Chapter 15. 컬렉션 프레임워크

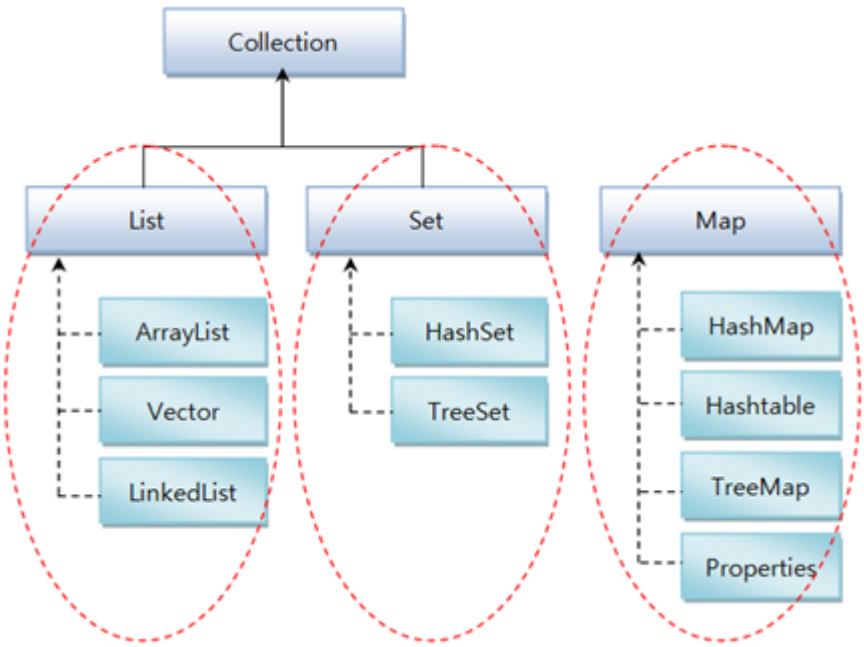
15.1 컬렉션 프레임워크 소개

: 애플리케이션을 개발하다 보면 다수의 객체를 저장해 두고 필요할 때마다 꺼내서 사용하는 경우가 많다. 가장 간단한 방법은 배열을 이용하는 것이다.

- **예시**

```
// 길이 10인 배열 생성
Product[] array = new Product[10];
// 객체 추가
array[0] = new Product("Model1");
array[1] = new Product("Model2");
// 객체 검색
Product model1 = array[0];
Product model2 = array[1];
// 객체 삭제
array[0] = null;
array[1] = null;
```

- **컬렉션 프레임워크(Collection Framework)** : 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 java.util 패키지에 컬렉션과 관련된 인터페이스와 클래스들을 포함시켜 놓았다.

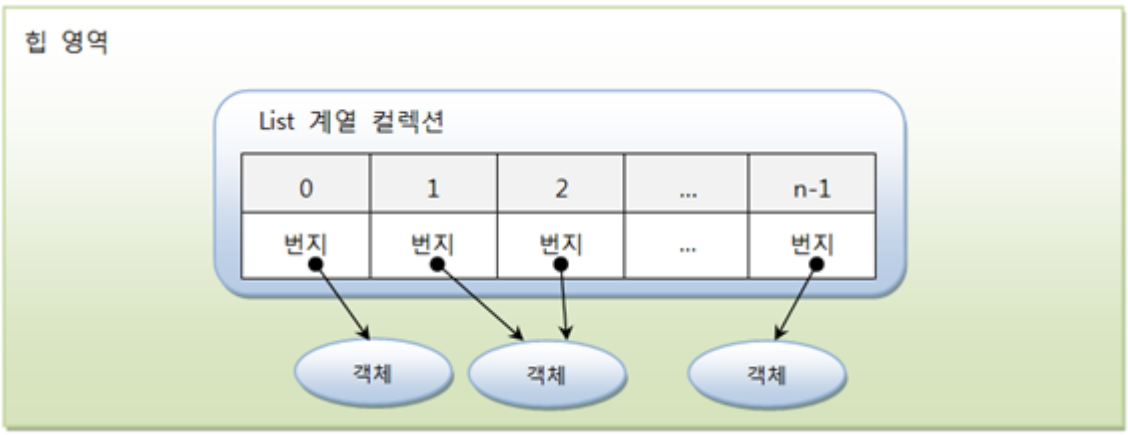


- **컬렉션 정리**

인터페이스 분류		특징	구현 클래스
Collection	List	- 순서를 유지하고 저장 - 중복 저장 가능	ArrayList, Vector, LinkedList
	Set	- 순서를 유지하지 않고 저장 - 중복 저장 안 됨	HashSet, TreeSet
Map		- 키와 값의 쌍으로 저장 - 키는 중복 저장 안 됨	HashMap, Hashtable, TreeMap, Properties

15.2 List 컬렉션

: 객체를 일렬로 늘어놓은 구조를 가지고 있다. 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공한다. 그리고 객체 자체를 저장하는 것이 아니라 객체의 번지를 참조한다.



- **List 인터페이스의 메소드들**

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨 끝에 추가
	void add(int index, E element)	주어진 인덱스에 객체를 추가
	E set(int index, E element)	주어진 인덱스에 저장된 객체를 주어진 객체로 바꿈
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	E get(int index)	주어진 인덱스에 저장된 객체를 리턴
	boolean isEmpty()	컬렉션이 비어 있는지 조사
	int size()	저장되어 있는 전체 객체 수를 리턴
객체 삭제	void clear()	저장된 모든 객체를 삭제
	E remove(int index)	주어진 인덱스에 저장된 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

- **예시**

```
List<String> list = ...;
list.add("홍길동");            // 맨끝에 객체 추가
list.add(1, "신용권");         // 지정된 인덱스에 객체 삽입
String str = list.get(1);     // 인덱스로 객체 찾기
list.remove(0);               // 인덱스로 객체 삭제
list.remove("신용권");         // 객체 삭제

// for문 사용
for(int i=0; i<list.size(); i++) {
    String str = list.get(i);
}
for(String str : list) {
}
```

15.2.1 ArrayList

: List 인터페이스의 구현 클래스로, 객체가 인덱스로 관리된다. 배열은 생성할 때 크기가 고정되고 사용 중에 크기를 변경할 수 없지만, ArrayList는 저장 용량(capacity)을 초과하면 자동으로 늘어난다.

- **예시**

```
// ArrayList 생성
// 기본 생성자로 객체를 생성하면 10개의 초기 용량(capacity)을 가진다.
List<String> list = new ArrayList<String>();

// String 객체를 30개를 저장할 수 있는 list
List<String> list = new ArrayList<String>(30);
```

> ArrayList에 객체를 추가하면 인덱스 0부터 차례대로 저장된다.
>
> 객체를 삽입하면 해당 인덱스부터 마지막 인덱스까지 모두 1씩 밀려난다. 객체를 제거하면 바로 뒤 인덱스부터 마지막 인덱스까지 모두 앞으로 1씩 당겨진다.
>
> 빈번한 객체 삭제와 삽입이 일어나면 **LinkedList**를 사용하고 인덱스 검색이나, 맨 마지막에 객체를 추가하는 경우에는 **ArrayList**가 적절하다.

