지하지 않는다는 사실을 다시 한 번 확인하자.

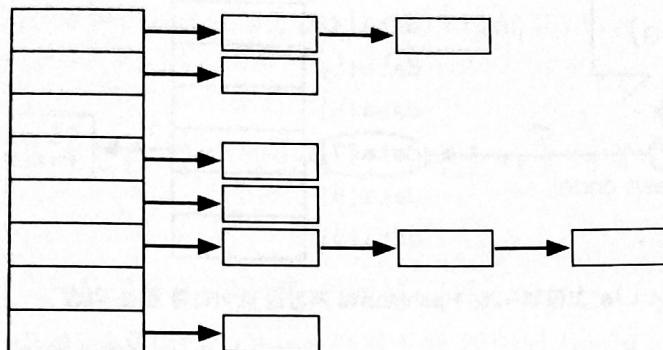
HashMap대한 설명은 이쯤에서 마무리하기로 하고 HashMap과 HashSet 등에 사용되는 해싱(hashing)에 대해서 알아보도록 하자.

해싱과 해시함수

해싱이란 해시함수(hash function)를 이용해서 데이터를 해시테이블(hash table)에 저장하고 검색하는 기법을 말한다. 해시함수는 데이터가 저장되어 있는 곳을 알려 주기 때문에 다양한 데이터 중에서도 원하는 데이터를 빠르게 찾을 수 있다.

해싱을 구현한 컬렉션 클래스로는 HashSet, HashMap, Hashtable 등이 있다. Hashtable은 컬렉션 프레임워크로 도입되면서 HashMap으로 대체되었으나 이전 소스와의 호환성 문제로 남겨 두고 있다. 가능하면 Hashtable 대신 HashMap을 사용하도록 하자.

해싱에서 사용하는 자료구조는 다음과 같이 배열과 링크드 리스트의 조합으로 되어 있다.



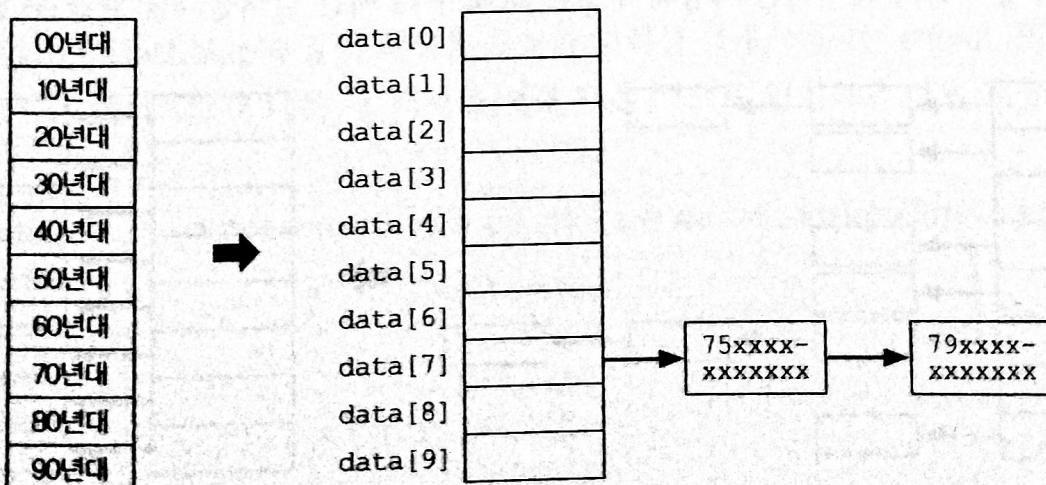
▲ 그림 11-23 배열과 링크드 리스트의 조합

저장할 데이터의 키를 해시함수에 넣으면 배열의 한 요소를 얻게 되고, 다시 그 곳에 연결되어 있는 링크드 리스트에 저장하게 된다.

이해를 돋기 위해 실생활에 비유한 예를 하나 들어보겠다. 한 간호사가 많은 환자들의 데이터 중에서, 원하는 환자의 데이터를 쉽게 찾을 수 있는 방법이 없을까를 고민하다가 주민등록번호의 맨 앞자리인 생년을 기준으로 데이터를 분류해서 10개의 서랍(배열)에 나눠 담는 방법을 생각해냈다. 예를 들면 71년생, 72년생과 같은 70년대생 환자들의 데이터는 같은 서랍에 저장된다.

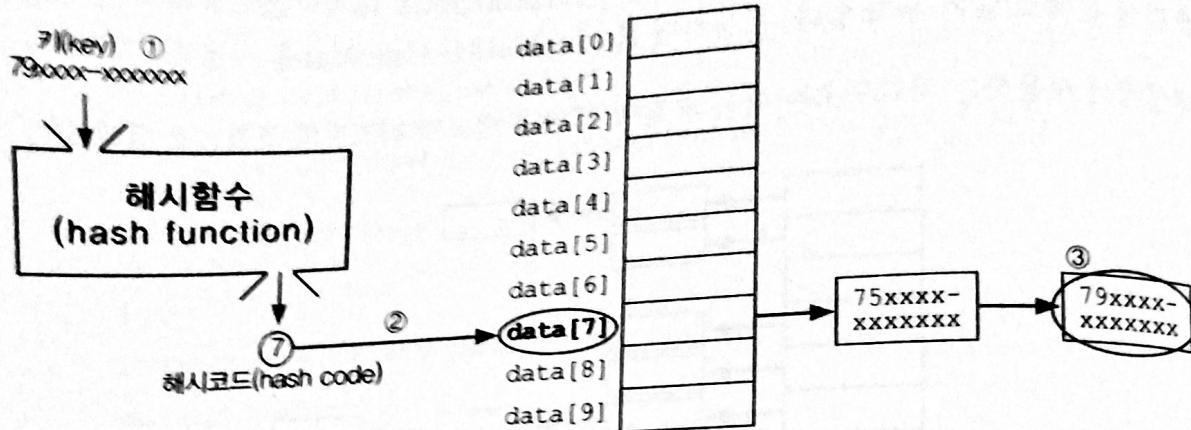
이렇게 분류해서 저장해두면 환자의 주민번호로 태어난 년대를 계산해서 어느 서랍에서 찾아야 할지를 쉽게 알 수 있다.

| 참고 | 동명이인이 있을 수 있기 때문에 이름보다는 주민등록번호를 키로 사용한다.



