KB금융그룹 | 국민의 평생 금융파트너

2024년 상반기 K-디지털 트레이닝

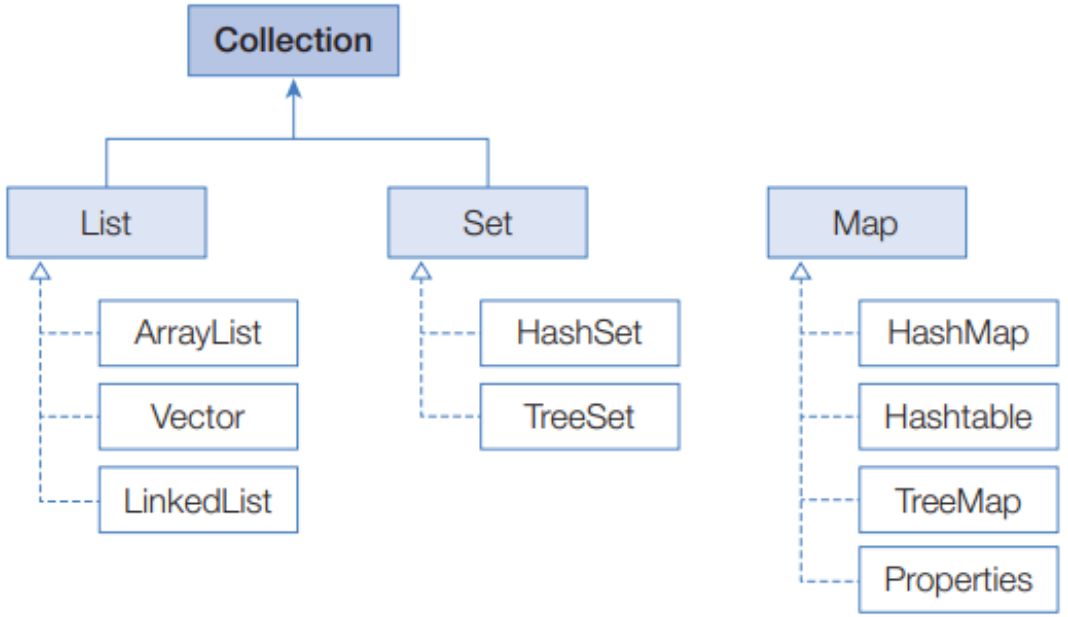
컬렉션 자료구조

[KB] IT's Your Life

KB 국민은행

컬렉션 프레임워크

- 널리 알려진 자료구조를 바탕으로 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 관련 인터페이스와 클래스들을 포함시켜 놓은 java.util 패키지
- 주요 인터페이스: List, Set, Map



인터페이스 분류		특징	구현 클래스
Collection	List	– 순서를 유지하고 저장 – 중복 저장 가능	ArrayList, Vector, LinkedList
	Set	– 순서를 유지하지 않고 저장 – 중복 저장 안됨	HashSet, TreeSet
Map		– 키와 값으로 구성된 엔트리 저장 – 키는 중복 저장 안됨	HashMap, Hashtable, TreeMap, Properties

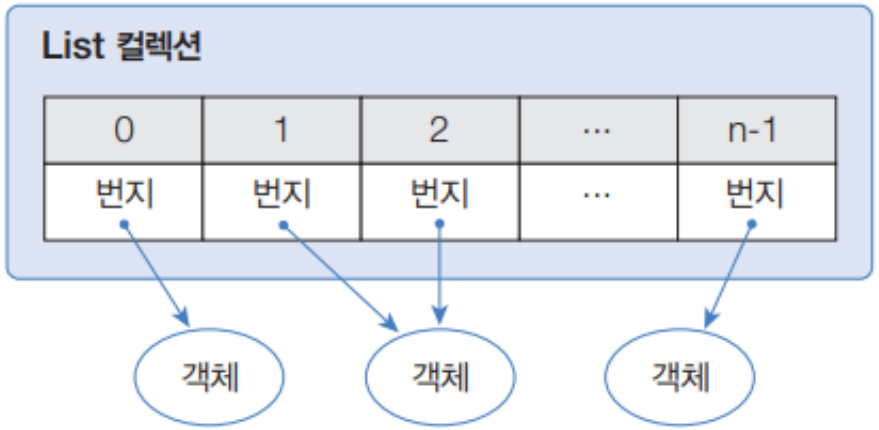
List 컬렉션

- 객체를 인덱스로 관리하기 때문에 객체를 저장하면 인덱스가 부여되고 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨 끝에 추가
	void add(int index, E element)	주어진 인덱스에 객체를 추가
	set(int index, E element)	주어진 인덱스의 객체를 새로운 객체로 바꿈
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	E get(int index)	주어진 인덱스에 저장된 객체를 리턴
	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