- **예제**

```
package arraylist;

import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();

        // String 객체 5개 저장
        list.add("Java");
        list.add("JDBC");
        list.add("Servlet/JSP");
        list.add(2, "Database");
        list.add("iBATIS");

        // 저장된 총 객체 수 얻기
        int size = list.size();
        System.out.println("총 객체수 : " + size);
        System.out.println();

        // 2번 인덱스의 객체 얻기
        String skill = list.get(2);
        System.out.println("2: " + skill);
        System.out.println();

        // 저장된 총 객체 수만큼 루핑
        for(int i=0; i<list.size(); i++) {
            String str = list.get(i);
            System.out.println(i + ":" + str);
        }
        System.out.println();

        // 2번 인덱스 객체(Database) 삭제됨
        // 2번 인덱스 객체(Servlet/JSP) 삭제됨
        list.remove(2);
```

```
        list.remove(2);
        list.remove("iBATIS");

        // 저장된 총 객체 수만큼 루핑
        for(int i=0; i<list.size(); i++) {
            String str = list.get(i);
            System.out.println(i + ":" + str);
        }
    }
}
```

실행 결과

```
총 객체수 : 5

2: Database

0:Java
1:JDBC
2:Database
3:Servlet/JSP
4:iBATIS

0:Java
1:JDBC
```

> 런타임 시 필요에 의해 객체들을 추가하는 것이 일반적이지만, 고정된 객체들로 구성된 List를 생성할 때도 있다.이런 경우에는 Arrays.asList(T...a) 메소드를 사용하는 것이 간편하다.

- **asList() 메소드 예제**

```
package arraylist;

import java.util.Arrays;
import java.util.List;

public class ArrayAsListExample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("홍길동", "신용권", "김자바");
        for(String name : list1) {
            System.out.println(name);
        }

        List<Integer> list2 = Arrays.asList(1, 2, 3);
        for(int value : list2) {
            System.out.println(value);
        }
    }
}
```

15.2.2 Vector

: Vector를 생성하기 위해서는 저장할 객체 타입을 타입 파라미터로 표기하고 기본 생성자를 호출하면 된다.

```
List<E> list = new Vector<E>();
```

- **ArrayList와 다른점**

  : Vector는 동기화된(synchronized) 메소드로 구성되어 있기 때문에 하나의 스레드가 실행을 완료해야만 다른 스레드를 실행할 수 있다. 그래서 멀티 스레드 환경에서 안전하게 객체를 추가, 삭제할 수 있다.(**Thread Safe**)

- **예제**

  Board.java(**게시물 정보 객체**)

```
package vector;

public class Board {
    String subject;
    String content;
    String writer;

    public Board(String subject, String content, String writer) {
        this.subject = subject;
        this.content = content;
        this.writer = writer;
    }
}
```

  VectorExample.java(**Board 객체를 저장하는 Vector**)

```
package vector;

import java.util.List;
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        List<Board> list = new Vector<Board>();

        // Board 객체를 저장
        list.add(new Board("제목1", "내용1", "글쓴이1"));
        list.add(new Board("제목2", "내용2", "글쓴이2"));
        list.add(new Board("제목3", "내용3", "글쓴이3"));
        list.add(new Board("제목4", "내용4", "글쓴이4"));
        list.add(new Board("제목5", "내용5", "글쓴이5"));

        // 2번 인덱스 객체(제목3) 삭제(뒤의 인덱스는 1씩 앞으로 당겨짐)
        list.remove(2);
        // 3번 인덱스 객체(제목5) 삭제
        list.remove(3);
```

```
        for(int i=0; i<list.size(); i++) {
            Board board = list.get(i);
            System.out.println(board.subject + "\t" +
                    board.content + "\t" +
                    board.writer);
        }
    }
}
```

실행 결과

```
제목1    내용1    글쓴이1
제목2    내용2    글쓴이2
제목4    내용4    글쓴이4
```

15.2.3 LinkedList

: ArrayList와 사용 방법은 똑같지만 내부 구조는 완전 다르다. LinkedList는 인접 참조를 링크해서 체인 처럼 관리한다. 특정 인덱스의 객체를 제거하면 앞뒤 링크만 병경되고 나머지 링크는 변경되지 않는다.

- **LinkedList 생성**

```
List<E> list = new LinkedList<E>();
```

- **예제(ArrayList와 LinkedList의 실행 성능 비교)**

```
package linked_list;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class LinkedListExample {
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<String>();
        List<String> list2 = new LinkedList<String>();

        long startTime;
        long endTime;

        startTime = System.nanoTime();
        for(int i=0; i<10000; i++) {
            list1.add(0, String.valueOf(i));
        }
        endTime = System.nanoTime();

        System.out.println("ArrayList 걸린시간: " +
```

```
                (endTime-startTime) + " ns");

        startTime = System.nanoTime();
        for(int i=0; i<10000; i++) {
            list2.add(0, String.valueOf(i));
        }
        endTime = System.nanoTime();

        System.out.println("LinkedList 걸린시간: " +
                (endTime-startTime) + " ns");
    }
}
```

실행 결과

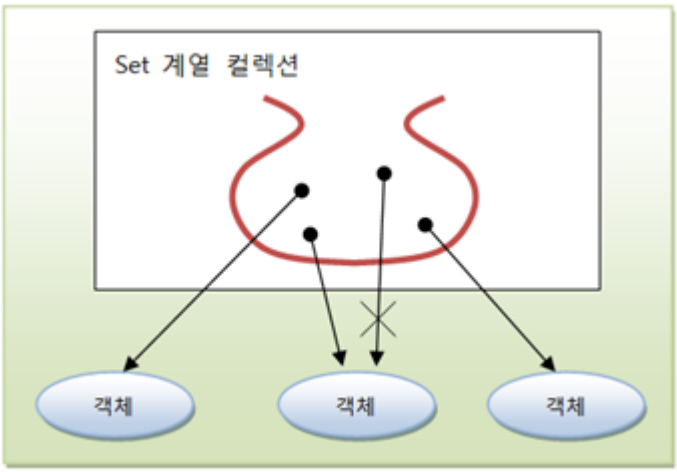
```
ArrayList 걸린시간: 9391773 ns
LinkedList 걸린시간: 3547017 ns
```

- **ArrayList와 LinkedList 비교**

구분	순차적으로 추가/삭제	중간에 추가/삭제	검색
ArrayList	빠르다	느리다	빠르다
LinkedList	느리다	빠르다	느리다

15.3 Set 컬렉션

: 저장 순서가 유지되지 않는다. 또한 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있다.



- **Set 인터페이스의 메소드들**

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 저장, 객체가 성공적으로 저장되면 true를 리턴하고 중복 객체면 false를 리턴
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	boolean isEmpty()	컬렉션이 비어 있는지 조사
	Iterator<E> iterator()	저장된 객체를 한 번씩 가져오는 반복자 리턴
	int size()	저장되어 있는 전체 객체 수 리턴
객체 삭제	void clear()	저장된 모든 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

- **예시**

```
Set<String> set = ...;
set.add("홍길동");               // 객체 추가
set.add("신용권");
set.remove("홍길동");    // 객체 삭제

// 전체 객체를 대상으로 한번씩 반복해서 가져오는 반복자(Iterator)
Set<String> set = ...;
Iterator<String> iterator = set.iterator();
```

- **Iterator 인터페이스 메소드**

리턴 타입	메소드명	설명
boolean	hasNext()	가져올 객체가 있으면 true를 리턴하고 없으면 false를 리턴한다.
E	next()	컬렉션에서 하나의 객체를 가져온다.
void	remove()	Set 컬렉션에서 객체를 제거한다.