▲ 그림 11-24 HashMap에 저장된 데이터를 찾는 과정

여기서 서랍은 해싱에 사용되는 자료구조 중 배열의 각 요소를 의미하며, 배열의 각 요소에는 링크드 리스트가 저장되어 있어서 실제 데이터는 링크드 리스트에 담겨지게 된다. 아래 그림은 79년생 환자의 주민번호를 키로 해시함수를 통해 7이라는 해시코드를 얻은 다음, 배열의 7번째 요소에 저장된 링크드 리스트에서 원하는 데이터를 검색하는 과정을 표현한 것이다.



▲ 그림 11-25 HashMap에 저장된 데이터를 찾는 과정

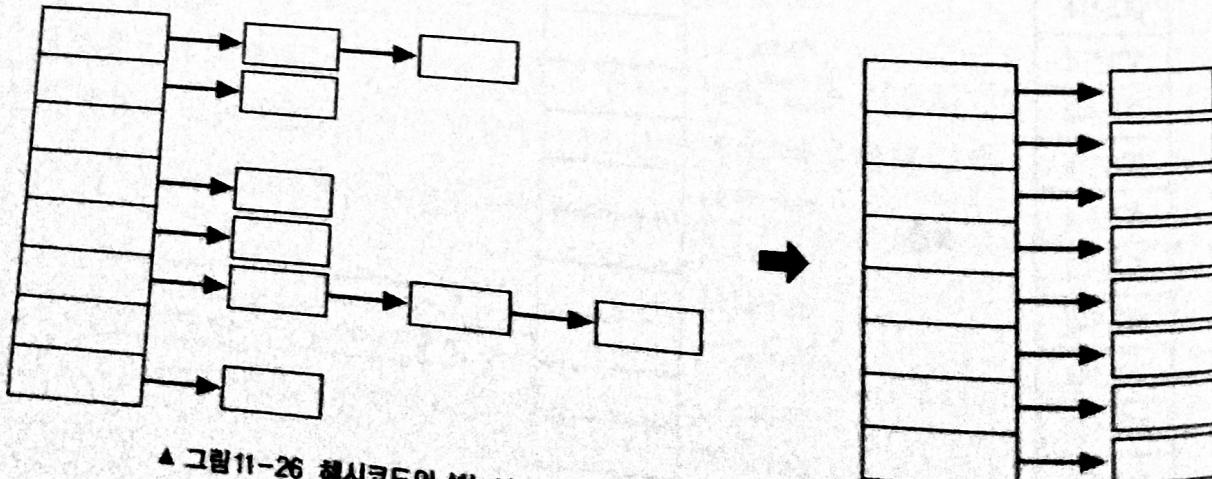
- ① 검색하고자 하는 값의 키로 해시함수를 호출한다.
- ② 해시함수의 계산결과(해시코드)로 해당 값이 저장되어 있는 링크드 리스트를 찾는다.
- ③ 링크드 리스트에서 검색한 키와 일치하는 데이터를 찾는다.

이미 배운 바와 같이 링크드 리스트는 검색에 불리한 자료구조이기 때문에 링크드 리스트의 크기가 커질수록 검색속도가 떨어지게 된다. 이는 하나의 서랍에 데이터의 수가 많을수록 검색에 시간이 더 걸리는 것과 같다.

반면에 배열은 배열의 크기가 커져도, 원하는 요소가 몇 번째에 있는 지만 알면 아래의 공식에 의해서 빠르게 원하는 값을 찾을 수 있다.

$$\text{배열의 인덱스가 } n \text{인 요소의 주소} = \text{배열의 시작주소} + \text{type의 size} * n$$

그래서 실생활과는 맞지 않지만, 하나의 서랍에 많은 데이터가 저장되어 있는 형태보다는 많은 서랍에 하나의 데이터만 저장되어 있는 형태가 더 빠른 검색결과를 얻을 수 있다.



▲ 그림 11-26 해시코드의 성능이 좋지 않은 경우(연결리스트)와 서버의 구조

하나의 링크드 리스트(서랍)에 최소한의 데이터만 저장되려면, 저장될 데이터의 크기를 고려해서 HashMap의 크기를 적절하게 지정해주어야 하고, 해시함수가 서로 다른 키(주민번호)에 대해서 중복된 해시코드(서랍위치)의 반환을 최소화해야 한다. 그래야 HashMap에서 빠른 검색시간을 얻을 수 있다.

그래서 해싱을 구현하는 과정에서 제일 중요한 것은 해시함수의 알고리즘이며, 이 예에서 사용된 해시함수의 알고리즘은 주어진 키(주민번호)의 첫 번째 문자를 뽑아서 정수로 반환하기만 하면 되므로 아래와 같이 코드로 표현할 수 있다.

```
int hashFunction(String key) {  
    return Integer.parseInt(key.substring(0,1));  
}
```

알고리즘이 간단한 만큼 성능은 좋지 않아서 서로 다른 키에 대해서 중복된 해시코드를 반환하는 경우가 많다.

실제로는 HashMap과 같이 해싱을 구현한 컬렉션 클래스에서는 Object클래스에 정의된 hashCode()를 해시함수로 사용한다. Object클래스에 정의된 hashCode()는 객체의 주소를 이용하는 알고리즘으로 해시코드를 만들어 내기 때문에 모든 객체에 대해 hashCode()를 호출한 결과가 서로 유일한 훌륭한 방법이다.

String클래스의 경우 Object로부터 상속받은 hashCode()를 오버라이딩해서 문자열의 내용으로 해시코드를 만들어 낸다. 그래서 서로 다른 String인스턴스일지라도 같은 내용의 문자열을 가졌다면 hashCode()를 호출하면 같은 해시코드를 얻는다.

HashSet에서 이미 설명했던 것과 같이 서로 다른 두 객체에 대해 equals()로 비교한 결과가 true인 동시에 hashCode()의 반환값이 같아야 같은 객체로 인식한다. HashMap에서도 같은 방법으로 객체를 구별하며, 이미 존재하는 키에 대한 값을 저장하면 기존의 값을 새로운 값으로 덮어쓴다.

그래서 새로운 클래스를 정의할 때 equals()를 재정의오버라이딩해야 한다면 hashCode()도 같이 재정의해서 equals()의 결과가 true인 두 객체의 해시코드hashCode()의 결과값이 항상 같도록 해주어야 한다.