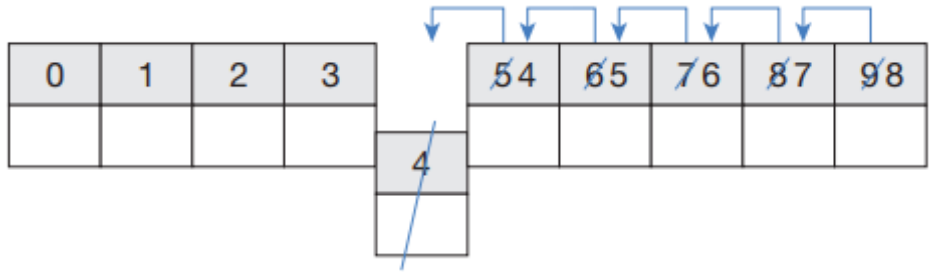
ArrayList

- ArrayList에 객체를 추가하면 내부 배열에 객체가 저장되고 제한 없이 객체를 추가할 수 있음
- 객체의 번지를 저장. 동일한 객체를 중복 저장 시 동일한 번지가 저장. null 저장 가능




```
List<E> list = new ArrayList<E>();    //E에 지정된 타입의 객체만 저장
List<E> list = new ArrayList<>();     //E에 지정된 타입의 객체만 저장
List list = new ArrayList();          //모든 타입의 객체를 저장
```

- ArrayList 컬렉션에 객체를 추가 시 인덱스 0번부터 차례대로 저장
- 특정 인덱스의 객체를 제거하거나 삽입하면 전체가 앞/뒤로 1씩 당겨지거나 밀림
- 빈번한 객체 삭제와 삽입이 일어나는 곳에선 바람직하지 않음



Board.java

```
package ch15.sec02.exam01;

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

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

  public String getSubject() { return subject; }
  public void setSubject(String subject) { this.subject = subject; }
  public String getContent() { return content; }
  public void setContent(String content) { this.content = content; }
  public String getWriter() { return writer; }
  public void setWriter(String writer) { this.writer = writer; }
}
```

ArrayListExample.java

```
package ch15.sec02.exam01;

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

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

    //객체 추가
    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"));

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

ArrayListExample.java

```
//특정 인덱스의 객체 가져오기
Board board = list.get(2);
System.out.println(board.getSubject() + "\t" + board.getContent() +
          "\t" + board.getWriter());
System.out.println();

//모든 객체를 하나씩 가져오기
for(int i=0; i<list.size(); i++) {
  Board b = list.get(i);
  System.out.println(b.getSubject() + "\t" + b.getContent() +
          "\t" + b.getWriter());
}
System.out.println();

//객체 삭제
list.remove(2);
list.remove(2);
```

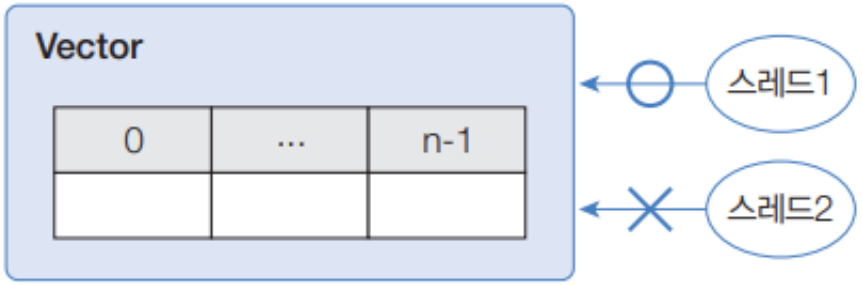
ArrayListExample.java

```
    //향상된 for문으로 모든 객체를 하나씩 가져오기
    for(Board b : list) {
      System.out.println(b.getSubject() + "\t" + b.getContent() +
              "\t" + b.getWriter());
    }
  }
}
```

```
총 객체 수: 5

제목3    내용3    글쓴이3

제목1    내용1    글쓴이1
제목2    내용2    글쓴이2
제목3    내용3    글쓴이3
제목4    내용4    글쓴이4
제목5    내용5    글쓴이5