- **Iterator 사용 예시**

```
Set<String> set = ...;
Iterator<String> iterator = set.iterator();
// 저장된 객체 수만큼 루핑
while(iterator.hasNext()) {
    // String 객체 하나를 가져옴
    String str = iterator.next();
```

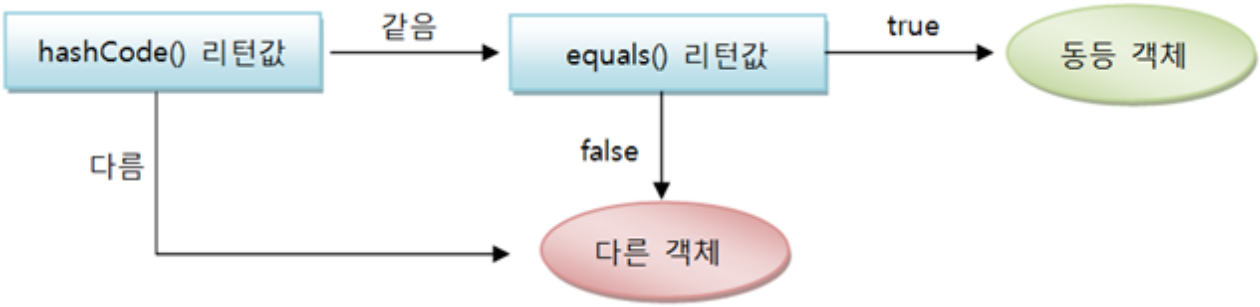
```
}

// 향상된 for문 사용
Set<String> set = ...;
for(String str : set) {
    // 저장된 객체 수만큼 루핑
}

// 객체 제거
while(iterator.hasNext()) {
    String str = iterator.next();
    if(str.equals("홍길동")) {
        iterator.remove();
    }
}
```

15.3.1 HashSet

: Set 인터페이스의 구현 클래스이다.

- **HashSet 생성자**

```
Set<E> set = new HashSet<E>();

// 예시
Set<String> set = new HashSet<String>();
```



- **예제(String 객체를 중복 없이 저장하는 HashSet)**

```
package hash_set;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class HashSetExample1 {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();

        // "Java"는 한 번만 저장됨
        set.add("Java");
```

```
        set.add("JDBC");
        set.add("Servlet/JSP");
        set.add("Java");
        set.add("iBATIS");

        // 저장된 객체 수 얻기
        int size = set.size();
        System.out.println("총 객체수: " + size);

        // 반복자 얻기
        Iterator<String> iterator = set.iterator();

        // 객체 수만큼 루핑
        while(iterator.hasNext()) {
            // 한 개의 객체를 가져온다.
            String element = iterator.next();
            System.out.println("\t" + element);
        }

        // 두 개의 객체 삭제
        set.remove("JDBC");
        set.remove("iBATIS");

        // 저장된 객체 수 얻기
        System.out.println("총 객체수: " + set.size());

        // 반복자 얻기
        iterator = set.iterator();

        // 객체 수만큼 루핑
        while(iterator.hasNext()) {
            String element = iterator.next();
            System.out.println("\t" + element);
        }

        // 모든 객체를 제거하고 비움
        set.clear();
        if(set.isEmpty()) {
            System.out.println("비어 있음");
        }
    }
}
```

```
총 객체수: 4
        Java
        JDBC
        Servlet/JSP
        iBATIS
총 객체수: 2
        Java
        Servlet/JSP
비어 있음
```

- **예제(중복 저장 없이 저장)**

  Member.java(hashCode()와 equals() 메소드 재정의)

```
package hash_set;

public class Member {
    public String name;
    public int age;

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

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Member) {
            Member member = (Member) obj;
            return member.name.equals(name) && (member.age == age);
        } else {
            return false;
        }
    }

    @Override
    // name 과 age 값이 같으면 동일한
    // hashCode 가 리턴
    public int hashCode() {
        return name.hashCode() + age;
    }
}
```

  HashSetExample2.java(Member 객체를 중복없이 저장하는 HashSet)

```
package hash_set;

import java.util.HashSet;
import java.util.Set;

public class HashSetExample2 {
    public static void main(String[] args) {
        Set<Member> set = new HashSet<Member>();

        // 인스턴스는 다르지만 내부 데이터가
        // 동일하므로 객체 1개만 저장
        set.add(new Member("홍길동", 30));
        set.add(new Member("홍길동", 30));

        // 저장된 객체 수 얻기
        System.out.println("총 객체수 : " + set.size());
```

```
    }
}
```

실행 결과

```
총 객체수 : 1
```

15.4 Map 컬렉션

: 키(key)와 값(value)으로 구성된 Entry 객체를 저장하는 구조를 가지고 있다. 여기서 키와 값은 모두 객체이다. 키는 중복될 수 없지만 값은 중복 저장될 수 있다.

- **Map 인터페이스의 메소드들**

기능	메소드	설명
객체 추가	V put(K key, V value)	주어진 키로 값을 저장, 새로운 키일 경우 null을 리턴하고 동일한 키가 있을 경우 값을 대체하고 이전 값을 리턴
객체 검색	boolean containsKey(Object key)	주어진 키가 있는지 여부
	boolean containsValue(Object value)	주어진 값이 있는지 여부
	Set<Map.Entry<K,V>> entrySet()	키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set에 담아서 리턴
	V get(Object key)	주어진 키가 있는 값을 리턴
	boolean isEmpty()	컬렉션이 비어 있는지 여부
	Set<K> keySet()	모든 키를 Set 객체에 담아서 리턴
	int size()	저장된 키의 총 수를 리턴
	Collection<V> values()	저장된 모든 값을 Collection에 담아서 리턴
객체 삭제	void clear()	모든 Map.Entry(키와 값)를 삭제
	V remove(Object key)	주어진 키와 일치하는 Map.Entry를 삭제하고 값을 리턴

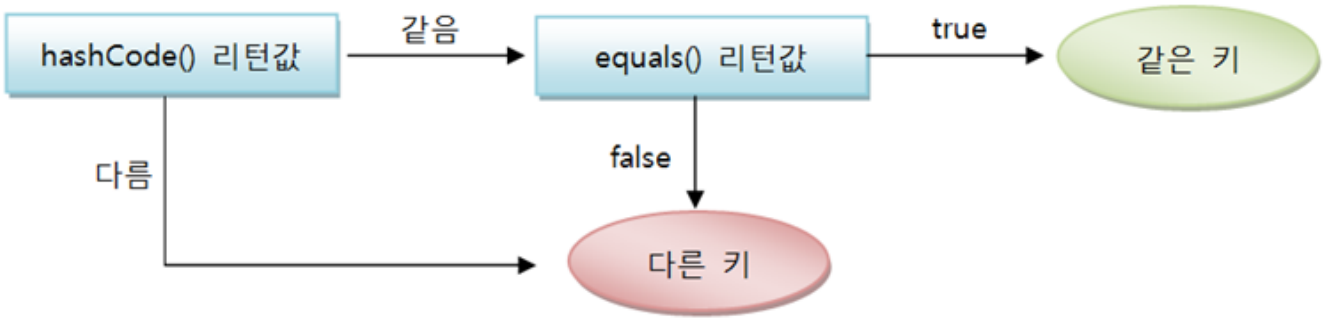
- **Map 사용 예시**

```
Map<String, Integer> map = ~;
map.put("홍길동", 30);                          // 객체 추가
int score = map.get("홍길동");           // 객체 찾기
map.remove("홍길동");                           // 객체 삭제

// keySet() 메소드로 모든 키 가져오기
Map<K, V> map = ~;
Set<K> keySet = map.keySet();
Iterator<K> keyIterator = key.Set.iterator();
while(keyIterator.hasNext()) {
    K key = keyIterator.next();
    V value = map.get(key);
}
```

```
// entrySet() 메소드로 키와 값의 쌍 모두 가져오기
Set<Map.Entry<K,V>> entrySet = map.entrySet();
Iterator<Map.Entry<K, V>> entryIterator = entrySet.iterator();
while(entryIterator.hasNext()) {
    Map.Entry<K, V> entry = entryIterator.next();
    K key = entry.getKey();
    V value = entry.getValue();
}
```

15.4.1 HashMap

: HashMap은 Map 인터페이스를 구현한 대표적인 Map 컬렉션이다. 동등객체, 즉 동일한 키가 될 조건은 hashCode()의 리턴값이 같아야하고, equals() 메소드가 true를 리턴해야 한다.