그렇지 않으면 HashMap과 같이 해싱을 구현한 컬렉션 클래스에서는 equal()의 호출 결과가 true지만 해시코드가 다른 두 객체를 서로 다른 것으로 인식하고 따로 저장할 것이다.

| 참고 | equals()로 비교한 결과가 false이고 해시코드가 같은 경우는 같은 링크드 리스트(서랍)에 저장된 서로 다른 두 데이터가 된다.

1.11 TreeMap

TreeMap은 이름에서 알 수 있듯이 이진검색트리의 형태로 키와 값의 쌍으로 이루어진 대이터를 저장한다. 그래서 검색과 정렬에 적합한 컬렉션 클래스이다.

이진 검색 트리와 Map에 대해서는 이미 충분히 설명했기 때문에 더 이상의 설명은 필요 없으리라 생각한다. 한가지 설명할 것이 있다면 HashMap과 TreeMap의 검색성능에 관한 것인데, 검색에 관한 대부분의 경우에서 HashMap이 TreeMap보다 더 뛰어나므로 HashMap을 사용하는 것이 좋다. 다만 범위검색이나 정렬이 필요한 경우에는 TreeMap을 사용하자.

설명

메서드	설명
TreeMap()	TreeMap 객체를 생성
TreeMap(Comparator c)	지정된 Comparator를 기준으로 정렬하는 TreeMap 객체를 생성
TreeMap(Map m)	주어진 Map에 저장된 모든 요소를 포함하는 TreeMap을 생성
TreeMap(SortedMap m)	주어진 SortedMap에 저장된 모든 요소를 포함하는 TreeMap을 생성
Map.Entry ceilingEntry(Object key)	지정된 key와 일치하거나 큰 것 중 제일 작은 것의 키와 값의 쌍(Map.Entry)를 반환. 없으면 null을 반환
Object ceilingKey(Object key)	지정된 key와 일치하거나 큰 것 중 제일 작은 것의 키를 반환. 없으면 null을 반환
void clear()	TreeMap에 저장된 모든 객체를 제거
Object clone()	현재 TreeMap을 복제해서 반환
Comparator comparator()	TreeMap의 정렬기준이 되는 Comparator를 반환 Comparator가 지정되지 않았다면 null을 반환
boolean containsKey(Object key)	TreeMap에 지정된 키(key)가 포함되어 있는지 알려줌 (포함되어 있으면 true)
boolean containsValue(Object value)	TreeMap에 지정된 값(value)가 포함되어 있는지 알려줌 (포함되어 있으면 true)
NavigableSet descendingKeySet()	TreeMap에 저장된 키를 역순으로 정렬해서 NavigableSet에 담아서 반환
Set entrySet()	TreeMap에 저장된 키와 값을 엔트리(키와 값의 결합)의 형태로 Set에 저장해서 반환
Map.Entry firstEntry()	TreeMap에 저장된 첫번째 (가장 작은) 키와 값의 쌍(Map.Entry)을 반환
Object firstKey()	TreeMap에 저장된 첫번째 (가장 작은) 키를 반환
Map.Entry floorEntry(Object key)	지정된 key와 일치하거나 작은 것 중에서 제일 큰 키의 쌍(Map.Entry)을 반환. 없으면 null을 반환
Object floorKey(Object key)	지정된 key와 일치하거나 작은 것 중에서 제일 큰 키를 반환. 없으면 null을 반환
Object get(Object key)	지정된 키(key)의 값(객체)을 반환
SortedMap headMap(Object toKey)	TreeMap에 저장된 첫번째 요소부터 지정된 범위에 속한 모든 요소가 담긴 SortedMap을 반환(이후 리프레시)

<code>NavigableMap headMap(Object toKey, boolean inclusive)</code>	TreeMap에 저장된 첫번째 요소부터 지정된 범위에 속한 모든 요소가 담긴 SortedMap을 반환. inclusive의 값이 true면 toKey도 포함
<code>Map.Entry higherEntry(Object key)</code>	지정된 key보다 큰 키 중에서 제일 작은 키의 쌍(Map.Entry)을 반환. 없으면 null을 반환
<code>Object higherKey(Object key)</code>	지정된 key보다 큰 키 중에서 제일 작은 키의 쌍(Map.Entry)을 반환. 없으면 null을 반환
<code>boolean isEmpty()</code>	TreeMap이 비어있는지 알려준다.
<code>Set keySet()</code>	TreeMap에 저장된 모든 키가 저장된 Set을 반환
<code>Map.Entry lastEntry()</code>	TreeMap에 저장된 마지막 키(가장 큰 키)의 쌍을 반환
<code>Object lastKey()</code>	TreeMap에 저장된 마지막 키(가장 큰 키)를 반환
<code>Map.Entry lowerEntry(Object key)</code>	지정된 key보다 작은 키 중에서 제일 큰 키의 쌍(Map.Entry)을 반환. 없으면 null을 반환
<code>Object lowerKey(Object key)</code>	지정된 key보다 작은 키 중에서 제일 큰 키의 쌍(Map.Entry)을 반환. 없으면 null을 반환
<code>NavigableSet navigableKeySet()</code>	TreeMap의 모든 키가 담긴 NavigableSet을 반환
<code>Map.Entry pollFirstEntry()</code>	TreeMap에서 제일 작은 키를 제거하면서 반환
<code>Map.Entry pollLastEntry()</code>	TreeMap에서 제일 큰 키를 제거하면서 반환
<code>Object put(Object key, Object value)</code>	지정된 키와 값을 TreeMap에 저장
<code>void putAll(Map map)</code>	Map에 저장된 모든 요소를 TreeMap에 저장
<code>Object remove(Object key)</code>	TreeMap에서 지정된 키로 저장된 값(객체)를 제거
<code>Object replace(Object k, Object v)</code>	기존의 키(k)의 값을 지정된 값(v)으로 변경.
<code>boolean replace(Object key, Object oldValue, Object newValue)</code>	기존의 키(k)의 값을 새로운 값(newValue)으로 변경 단, 기존의 값과 지정된 값(oldValue)가 일치해야함
<code>int size()</code>	TreeMap에 저장된 요소의 개수를 반환
<code>NavigableMap subMap(Object fromKey, boolean fromInclusive, Object toKey, boolean toInclusive)</code>	지정된 두개의 키 사이에 있는 모든 요소들이 담긴 NavigableMap을 반환. fromInclusive가 true면 범위에 fromKey 포함. toInclusive가 true면 범위에 toKey 포함
<code>SortedMap subMap(Object fromKey, Object toKey)</code>	지정된 두 개의 키 사이에 있는 모든 요소들이 담긴 SortedMap을 반환(toKey는 포함되지 않는다.)
<code>SortedMap tailMap(Object fromKey)</code>	지정된 키부터 마지막 요소의 범위에 속한 요소가 담긴 SortedMap을 반환
<code>NavigableMap tailMap(Object fromKey, boolean inclusive)</code>	지정된 키부터 마지막 요소의 범위에 속한 요소가 담긴 NavigableMap을 반환. inclusive가 true면 fromKey 포함
<code>Collection values()</code>	TreeMap에 저장된 모든 값을 컬렉션의 형태로 반환

▲ 표 11-19 TreeMap의 생성자와 메서드

| 참고 | JDK1.8부터 새로 추가된 메서드 중 람다와 스트림에 관련된 것들은 제외됨

▼ 예제 11-34/ch11/TreeMapEx1.java

```
import java.util.*;
class TreeMapEx1 {
    public static void main(String[] args) {
        String[] data = { "A", "K", "A", "K", "D", "K", "A", "K", "K", "Z", "D" };
        TreeMap map = new TreeMap();
        for(int i=0; i < data.length; i++) {
            if(map.containsKey(data[i])) {
                Integer value = (Integer)map.get(data[i]);
                map.put(data[i], new Integer(value.intValue() + 1));
            } else {
                map.put(data[i], new Integer(1));
            }
        }
        Iterator it = map.entrySet().iterator();
        System.out.println("= 기본정렬 =" );
        while(it.hasNext()) {
            Map.Entry entry = (Map.Entry)it.next();
            int value = ((Integer)entry.getValue()).intValue();
            System.out.println(entry.getKey() + " : " + printBar('#', value)
                               + " " + value );
        }
        System.out.println();
        // map을 ArrayList로 변환한 다음에 Collections.sort()로 정렬
        Set set = map.entrySet();
        List list = new ArrayList(set); // ArrayList(Collection c)
        // static void sort(List list, Comparator c)
        Collections.sort(list, new ValueComparator());
        it = list.iterator();
        System.out.println("= 값의 크기가 큰 순서로 정렬 =" );
        while(it.hasNext()) {
            Map.Entry entry = (Map.Entry)it.next();
            int value = ((Integer)entry.getValue()).intValue();
            System.out.println(entry.getKey() + " : " + printBar('#', value)
                               + " " + value );
        }
    } // public static void main(String[] args)
    static class ValueComparator implements Comparator {
        public int compare(Object o1, Object o2) {
            if(o1 instanceof Map.Entry && o2 instanceof Map.Entry) {
                Map.Entry e1 = (Map.Entry)o1;
                Map.Entry e2 = (Map.Entry)o2;
                int v1 = ((Integer)e1.getValue()).intValue();
                int v2 = ((Integer)e2.getValue()).intValue();
                if(v1 < v2) return -1;
                else if(v1 > v2) return 1;
                else return 0;
            }
        }
    }
}
```

```

        return v2 - v1;
    }
    return -1;
}
} // static class ValueComparator implements Comparator {
public static String printBar(char ch, int value) {
    char[] bar = new char[value];
    for(int i=0; i < bar.length; i++) {
        bar[i] = ch;
    }
    return new String(bar);
}
}

```

▼ 실행결과

= 기본정렬 =
A : ##### 3
D : ## 2
K : ##### 6
Z : # 1

= 값의 크기가 큰 순서로 정렬 =
K : ##### 6
A : #### 3
D : ## 2
Z : # 1

이 예제는 예제11-33 HashMapEx4.java를 TreeMap을 이용해서 변형한 것인데 TreeMap을 사용했기 때문에 HashMapEx4.java의 결과와는 달리 키가 오름차순으로 정렬되어 있는 것을 알 수 있다. 키가 String인스턴스이기 때문에 String클래스에 정의된 정렬기준에 의해서 정렬된 것이다.

그리고 Comparator를 구현한 클래스와 Collections.sort(List list, Comparator c)를 이용해서 값에 대한 내림차순으로 정렬하는 방법을 보여 준다.

1.12 Properties

Properties는 HashMap의 구버전인 Hashtable을 상속받아 구현한 것으로, Hashtable은 키와 값을 (Object, Object)의 형태로 저장하는데 비해 Properties는 (String, String)의 형태로 저장하는 보다 단순화된 컬렉션 클래스이다.

주로 애플리케이션의 환경 설정과 관련된 속성(property)을 저장하는데 사용되며 데이터를 파일로부터 읽고 쓰는 편리한 기능을 제공한다. 그래서 간단한 입출력은 Properties를 활용하면 몇 줄의 코드로 쉽게 해결될 수 있다.

메서드	설명
Properties()	Properties 객체를 생성한다.
Properties(Properties defaults)	지정된 Properties에 저장된 목록을 가진 Properties 객체를 생성한다.
String getProperty(String key)	지정된 키(key)의 값(value)을 반환한다.
String getProperty(String key, String defaultValue)	지정된 키(key)의 값(value)을 반환한다. 키를 못찾으면 defaultValue를 반환한다.
void list(PrintStream out)	지정된 PrintStream에 저장된 목록을 출력한다.
void list(PrintWriter out)	지정된 PrintWriter에 저장된 목록을 출력한다.
void load(InputStream inStream)	지정된 InputStream으로부터 목록을 읽어서 저장한다.
void load(Reader reader)	지정된 Reader으로부터 목록을 읽어서 저장한다.
void loadFromXML(InputStream in)	지정된 InputStream으로부터 XML문서를 읽어서, XML 문서에 저장된 목록을 읽어다 담는다. (load & store)
Enumeration propertyNames()	목록의 모든 키(key)가 담긴 Enumeration을 반환한다.
void save(OutputStream out, String header)	deprecated되었으므로 store()를 사용하자.
Object setProperty(String key, String value)	지정된 키와 값을 저장한다. 이미 존재하는 키(key)면 새로운 값(value)로 바뀐다.
void store(OutputStream out, String comments)	저장된 목록을 지정된 OutputStream에 출력(저장)한다. comments는 목록에 대한 주석으로 저장된다.
void store(Writer writer, String comments)	저장된 목록을 지정된 Writer에 출력(저장)한다. comments는 목록에 대한 설명(주석)으로 저장된다.
void storeToXML(OutputStream os, String comment)	저장된 목록을 지정된 출력스트림에 XML문서로 출력(저장)한다. comment는 목록에 대한 설명(주석)으로 저장된다.
void storeToXML(OutputStream os, String comment, String encoding)	저장된 목록을 지정된 출력스트림에 해당 인코딩의 XML 문서로 출력(저장)한다. comment는 목록에 대한 설명(주석)으로 저장된다.
Set stringPropertyNames()	Properties에 저장되어 있는 모든 키(key)를 Set에 담아서 반환한다.