- **HashMap 생성자 호출**

```
// K = 키
// V = 값
Map<K, V> map = new HashMap<K, V>();
```

- **예제(이름을 키로 점수를 값으로 저장하기)**

```
package hash_map;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HashMapExample1 {
    public static void main(String[] args) {
        // Map 컬렉션 생성
        Map<String, Integer> map = new HashMap<String, Integer>();

        // 객체 저장
        map.put("신용권", 85);
        map.put("홍길동", 90);
        map.put("동장군", 80);
        // "홍길동" 키 값이 같기 때문에
        // 제일 마지막에 저장한 값으로 대치
```

```
        map.put("홍길동", 95);

        // 저장된 총 Entry 수 얻기
        System.out.println("총 Entry 수: " + map.size());

        // 객체 찾기
        // 이름(키)으로 점수(값)를 검색
        System.out.println("\t홍길동 : " + map.get("홍길동"));
        System.out.println();

        // 객체를 하나씩 처리
        // Key Set 얻기
        Set<String> keySet = map.keySet();

        // 반복해서 키를 얻고 값을 Map 에서 얻어냄
        Iterator<String> keyIterator = keySet.iterator();
        while(keyIterator.hasNext()) {
            String key = keyIterator.next();
            Integer value = map.get(key);
            System.out.println("\t" + key + " : " + value);
        }
        System.out.println();

        // 객체 삭제
        // 키로 Map.Entry 를 제거
        map.remove("홍길동");
        System.out.println("총 Entry 수: " + map.size());

        // 객체를 하나씩 처리
        // Map.Entry Set 얻기
        Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
        Iterator<Map.Entry<String, Integer>> entryIterator =
                entrySet.iterator();

        // 반복해서 Map.Entry 를 얻고
        // 키와 값을 얻어냄
        while(entryIterator.hasNext()) {
            Map.Entry<String, Integer> entry = entryIterator.next();
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println("\t" + key + " : " + value);
        }
        System.out.println();

        // 객체 전체 삭제
        // 모든 Map.Entry 삭제
        map.clear();
        System.out.println("총 Entry 수 : " + map.size());
    }
}
```

실행 결과

```
총 Entry 수: 3
        홍길동 : 95

        홍길동 : 95
        신용권 : 85
        동장군 : 80

총 Entry 수: 2
        신용권 : 85
        동장군 : 80

총 Entry 수 : 0
```

- **예제(학번과 이름이 동일한 경우 같은 키로 인식)**

  Student.java

```
package hash_map;

public class Student {
    public int sno;
    public String name;

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

    public boolean equals(Object obj) {
        // 학번과 이름이 동일할 경우 true를 리턴
        if(obj instanceof Student) {
            Student student = (Student) obj;
            return (sno==student.sno) && (name.equals(student.name));
        } else {
            return false;
        }
    }

    public int hashCode() {
        // 학번과 이름이 같다면 동일한 값을 리턴
        return sno + name.hashCode();
    }
}
```

  HashMapExample2.java

```
package hash_map;

import java.util.HashMap;
import java.util.Map;

public class HashMapExample2 {
```

```
    public static void main(String[] args) {
        Map<Student, Integer> map = new HashMap<Student, Integer>();

        // 학번과 이름이 동일한 Student 를 키로 저장
        map.put(new Student(1, "홍길동"), 95);
        map.put(new Student(1, "홍길동"), 95);

        // 저장된 총 Map.Entry 수 얻기
        System.out.println("총 Entry 수: " + map.size());
    }
}
```

실행 결과

```
총 Entry 수: 1
```

15.4.2 Hashtable

: HashMap과 동일한 내부 구조를 가지고 있다. Hashtable도 키로 사용할 객체는 hashCode()와 equals() 메소드를 재정의해서 동등 객체가 될 조건을 정해야 한다. HashMap과 차이점은 멀티 스레드 환경에서 안전하게 객체를 추가, 삭제할 수 있다.

- **Hashtable 생성자 호출**

```
// K = 키 타입
// V = 값 타입
Map<K, V> map = new Hashtable<K, V>();
```

- **예제(아이디와 비밀번호 검사하기)**

```
package hash_table;

import java.util.Hashtable;
import java.util.Map;
import java.util.Scanner;

public class HashtableExample {
    public static void main(String[] args) {
        Map<String, String> map = new Hashtable<String, String>();

        // 아이디와 비밀번호를 미리 저장시킨다.
        map.put("spring", "12");
        map.put("summer", "123");
        map.put("fall", "1234");
        map.put("winter", "12345");

        // 키보드로부터 입력된 내용을 받기 위해 생성
        Scanner scanner = new Scanner(System.in);
```

```
        while(true) {
            System.out.println("아이디와 비밀번호를 입력해주세요");
            System.out.print("아이디: ");

            // 키보드로 입력한 아이디를 읽는다.
            String id = scanner.nextLine();
            System.out.print("비밀번호: ");

            // 키보드로 입력한 비밀번호를 읽는다.
            String password = scanner.nextLine();
            System.out.println();

            // 아이디인 키가 존재하는지 확인한다.
            if(map.containsKey(id)) {
                // 비밀번호를 비교한다.
                if(map.get(id).equals(password)) {
                    System.out.println("로그인 되었습니다.");
                    break;
                } else {
                    System.out.println("비밀번호가 일치하지 않습니다.");
                }
            } else {
                System.out.println("입력하신 아이디가 존재하지 않습니다.");
            }
        }
    }
}
```

실행 결과

```
아이디와 비밀번호를 입력해주세요
아이디: sum'
비밀번호: 12

입력하신 아이디가 존재하지 않습니다.
아이디와 비밀번호를 입력해주세요
아이디: summer
비밀번호: 123

로그인 되었습니다.
```

15.4.3 Properties

: Hashtable의 하위 클래스이다. Hashtable은 키와 값을 다양한 타입으로 지정이 가능한데 Properties는 키와 값을 String 타입으로 제한한 컬렉션이다. Properties는 애플리케이션의 옵션 정보, 데이터베이스 연결 정보 그리고 국제화 정보가 저장된 파일을 읽을 때 사용된다.

- **프로퍼티 파일**

  : driver, url, uername, password는 키가 되고 그 뒤의 문자열은 값이 된다.