▲ 표11-20 Properties의 생성자와 메서드

▼ 예제 11-35/ch11/PropertiesEx1.java

```
import java.util.*;
class PropertiesEx1 {
```

```

public static void main(String[] args) {
    Properties prop = new Properties();
    // prop에 키와 값(key, value)을 저장한다.
    prop.setProperty("timeout", "30");
    prop.setProperty("language", "kr");
    prop.setProperty("size", "10");
    prop.setProperty("capacity", "10");

    // prop에 저장된 요소들을 Enumeration을 이용해서 출력한다.
    Enumeration e = prop.propertyNames();

    while(e.hasMoreElements()) {
        String element = (String)e.nextElement();
        System.out.println(element + "=" + prop.getProperty(element));
    }

    System.out.println();
    prop.setProperty("size", "20"); // size의 값을 20으로 변경한다.
    System.out.println("size=" + prop.getProperty("size"));
    System.out.println("capacity=" + prop.getProperty("capacity", "20"));
    System.out.println("loadfactor=" + prop.getProperty("loadfactor", "0.75"));

    System.out.println(prop); // prop에 저장된 요소들을 출력한다.
    prop.list(System.out); // prop에 저장된 요소들을 화면(System.out)에 출력한다.
}
}

```

▼ 실행결과

```

capacity=10
size=10
timeout=30
language=kr

size=20
capacity=10
loadfactor=0.75 ← loadfactor라는 키가 없기 때문에 디폴트 값으로 지정한 0.75가 출력되었다.
{capacity=10, size=20, timeout=30, language=kr} ← System.out.println(prop);
-- listing properties --
capacity=10
size=20
timeout=30
language=kr

```

prop.list(System.out);에
의해서 출력된 내용

Properties의 기본적인 메서드를 이용해서 저장하고 읽어오고 출력하는 방법을 보여주는 간단한 예제이다. 데이터를 저장하는데 사용되는 setProperty()는 단순히 Hashtable의 put메서드를 호출할 뿐이다. 그리고 setProperty()는 기존에 같은 키로 저장된 값이 있는 경우 그 값을 Object타입으로 반환하며, 그렇지 않을 때는 null을 반환한다.

Object setProperty(String key, String value)

`getProperty()`는 Properties에 저장된 값을 읽어오는 일을 하는데, 만일 읽어오려는 키가 존재하지 않으면 지정된 기본값(defaultValue)을 반환한다.

```
String getProperty(String key)
String getProperty(String key, String defaultValue)
```

Properties는 Hashtable을 상속받아 구현한 것이라 Map의 특성상 저장순서를 유지하지 않기 때문에 예제의 결과에 출력된 순서가 저장순서와는 무관하다는 것을 확인하자.

Properties는 컬렉션프레임워크 이전의 구버전이므로 Iterator가 아닌 Enumeration을 사용한다. 그리고 list메서드를 이용하면 Properties에 저장된 모든 데이터를 화면 또는 파일에 편리하게 출력할 수 있다.

```
void list(PrintStream out)
void list(PrintWriter out)
```

System.out은 화면과 연결된 표준출력으로 System클래스에 정의된 PrintStream타입의 static변수이다. 보다 자세한 것은 '15장 입출력(I/O)'에서 설명할 것이므로 지금은 일단 이 정도만 이해하고 넘어가자.

▼ 예제 11-36/ch11/PropertiesEx2.java

```
import java.io.*;
import java.util.*;

class PropertiesEx2 {
    public static void main(String[] args) {
        // commandline에서 inputfile을 지정해주지 않으면 프로그램을 종료한다.
        if(args.length != 1) {
            System.out.println("USAGE: java PropertiesEx2 INPUTFILENAME");
            System.exit(0);
        }

        Properties prop = new Properties();
        String inputFile = args[0];
        try {
            prop.load(new FileInputStream(inputFile));
        } catch(IOException e) {
            System.out.println("지정된 파일을 찾을 수 없습니다.");
            System.exit(0);
        }

        String name = prop.getProperty("name");
        String[] data = prop.getProperty("data").split(",");
        int max = 0, min = 0;
        int sum = 0;

        for(int i=0; i < data.length; i++) {
            int intValue = Integer.parseInt(data[i]);
            if (i==0) max = min = intValue;
            else if (intValue > max) max = intValue;
            else if (intValue < min) min = intValue;
            sum += intValue;
        }
        System.out.println("Name: " + name);
        System.out.println("Max: " + max);
        System.out.println("Min: " + min);
        System.out.println("Sum: " + sum);
        System.out.println("Avg: " + (sum / data.length));
    }
}
```

```

        if (max < intValue)
            max = intValue;
        else if (min > intValue)
            min = intValue;

        sum += intValue;
    }

    System.out.println("이름 :" + name);
    System.out.println("최대값 :" + max);
    System.out.println("최소값 :" + min);
    System.out.println("합계 :" + sum);
    System.out.println("평균 :" + (sum*100.0/data.length)/100);
}

```

▼ 실행결과

c:\jdk1.8\work\ch11>java PropertiesEx2 input.txt

이름 :Seong Namkung

최대값 :35

최소값 :1

합계 :111

평균 :11.1

c:\jdk1.8\work\ch11>type input.txt

이것은 주석입니다.

여러 줄도 가능하고요.

name=Seong Namkung

data=9,1,5,2,8,13,26,11,35,1

c:\jdk1.8\work\ch11>

외부파일(input.txt)로부터 데이터를 입력받아서 계산결과를 보여주는 예제이다. 외부파일의 형식은 라인단위로 키와 값이 '='로 연결된 형태이어야 하며 주석라인은 첫 번째 문자가 '#'이어야 한다.

정해진 규칙대로만 파일을 작성하면 load()를 호출하는 것만으로 쉽게 데이터를 읽어 올 수 있다. 다만 인코딩(encoding)문제로 한글이 깨질 수 있기 때문에 한글을 입력받으려면 아래와 같이 코드를 변경해야한다.

```

String name = prop.getProperty("name");

try {
    name = new String(name.getBytes("8859_1"), "EUC-KR");
} catch (Exception e) {}

```

파일로부터 읽어온 데이터의 인코딩을 라틴문자집합(8859_1)에서 한글완성형(EUC-KR) 또는 KSC5601)으로 변환해주는 과정을 추가한 것인데 이에 대한 자세한 설명은 생략하겠다. 우리가 사용하고 있는 OS의 기본 인코딩(encoding)이 유니코드가 아니라서 이런 변환이 필요한 것이라는 것만 이해하고 넘어가자.

| 참고 | key에 문자 '='를 포함시키고자 한다면 escape문자 '\'를 사용하여 '\='와 같이 표현한다.

▼ 예제 11-37/ch11/PropertiesEx3.java

```
import java.util.*;
import java.io.*;

class PropertiesEx3 {
    public static void main(String[] args) {
        Properties prop = new Properties();

        prop.setProperty("timeout", "30");
        prop.setProperty("language", "한글");
        prop.setProperty("size", "10");
        prop.setProperty("capacity", "10");

        try {
            prop.store(new FileOutputStream("output.txt")
                      , "Properties Example");
            prop.storeToXML(new FileOutputStream("output.xml")
                           , "Properties Example");
        } catch(IOException e) {
            e.printStackTrace();
        }
    } // main의 끝
}
```

▼ 실행결과 - output.txt의 내용

```
#Properties Example
#Mon Nov 09 16:16:05 KST 2015
capacity=10
size=10
timeout=30
language=\uD55C\uAE00 ← 한글이 unicode로 바뀌어 있다.
```

▼ 실행결과 - output.xml의 내용

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Properties Example</comment>
<entry key="capacity">10</entry>
<entry key="size">10</entry>
<entry key="timeout">30</entry>
<entry key="language">한글</entry> ← 한글이 그대로 저장되어 있다. (메모장에서는 괜찮은데,
</properties>
```

도스창에서는 글자가 깨져 보인다.)

이전 예제와는 반대로 Properties에 저장된 데이터를 store()와 storeToXML()를 이용해
서 파일로 저장하는 방법을 보여 준다. 여기서도 한글문제가 발생하는데 storeToXML()
로 저장한 XML은 Editplus나 Eclipse에서 한글편집이 가능하므로 데이터에 한글이 포함
된 경우 store()보다는 storeToXML()을 이용해서 XML로 저장하는 것이 좋다.

▼ 예제 11-38/ch11/PropertiesEx4.java

```

import java.util.*;

class PropertiesEx4{
    public static void main(String[] args) {
        Properties sysProp = System.getProperties();
        System.out.println("java.version :"
                           + sysProp.getProperty("java.version"));
        System.out.println("user.language :"
                           + sysProp.getProperty("user.language"));
        sysProp.list(System.out);
    }
}

```

▼ 실행결과

```

java.version :1.8.0_51
user.language :ko
-- listing properties --
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\jdk1.8\jre\bin
java.vm.version=25.51-b03
...중간 생략...
java.vm.info=mixed mode
java.version=1.8.0_51
java.ext.dirs=C:\jdk1.8\jre\lib\ext;C:\WINDOWS\Sun\...
sun.boot.class.path=C:\jdk1.8\jre\lib\resources.jar;C:\jd...
sun.stderr.encoding=ms949
java.vendor=Oracle Corporation
file.separator=\ 
java.vendor.url.bug=http://bugreport.sun.com/bugreport/
sun.cpu.endian=little
sun.io.unicode.encoding=UnicodeLittle
sun.stdout.encoding=ms949
sun.desktop=windows
sun.cpu.isalist=pentium_pro+mmx pentium_pro pentium+m...

```

시스템 속성을 가져오는 방법을 보여주는 예제이다. System클래스의 getProperties()를 호출하면 시스템과 관련된 속성이 저장된 Properties를 가져올 수 있다. 이 중에서 원하는 속성을 getProperty()를 통해 얻을 수 있다.

이 예제를 실행시켜보고 어떠한 속성들이 있는지 살펴보고 어떤 속성을 통해 어떤 값을 얻을 수 있는지 확인하도록 하자.

| 참고 | 결과 중에 내용이 길어서 '...'로 생략된 값들이 있는데, getProperty()를 이용해서 출력하면 생략되지 않은 전체 값을 얻을 수 있다.

1.13 Collections

Arrays가 배열과 관련된 메서드를 제공하는 것처럼, Collections는 컬렉션과 관련된 메서드를 제공한다. fill(), copy(), sort(), binarySearch() 등의 메서드는 두 클래스에 모두 포함되어 있으며 같은 기능을 한다. 이 메서드들은 Arrays에서 이미 배웠으므로 설명을 생략한다.

| 주의 | java.util.Collection은 인터페이스이고, java.util.Collections는 클래스이다.

컬렉션의 동기화

멀티 쓰레드(multi-thread) 프로그래밍에서는 하나의 객체를 여러 쓰레드가 동시에 접근할 수 있기 때문에 데이터의 일관성(consistency)을 유지하기 위해서는 공유되는 객체에 동기화(synchronization)가 필요하다.

Vector와 Hashtable과 같은 구버전(JDK1.2 이전)의 클래스들은 자체적으로 동기화 처리가 되어 있는데, 멀티쓰레드 프로그래밍이 아닌 경우에는 불필요한 기능이 되어 성능을 떨어뜨리는 요인으로 된다.

그래서 새로 추가된 ArrayList와 HashMap과 같은 컬렉션은 동기화를 자체적으로 처리하지 않고 필요한 경우에만 java.util.Collections클래스의 동기화 메서드를 이용해서 동기화처리가 가능하도록 변경하였다.

Collections클래스에는 다음과 같은 동기화 메서드를 제공하고 있으므로, 동기화가 필요할 때 해당하는 것을 사용하면 된다.

```
static Collection synchronizedCollection(Collection c)
static List      synchronizedList(List list)
static Set       synchronizedSet(Set s)
static Map       synchronizedMap(Map m)
static SortedSet synchronizedSortedSet(SortedSet s)
static SortedMap synchronizedSortedMap(SortedMap m)
```

이 들을 사용하는 방법은 다음과 같다.