- **database.properties(키=값으로 구성된 프로퍼티)**

```
driver = oracle.jdbc.OracleDriver
url = jdbc:oracle:thin:@localhost:1521:orcl
username = scott
password = tiger
```

- **예제(프로퍼티 파일로부터 읽기)**

```
package properties;

import java.io.FileReader;
import java.net.URLDecoder;
import java.util.Properties;

public class PropertiesExample {
    public static void main(String[] args) throws Exception {
        // 프로퍼티 파일을 읽기 위해서는
        // Properties 객체를 생성하고,
        // load() 메소드를 호출하면 된다.
        Properties properties = new Properties();

        //프로퍼티 파일의 경로를 얻으려면 Class의
        // getResource() 메소드를 이용하면 된다.
        // 그리고 getPath() 메소드는
        // URL 파일의 절대 경로를 리턴한다.
        String path = PropertiesExample.class.getResource(
                "database.properties"
        ).getPath();
        path = URLDecoder.decode(path, "utf-8");
        properties.load(new FileReader(path));

        String driver = properties.getProperty("driver");
        String url = properties.getProperty("url");
        String username = properties.getProperty("username");
        String password = properties.getProperty("password");

        System.out.println("driver : " + driver);
        System.out.println("url : " + url);
        System.out.println("username : " + username);
        System.out.println("password : " + password);
    }
}
```

실행 결과

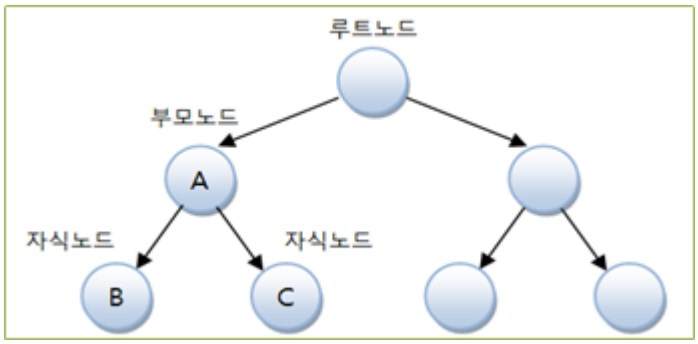
```
driver : oracle.jdbc.OracleDriver
url : jdbc:oracle:thin:@localhost:1521:orcl
username : scott
password : tiger
```

15.5 검색 기능을 강화시킨 컬렉션

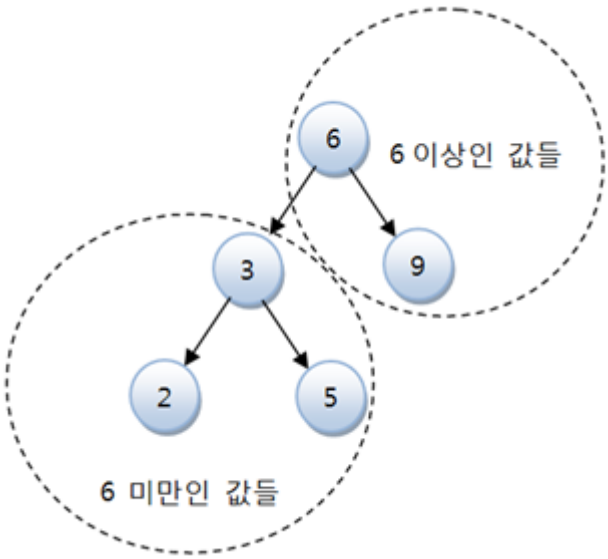
: 검색 기능을 강화시킨 TreeSet과 TreeMap을 제공하고 있다. TreeSet은 Set 컬렉션이고, TreeMap은 Map 컬렉션이다. 이 컬렉션들은 이진트리를 이용해서 계층적 구조(Tree 구조)를 가지면서 객체를 저장한다.

15.5.1 이진 트리 구조

: 여러 개의 노드(node)가 트리 형태로 연결된 구조. 각 노드에 최대 2개의 노드를 연결할 수 있는 구조를 가지고 있다.

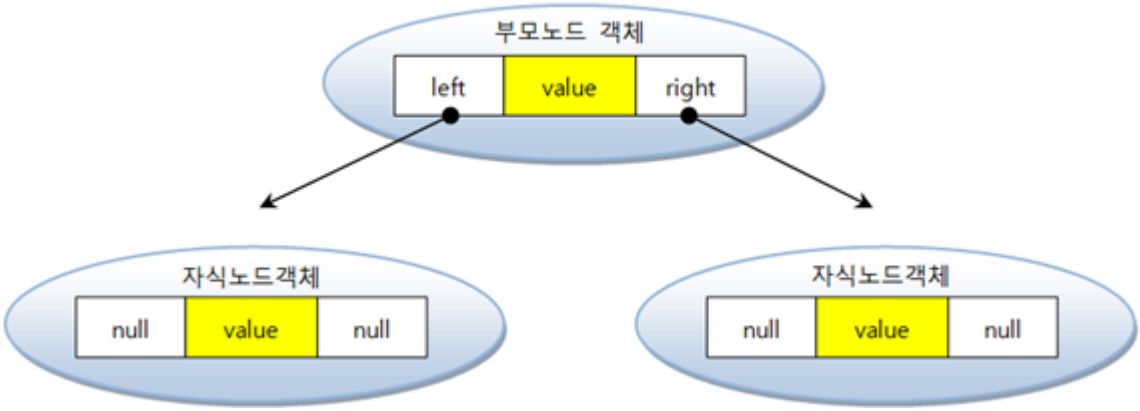


- **값들이 정렬되어 있어 그룹핑이 쉽다**



15.5.2 TreeSet

: TreeSet은 이진 트리를 기반으로한 Set 컬렉션이다.



> 부모값과 비교해서 낮은 것은 왼쪽, 높은 것은 오른쪽

- **TreeSet 기본 생성자 호출**

```
TreeSet<E> treeSet = new TreeSet<E>();
```

- **TreeSet의 검색 관련 메소드들**

리턴 타입	메소드	설명
E	firse()	제일 낮은 객체를 리턴
E	last()	제일 높은 객체를 리턴
E	lower(E e)	주어진 객체보다 바로 아래 객체를 리턴
E	higher(E e)	주어진 객체보다 바로 위 객체를 리턴
E	floor(E e)	주어진 객체와 동등한 객체가 있으면 리턴, 만약 없다면 주어진 객체의 바로 아래의 객체를 리턴
E	ceiling(E e)	주어진 객체와 동등한 객체가 있으면 리턴, 만약 없다면 주어진 객체의 바루 위의 객체를 리턴
E	pollFirse()	제일 낮은 객체를 꺼내오고 컬렉션에서 제거함
E	pollLast()	제일 높은 객체를 꺼내오고 컬렉션에서 제거함

- **예제(특정 객체 찾기)**

```
package treeset;

import java.util.TreeSet;

public class TreeSetExample1 {
    public static void main(String[] args) {
        TreeSet<Integer> scores = new TreeSet<Integer>();
        scores.add(new Integer(87));
        scores.add(new Integer(98));
```

```
        scores.add(new Integer(75));
        scores.add(new Integer(95));
        scores.add(new Integer(80));

        Integer score = null;

        score = scores.first();
        System.out.println("가장 낮은 점수: " + score);

        score = scores.last();
        System.out.println("가장 높은 점수: " + score + '\n');

        score = scores.lower(new Integer(95));
        System.out.println("95점 아래 점수: " + score);

        score = scores.higher(new Integer(95));
        System.out.println("95점 위의 점수: " + score + "\n");

        score = scores.floor(new Integer(95));
        System.out.println("95점 이거나 바로 아래 점수: " + score);

        score = scores.ceiling(new Integer(85));
        System.out.println("85점 이거나 바로 위의 점수: " + score + "\n");

        while(!scores.isEmpty()) {
            score = scores.pollFirst();
            System.out.println(score + "(남은 객체 수: " + scores.size() + ")");
        }
    }
}
```