```
List syncList = Collections.synchronizedList(new ArrayList(...));
```

| 참고 | 멀티쓰레딩과 동기화에 대해서는 13장 쓰레드(Thread)에서 자세히 다룬 것이므로 지금은 동기화가 필요한 경우에 Collections클래스의 동기화메서드를 사용하면 된다는 것만 알면 된다.

변경불가 컬렉션 만들기

컬렉션에 저장된 데이터를 보호하기 위해서 컬렉션을 변경할 수 없게, 즉 읽기전용으로 만들어야 할 때가 있다. 주로 멀티 쓰레드 프로그래밍에서 여러 쓰레드가 하나의 컬렉션을 공유하다보면 데이터가 손상될 수 있는데, 이를 방지하려면 아래의 메서드들을 이용하자.

```
static Collection unmodifiableCollection(Collection c)
static List    unmodifiableList(List list)
```

```

static Set      unmodifiableSet(Set s)
static Map      unmodifiableMap(Map m)
static NavigableSet unmodifiableNavigableSet(NavigableSet s)
static SortedSet unmodifiableSortedSet(SortedSet s)
static NavigableMap unmodifiableNavigableMap(NavigableMap m)
static SortedMap unmodifiableSortedMap(SortedMap m)

```

싱글톤 컬렉션 만들기

인스턴스를 new 연산자가 아닌 메서드를 통해서만 생성할 수 있게 함으로써 생성할 수 있는 인스턴스의 개수를 제한하는 방법에 대해서 6장에서 배웠다. 이러한 기능을 제공하는 것이 바로 'singleton'으로 시작하는 메서드이다.

| 참고 | 싱글턴에 대해 기억이 잘 안 난다면, p.351을 참고하자.

```

static List singletonList(Object o)
static Set singleton(Object o)      // singletonSet이 아님
static Map singletonMap(Object key, Object value)

```

매개변수로 저장할 요소를 지정하면, 해당 요소를 저장하는 컬렉션을 반환한다. 그리고 반환된 컬렉션은 변경할 수 없다.

한 종류의 객체만 저장하는 컬렉션 만들기

컬렉션에 모든 종류의 객체를 저장할 수 있다는 것은 장점이기도하고 단점이기도 하다. 대부분의 경우 한 종류의 객체를 저장하며, 컬렉션에 지정된 종류의 객체만 저장할 수 있도록 제한하고 싶을 때 아래의 메서드를 사용한다.

```

static Collection checkedCollection(Collection c, Class type)
static List checkedList(List list, Class type)
static Set checkedSet(Set s, Class type)
static Map checkedMap(Map m, Class keyType, Class valueType)
static Queue checkedQueue(Queue queue, Class type)
static NavigableSet checkedNavigableSet(NavigableSet s, Class type)
static SortedSet checkedSortedSet(SortedSet s, Class type)
static NavigableMap checkedNavigableMap(NavigableMap m, Class keyType,
                                         Class valueType)
static SortedMap checkedSortedMap(SortedMap m, Class keyType,
                                  Class valueType)

```

사용방법은 다음과 같이 두 번째 매개변수에 저장할 객체의 클래스를 지정하면 된다.

```

List list = new ArrayList();
List checkedList = checkedList(list, String.class); // String만 저장가능
checkedList.add("abc");                // OK.
checkedList.add(new Integer(3)); // 예러. ClassCastException 발생

```

간단히 처리할 수 있는데도 이런 메서드들을 제공하는 이유는 호환성 때문이다. 자네릭스는 JDK1.5부터 도입된 기능이므로 JDK1.5이전에 작성된 코드를 사용할 때는 이 메서드들이 필요할 수 있다.

아직 소개하지 않은 메서드들은 다음 예제를 통해서 그 사용법을 충분히 이해할 수 있을 것이므로 설명을 생략한다.

▼ 예제 11-39/ch11/CollectionsEx.java

```
import java.util.*;
import static java.util.Collections.*;

class CollectionsEx {
    public static void main(String[] args) {
        List list = new ArrayList();
        System.out.println(list);

        addAll(list, 1, 2, 3, 4, 5);
        System.out.println(list);

        rotate(list, 2); // 오른쪽으로 두 칸씩 이동
        System.out.println(list);

        swap(list, 0, 2); // 첫 번째와 세 번째를 교환(swap)
        System.out.println(list);

        shuffle(list); // 저장된 요소의 위치를 임의로 변경
        System.out.println(list);

        sort(list); // 정렬
        System.out.println(list);

        sort(list, reverseOrder()); // 역순 정렬 reverse(list); 와 동일
        System.out.println(list);

        int idx = binarySearch(list, 3); // 3이 저장된 위치(index)를 반환
        System.out.println("index of 3 = " + idx);

        System.out.println("max=" + max(list));
        System.out.println("min=" + min(list));
        System.out.println("min=" + max(list, reverseOrder()));

        fill(list, 9); // list를 9로 채운다.
        System.out.println("list=" + list);

        // list와 같은 크기의 새로운 list를 생성하고 2로 채운다. 단, 결과는 변경불가
        List newList = nCopies(list.size(), 2);
        System.out.println("newList=" + newList);

        System.out.println(disjoint(list, newList)); // 공통요소가 없으면 true

        copy(list, newList);
        System.out.println("newList=" + newList);
        System.out.println("list=" + list);
    }
}
```

컬렉션에 저장할 요소의 타입을 제한하는 것은 다음 장에서 배울 지네릭스(generics)로 간단히 처리할 수 있는데도 이런 메서드들을 제공하는 이유는 호환성 때문이다. 지네릭스는 JDK1.5부터 도입된 기능이므로 JDK1.5이전에 작성된 코드를 사용할 때는 이 메서드들이 필요할 수 있다.

아직 소개하지 않은 메서드들은 다음 예제를 통해서 그 사용법을 충분히 이해할 수 있을 것이므로 설명을 생략한다.

▼ 예제 11-39/ch11/CollectionsEx.java

```
import java.util.*;
import static java.util.Collections.*;

class CollectionsEx {
    public static void main(String[] args) {
        List list = new ArrayList();
        System.out.println(list);

        addAll(list, 1, 2, 3, 4, 5);
        System.out.println(list);

        rotate(list, 2); // 오른쪽으로 두 칸씩 이동
        System.out.println(list);

        swap(list, 0, 2); // 첫 번째와 세 번째를 교환(swap)
        System.out.println(list);

        shuffle(list); // 저장된 요소의 위치를 임의로 변경
        System.out.println(list);

        sort(list); // 정렬
        System.out.println(list);

        sort(list, reverseOrder()); // 역순 정렬 reverse(list);와 동일
        System.out.println(list);

        int idx = binarySearch(list, 3); // 3이 저장된 위치(index)를 반환
        System.out.println("index of 3 = " + idx);

        System.out.println("max=" + max(list));
        System.out.println("min=" + min(list));
        System.out.println("min=" + max(list, reverseOrder()));

        fill(list, 9); // list를 9로 채운다.
        System.out.println("list=" + list);

        // list와 같은 크기의 새로운 list를 생성하고 2로 채운다. 단, 결과는 변경불가
        List newList = nCopies(list.size(), 2);
        System.out.println("newList=" + newList);

        System.out.println(disjoint(list,
```

```
replaceAll(list, 2, 1);
System.out.println("list="+list);

Enumeration e = enumeration(list);
ArrayList list2 = list(e);

System.out.println("list2="+list2);
}

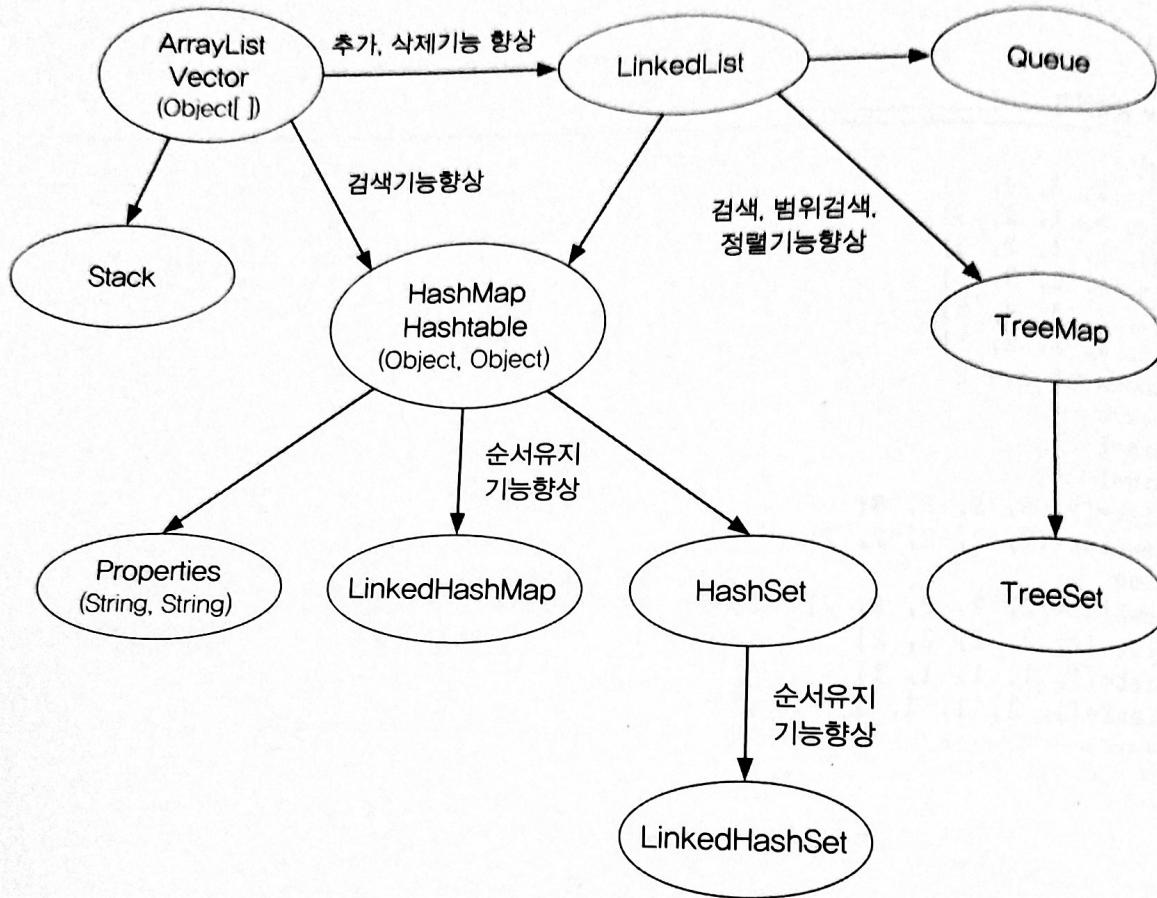
}
```

▼ 실행결과

```
[]
[1, 2, 3, 4, 5]
[4, 5, 1, 2, 3]
[1, 5, 4, 2, 3]
[4, 1, 2, 3, 5]
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
index of 3 = 2
max=5
min=1
min=1
list=[9, 9, 9, 9, 9]
newList=[2, 2, 2, 2, 2]
true
newList=[2, 2, 2, 2, 2]
list=[2, 2, 2, 2, 2]
list=[1, 1, 1, 1, 1]
list2=[1, 1, 1, 1, 1]
```

1.14 컬렉션 클래스 정리 & 요약

지금까지 소개한 컬렉션 클래스의 특징과 관계를 그림으로 정리해보았다. 각 컬렉션 클래스마다 장단점이 있으므로 구현원리와 특징을 잘 이해해서 상황에 가장 적합한 것을 선택하여 사용하길 바란다.



▲ 그림 11-27 컬렉션 클래스간의 관계

컬렉션	특징
ArrayList	배열기반, 데이터의 추가와 삭제에 불리, 순차적인 추가삭제는 제일 빠름. 임의의 요소에 대한 접근성(accessibility)이 뛰어남.
LinkedList	연결기반. 데이터의 추가와 삭제에 유리. 임의의 요소에 대한 접근성이 좋지 않다.
HashMap	배열과 연결이 결합된 형태. 추가, 삭제, 검색, 접근성이 모두 뛰어남. 검색에는 최고성을 보인다.
TreeMap	연결기반. 정렬과 검색(특히 범위검색)에 적합. 검색성능은 HashMap보다 떨어짐.
Stack	Vector를 상속받아 구현
Queue	LinkedList가 Queue인터페이스를 구현
Properties	Hashtable을 상속받아 구현
HashSet	HashMap을 이용해서 구현
TreeSet	TreeMap을 이용해서 구현
LinkedHashMap	HashMap과 HashSet에 저장순서유지기능을 추가
LinkedHashSet	

▲ 표 11-21 컬렉션 클래스의 특징