실행 결과

```
가장 낮은 점수: 75
가장 높은 점수: 98

95점 아래 점수: 87
95점 위의 점수: 98

95점 이거나 바로 아래 점수: 95
85점 이거나 바로 위의 점수: 87

75(남은 객체 수: 4)
80(남은 객체 수: 3)
87(남은 객체 수: 2)
95(남은 객체 수: 1)
98(남은 객체 수: 0)
```

- **TreeSet의 정렬 메소드들**

리턴 타입	메소드	설명
Iterator<E>	descendingIterator()	내림차순으로 정렬된 Iterator를 리턴
NavigableSet<E>	descendingSet()	내림차순으로 정렬된 NavigableSet을 반환

- **예제**

```
package treeset;

import java.util.NavigableSet;
import java.util.TreeSet;

public class TreeSetExample2 {
    public static void main(String[] args) {
        TreeSet<Integer> scores = new TreeSet<Integer>();
        scores.add(new Integer(87));
        scores.add(new Integer(98));
        scores.add(new Integer(75));
        scores.add(new Integer(95));
        scores.add(new Integer(80));

        NavigableSet<Integer> descenginSet = scores.descendingSet();
        for(Integer score : descenginSet) {
            System.out.println(score + " ");
        }
        System.out.println();

        NavigableSet<Integer> ascendingSet = descenginSet.descendingSet();

        for(Integer score : ascendingSet) {
            System.out.println(score + " ");
        }
    }
}
```

실행 결과

```
98
95
87
80
75

75
80
87
95
98
```

- **TreeSet이 가지고 있는 범위 검색과 관련된 메소드들**

리턴 탕비	메소드	설명
NavigableSet<E>	headSet(E toElement, boolean inclusive)	주어진 객체보다 낮은 객체들을 NavigableSet으로 리턴, 주어진 객체 포함 여부는 두번째 매개값에 따라 달라짐
NavigableSet<E>	tailSet(E fromElement, boolean inclusive)	주어진 객체보다 높은 객체들을 NavigableSet으로 리턴, 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableSet<E>	subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	시작과 끝으로 주어진 객체 사이의 객체들을 NavigableSet으로 리턴, 시작과 끝 객체의 포함 여부는 두 번째, 네 번째 매개값에 따라 달라짐

- **예제(영어 단어를 정렬하고, 범위 검색해보기)**

```
package treeset;

import java.util.NavigableSet;
import java.util.TreeSet;

public class TreeSetExample3 {
    public static void main(String[] args) {
        TreeSet<String> treeSet = new TreeSet<String>();
        treeSet.add("apple");
        treeSet.add("forever");
        treeSet.add("description");
        treeSet.add("ever");
        treeSet.add("zoo");
        treeSet.add("base");
        treeSet.add("guess");
        treeSet.add("cherry");

        System.out.println("[c~f 사이의 단어 검색]");

        // "c" <= 검색 단어 <= "f"
        NavigableSet<String> rangeSet = treeSet.subSet(
                "c", true, "f", true
        );

        for(String word : rangeSet) {
            System.out.println(word);
        }
    }
}
```

실행 결과

```
[c~f 사이의 단어 검색]
cherry
description
ever
```

15.5.3 TreeMap

: TreeSet과의 차이점은 키와 값이 저장된 Map.Entry를 저장한다는 점이다. TreeMap에 객체를 저장하면 자동으로 정렬된다.

- **TreeMap 생성자 호출**

```
// K = 키 타입
// V = 값 타입
TreeMap<K, V> treeMap = new TreeMap<K, V>();
```

- **TreeMap이 가지고 있는 검색 관련 메소드들**

리턴 타입	메소드	설명
Map.Entry<K,V>	firstEntry()	제일 낮은 Map.Entry를 리턴
Map.Entry<K,V>	lastEntry()	제일 높은 Map.Entry를 리턴
Map.Entry<K,V>	lowerEntry(K key)	주어진 키보다 바로 아래 Map.Entry를 리턴
Map.Entry<K,V>	higherEntry(K key)	주어진 키보다 바로 위 Map.Entry를 리턴
Map.Entry<K,V>	floorEntry(K key)	주어진 키와 동등한 키가 있으면 해당 Map.Entry를 리턴, 없다면 주어진 키 바로 아래의 Map.Entry를 리턴
Map.Entry<K,V>	ceilingEntry(K key)	주어진 키와 동등한 키가 있으면 해당 Map.Entry를 리턴, 없다면 주어진 키 바로 위의 Map.Entry를 리턴
Map.Entry<K,V>	pollFirstEntry()	제일 낮은 Map.Entry를 꺼내오고 컬렉션에서 제거함
Map.Entry<K,V>	pollLastEntry()	제일 높은 Map.Entry를 꺼내오고 컬렉션에서 제거함.

- **예제(특정 Map.Entry 찾기)**

```
package treemap;
```

```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample1 {
    public static void main(String[] args) {
        TreeMap<Integer, String> scores = new TreeMap<
                Integer, String>();
        scores.put(new Integer(87), "홍길동");
        scores.put(new Integer(98), "이동수");
        scores.put(new Integer(75), "박길순");
        scores.put(new Integer(95), "신용권");
        scores.put(new Integer(80), "김자바");

        Map.Entry<Integer, String> entry = null;

        entry = scores.firstEntry();
        System.out.println("가장 낮은 점수: " + entry.getKey() +
                "-" + entry.getValue());

        entry = scores.lastEntry();
        System.out.println("가장 높은 점수: " + entry.getKey() +
                "-" + entry.getValue() + '\n');

        entry = scores.lowerEntry(new Integer(95));
        System.out.println("95점 아래 점수: " + entry.getKey() +
                "-" + entry.getValue());

        entry = scores.higherEntry(new Integer(95));
        System.out.println("95점 위의 점수: " + entry.getKey() +
                "-" + entry.getValue() + '\n');

        entry = scores.floorEntry(new Integer(95));
        System.out.println("95점 이거나 바로 아래 점수: " +
                entry.getKey() + "-" + entry.getValue());

        entry = scores.ceilingEntry(new Integer(85));
        System.out.println("85점 이거나 바로 위의 점수: " +
                entry.getKey() + "-" + entry.getValue() + "\n");

        while(!scores.isEmpty()) {
            entry = scores.pollFirstEntry();
            System.out.println(entry.getKey() + "-" + entry.getValue() +
                    "(남은 객체 수: " + scores.size() + ")");
        }
    }
}
```

실행 결과

```
가장 낮은 점수: 75-박길순
가장 높은 점수: 98-이동수

95점 아래 점수: 87-홍길동
```

```
95점 위의 점수: 98-이동수

95점 이거나 바로 아래 점수: 95-신용권
85점 이거나 바로 위의 점수: 87-홍길동

75-박길순(남은 객체 수: 4)
80-김자바(남은 객체 수: 3)
87-홍길동(남은 객체 수: 2)
95-신용권(남은 객체 수: 1)
98-이동수(남은 객체 수: 0)
```

- **TreeMap이 가지고 있는 정렬과 관련된 메소드들**

리턴 타입	메소드	설명
NavigableSet<K>	descendingKeySet()	내림차순으로 정렬된 키의 NavigableSet을 리턴
NavigableMap<K,V>	descendingMap()	내림차순으로 정렬된 Map.Entry의 NaVigableMap을 리턴

- **예제(객체 정렬하기)**

```
package treemap;

import java.util.*;

public class TreeMapExample2 {
    public static void main(String[] args) {
        TreeMap<Integer, String> scores = new TreeMap<
                Integer, String>();
        scores.put(new Integer(87), "홍길동");
        scores.put(new Integer(98), "이동수");
        scores.put(new Integer(75), "박길순");
        scores.put(new Integer(95), "신용권");
        scores.put(new Integer(80), "김자바");

        NavigableMap<Integer, String> descendingMap =
                scores.descendingMap();
        Set<Map.Entry<Integer, String>> descendingEntrySet =
                descendingMap.entrySet();
        for(Map.Entry<Integer, String> entry : descendingEntrySet) {
            System.out.print(entry.getKey() + "-" + entry.getValue());
        }
        System.out.println();

        NavigableMap<Integer, String> ascendingMap =
                descendingMap.descendingMap();
        Set<Map.Entry<Integer,String>> ascendingEntrySet =
                ascendingMap.entrySet();
```

```
        for(Map.Entry<Integer,String> entry : ascendingEntrySet) {
            System.out.print(entry.getKey() + "-" +
                    entry.getValue() + " ");
        }
    }
}
```

실행 결과

```
98-이동수95-신용권87-홍길동80-김자바75-박길순
75-박길순 80-김자바 87-홍길동 95-신용권 98-이동수
```

- **TreeMap이 가지고 있는 범위 검색과 관련된 메소드들**

리턴 타입	메소드	설명
NavigableMap<K,V>	heapMap(K toKey, boolean inclusive)	주어진 키보다 작은 Map.Entry들을 NavigableMap으로 리턴, 주어진 키의 Map.Entry 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableMap<K,V>	tailMap(K fromKey, boolean inclusive)	주어진 객체보다 높은 Map.Entry들을 NavigableMap으로 리턴, 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableMap<K,V>	subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	시작과 끝으로 주어진 키 사이의 Map.Entry들을 NavigableMap 컬렉션으로 반환, 시작과 끝 키의 Map.Entry 포함 여부는 두 번째, 네 번째 매개값에 따라 달라짐

- **예제(키로 정렬하고 범위 검색하기)**

```
package treemap;

import java.util.Map;
import java.util.NavigableMap;
import java.util.TreeMap;

public class TreeMapExample3 {
    public static void main(String[] args) {
        TreeMap<String,Integer> treeMap = new TreeMap<String, Integer>();
        treeMap.put("apple", new Integer(10));
        treeMap.put("forever", new Integer(60));
        treeMap.put("description", new Integer(40));
        treeMap.put("ever", new Integer(50));
        treeMap.put("zoo", new Integer(10));
        treeMap.put("base", new Integer(20));
```

```
        treeMap.put("guess", new Integer(70));
        treeMap.put("cherry", new Integer(30));

        System.out.println("[c~f 사이의 단어 검색]");
        NavigableMap<String, Integer> rangeMap =
                treeMap.subMap("c", true,
                        "f", true);
        for(Map.Entry<String, Integer> entry : rangeMap.entrySet()) {
            System.out.println(entry.getKey() + "-" +
                    entry.getValue() + "페이지");
        }
    }
}
```

실행 결과

```
[c~f 사이의 단어 검색]
cherry-30페이지
description-40페이지
ever-50페이지
```

15.5.4 Comparable과 Comparator

: Integer, Double, String은 모두 Comparable 인터페이스를 구현하고 있다. 사용자 정의 클래스도 Comparable을 구현한다면 자동 정렬이 가능하다.

- **compareTo() 메소드** : 사용자 정의 클래스에서 이 메소드를 오버라이딩하여 다음과 같이 리턴 값을 만들어 내야 한다.

리턴 타입	메소드	설명
int	compareTo(T o)	주어진 객체와 같으면 0을 리턴 주어진 객체보다 적으면 음수를 리턴 주어진 객체보다 크면 양수를 리턴

- **예제(사용자 정의 객체를 나이 순으로 정렬하기**

  Person.java(**Comparable 구현 클래스**)

```
package comparable;

public class Person implements Comparable<Person>{
    public String name;
    public int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    // compareTo 오버라이딩
    @Override
    public int compareTo(Person o) {
        if(age < o.age) return -1;
        else if(age == o.age) return 0;
        else return 1;
    }
}
```

ComparableExample.java

```
package comparable;

import java.util.Iterator;
import java.util.TreeSet;

public class ComparableExample {
    public static void main(String[] args) {
        TreeSet<Person> treeSet = new TreeSet<Person>();

        // 저장될 때 나이 순으로 정렬됨.
        treeSet.add(new Person("홍길동", 45));
        treeSet.add(new Person("감자바", 25));
        treeSet.add(new Person("박지원", 31));

        // 왼쪽 마지막 노드에서
        // 오른쪽 마지막 노드까지
        // 반복해서 가져오기
        // (오름차순)
        Iterator<Person> iterator = treeSet.iterator();
        while(iterator.hasNext()) {
            Person person = iterator.next();
            System.out.println(person.name + " : " + person.age);
        }
    }
}
```

실행 결과

```
감자바 : 25
박지원 : 31
홍길동 : 45
```

- **TreeSet과 TreeMap의 키 사용시 주의할 점**

  : Key 객체가 Comparable을 구현하고 있지 않을 경우에는 저장하는 순간 ClassCastException이 발생한다.

- **Comparable 비구현 객체 정렬 방법**

```
// 오름차순 정렬자 사용(AscendingComparator)
TreeSet<E> treeSet = new TreeSet<E>(new AscendingComparator());

// 내림차순 정렬자 사용(DescendingComparator)
TreeMap<K,V> treeMap = new TreeMap<K,V>(new DescendingComparator());
```

- **Comparator 인터페이스를 구현한 정렬자 객체의 메소드**

리턴 타입	메소드	설명
int	compare(T o1, To2)	o1과 o2가 동등하다면 0을 리턴 o1이 o2보다 앞에 오게 하려면 음수를 리턴 o1이 o2보다 뒤에 오게 하려면 양수를 리턴

- **예제(내림차순 정렬자)**

  Fruit.java(**Comparable을 구현하지 않은 클래스**)

```
package comparable;

public class Fruit {
    public String name;
    public int price;

    public Fruit(String nama, int price) {
        this.name = nama;
        this.price = price;
    }
}
```

  DescendingComparator.java(**Fruit의 내림차순 정렬자**)

```
package comparable;

import java.util.Comparator;

public class DescendingComparator implements Comparator<Fruit> {
    @Override
    public int compare(Fruit o1, Fruit o2) {
        if(o1.price < o2.price) return 1;
        else if(o1.price == o2.price) return 0;
        else return -1;
    }
}
```

  ComparatorExample.java(**내림차순 정렬자를 사용하는 TreeSet**)

```
package comparable;

import java.util.Iterator;
import java.util.TreeSet;
```

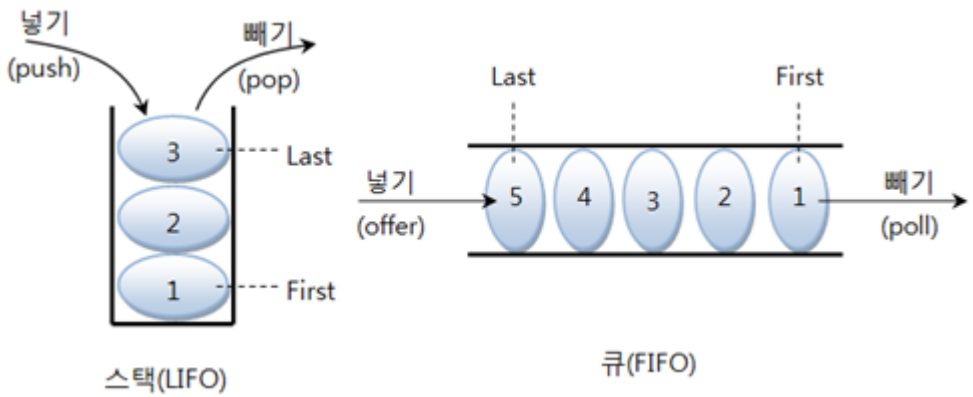
```
public class ComparatorExample {
    public static void main(String[] args) {
        /*
        TreeSet<Fruit> treeSet = new TreeSet<Fruit>();
        // Fruit 이 Comparable 을 구현하지
        // 않았기 때문에 예외 발생!
        treeSet.add(new Fruit("포도", 3000));
        treeSet.add(new Fruit("수박", 10000));
        treeSet.add(new Fruit("딸기", 6000));
        */

        // 내림차순 정렬자 제공
        TreeSet<Fruit> treeSet =
                new  TreeSet<Fruit>(new DescendingComparator());

        // 저장될 때 가격을 기준으로 내림차순 정렬됨
        treeSet.add(new Fruit("포도", 3000));
        treeSet.add(new Fruit("수박", 10000));
        treeSet.add(new Fruit("딸기", 6000));
        Iterator<Fruit> iterator = treeSet.iterator();
        while(iterator.hasNext()) {
            Fruit fruit = iterator.next();
            System.out.println(fruit.name + " : " + fruit.price);
        }
    }
}
```

실행 결과

```
수박 : 10000
딸기 : 6000
포도 : 3000
```

15.6 LIFO와 FIFO 컬렉션

- **LIFO(Last In First Out)** : 후입선출, 나중에 넣은 객체가 먼저 빠져나가는 자료구조
- **FIFO(First In First Out** : 선입선출, 먼저 넣은 객체가 먼저 빠져나가는 자료구조



15.6.1 Stack

- **Stack 클래스의 주요 메소드들**

리턴 타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣는다.
E	peek()	스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거하지 않는다.
E	pop()	스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거한다.

- **Stack 객체 생성**

```
Stack<E> stack = new Stack<E>();
```

- **예제(동전 케이스)**

  Coin.java(**동전 클래스**)

```
package stack;

public class Coin {
    private int value;

    public Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

  StackExample.java(**Stack을 이용한 동전케이스**)

```
package stack;

import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<Coin> coinBox = new Stack<Coin>();

        coinBox.push(new Coin(100));
        coinBox.push(new Coin(50));
        coinBox.push(new Coin(500));
        coinBox.push(new Coin(10));
```

```
        while(!coinBox.isEmpty()) {
            Coin coin = coinBox.pop();
            System.out.println("꺼내온 동전 : " + coin.getValue()
            + "원");
        }
    }
}
```

실행 결과

```
꺼내온 동전 : 10원
꺼내온 동전 : 500원
꺼내온 동전 : 50원
꺼내온 동전 : 100원
```

15.6.2 Queue

- **Queue 인터페이스의 메소드들**

리턴 타입	메소드	설명
boolean	offer(E e)	주어진 객체를 넣는다.
E	peek()	객체 하나를 가져온다. 객체를 큐에서 제거하지 않는다.
E	poll()	객체 하나를 가져온다. 객체를 큐에서 제거한다.

> Queue 인터페이스를 구현한 대표적인 클래스는 LinkedList이다.

- **Queue 객체 생성**

```
Queue<E> queue = new LinkedList<E>();
```

- **예제(메시지 큐 구현)**

  Message.java(**Message 클래스**)

```
package queue;

public class Message {
    public String command;
    public String to;

    public Message(String command, String to) {
        this.command = command;
        this.to = to;
    }
}
```

QueueExample.java(**Queue를 이용한 메시지 큐**)

```
package queue;

import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Message> messageQueue = new LinkedList<Message>();

        // 메시지 저장
        messageQueue.offer(
                new Message("sendMail", "홍길동")
        );
        messageQueue.offer(
                new Message("sendSMS", "신용권")
        );
        messageQueue.offer(
                new Message("sendKakaotalk", "홍두께")
        );

        // 메시지 큐가 비어있는지 확인
        while(!messageQueue.isEmpty()) {
            // 메시지 큐에서 한 개의 메시지 꺼냄
            Message message = messageQueue.poll();
            switch (message.command) {
                case "sendMail":
                    System.out.println(message.to +
                            "님에게 메일을 보냅니다.");
                    break;
                case "sendSMS":
                    System.out.println(message.to +
                            "님에게 SMS를 보냅니다.");
                    break;
                case "sendKakaotalk":
                    System.out.println(message.to +
                            "님에게 카카오톡을 보냅니다.");
                    break;
            }
```

```
        }
    }
}
```

실행 결과

```
홍길동님에게 메일을 보냅니다.
신용권님에게 SMS를 보냅니다.
홍두께님에게 카카오톡을 보냅니다.
```

15.7 동기화된 컬렉션

: Vector와 Hashtable은 동기화된(synchronized) 메소드로 구성되어 있기 때문에 멀티 스레드 환경에서 안전하게 요소를 처리할 수 있다. 하지만 ArrayList, HashSet, HashMap은 그렇지 않기 때문에 싱글 스레드 환경에서 사용하다가 멀티 스레드 환경으로 전달할 필요도 있을 것이다. 이런 경우를 대비해서 비동기화된 메소드를 동기화된 메소드로 래핑하는 Collection의 **synchronizedXXX() 메소드**를 제공하고 있다.

리턴 타입	메소드(매개변수)	설명
List<T>	synchronizedList(List<T> list)	List를 동기화된 List로 리턴
Map<K,V>	synchronizedMap(Map<K,V> m)	Map을 동기화된 Map으로 리턴
Set<T>	synchronizedSet(Set<T> s)	Set을 동기화된 Set으로 리턴

- **래핑하는 예시**

```
// ArrayList를 동기화된 List로
List<T> list = Collections.synchronizedList(new ArrayList<T>());

// HashSet을 동기화된 Set으로
Set<E> set = Collections.synchronizedSet(new HashSet<E>());

// HashMap을 동기화된 Map으로
Map<K, V> map = Collections.synchronizedMap(new HashMap<K, V>());
```

15.8 병렬 처리를 위한 컬렉션

: 멀티 스레드가 컬렉션의 요소를 병렬적으로 처리할 수 있도록 특별한 컬렉션을 제공하고 있다. java.util.concurrent 패키지의 ConcurrentHashMap과 ConcurrentLinkedQueue이다. ConcurrentHashMap은 Map 구현 클래스이고, ConcurrentLinkedQueue는 Queue 구현 클래스이다.

- **ConcurrentHashMap 객체 생성**

```
Map<K,V> map = new ConcurrentHashMap<K,V>();
```

- **ConcurrentLinkedQueue 객체 생성**

```
Queue<E> queue = new ConcurrentLinkedQueue<E>();
```