제목1    내용1    글쓴이1
제목2    내용2    글쓴이2
제목5    내용5    글쓴이5
```

Vector

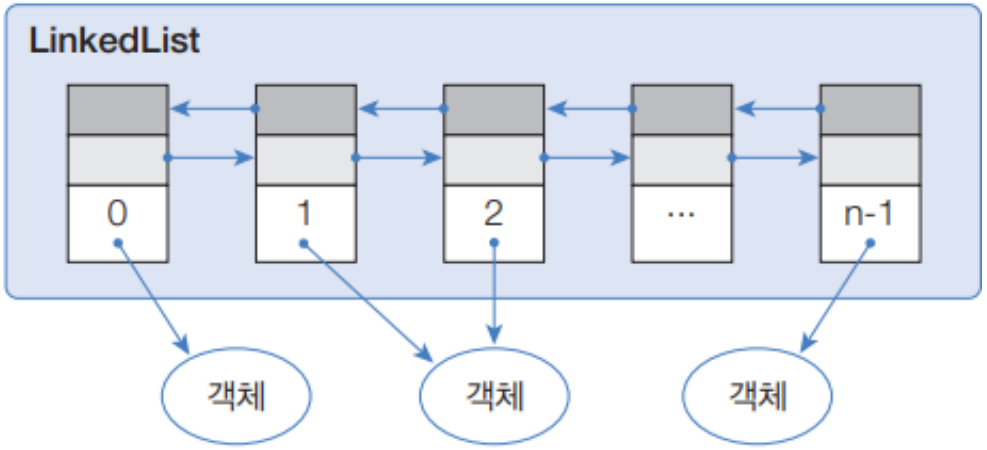
- 동기화된 메소드로 구성되어 있어 멀티 스레드가 동시에 Vector() 메소드를 실행할 수 없음
- 멀티 스레드 환경에서는 안전하게 객체를 추가 또는 삭제할 수 있음

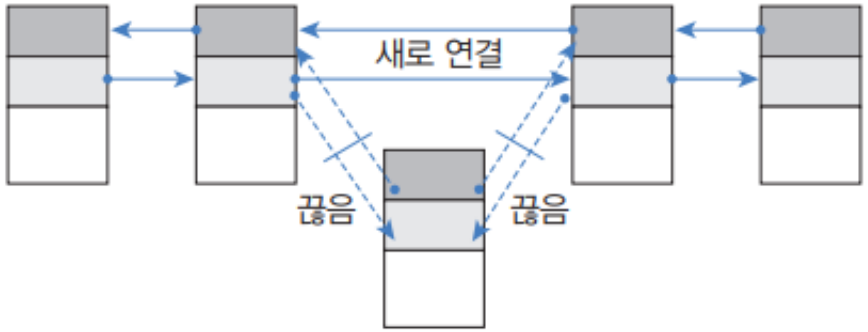



```
List<E> list = new Vector<E>();    //E에 지정된 타입의 객체만 저장
List<E> list = new Vector<>();     //E에 지정된 타입의 객체만 저장
List list = new Vector();          //모든 타입의 객체를 저장
```

LinkedList

- 인접 객체를 체인처럼 연결해서 관리. 객체 삭제와 삽입이 빈번한 곳에서 ArrayList보다 유리






```
List<E> list = new LinkedList<E>();    //E에 지정된 타입의 객체만 저장
List<E> list = new LinkedList<>();     //E에 지정된 타입의 객체만 저장
List list = new LinkedList();          //모든 타입의 객체를 저장
```

ArrayListExample.java

```
package ch15.sec02.exam03;

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

public class LinkedListExample {
  public static void main(String[] args) {
    //ArrayList 컬렉션 객체 생성
    List<String> list1 = new ArrayList<String>();

    //LinkedList 컬렉션 객체 생성
    List<String> list2 = new LinkedList<String>();

    //시작 시간과 끝 시간을 저장할 변수 선언
    long startTime;
    long endTime;
```

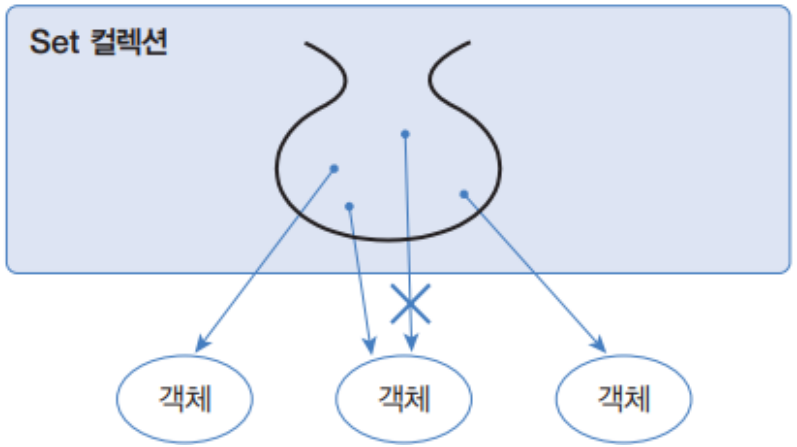
ArrayListExample.java

```
    //ArrayList 컬렉션에 저장하는 시간 측정
    startTime = System.nanoTime();
    for(int i=0; i<10000; i++) {
      list1.add(0, String.valueOf(i));
    }
    endTime = System.nanoTime();
    System.out.printf("%-17s %8d ns \n", "ArrayList 걸린 시간: ", (endTime-startTime) );

    //LinkedList 컬렉션에 저장하는 시간 측정
    startTime = System.nanoTime();
    for(int i=0; i<10000; i++) {
      list2.add(0, String.valueOf(i));
    }
    endTime = System.nanoTime();
    System.out.printf("%-17s %8d ns \n", "LinkedList 걸린 시간: ", (endTime-startTime) );
  }
}
```

```
ArrayList 걸린 시간:  14904300 ns
LinkedList 걸린 시간:   3235900 ns
```

Set 컬렉션

- Set 컬렉션은 저장 순서가 유지되지 않음
- 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있음(수학의 집합 개념)



기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 성공적으로 저장하면 true를 리턴하고 중복 객체면 false를 리턴
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	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

HashSetExample.java

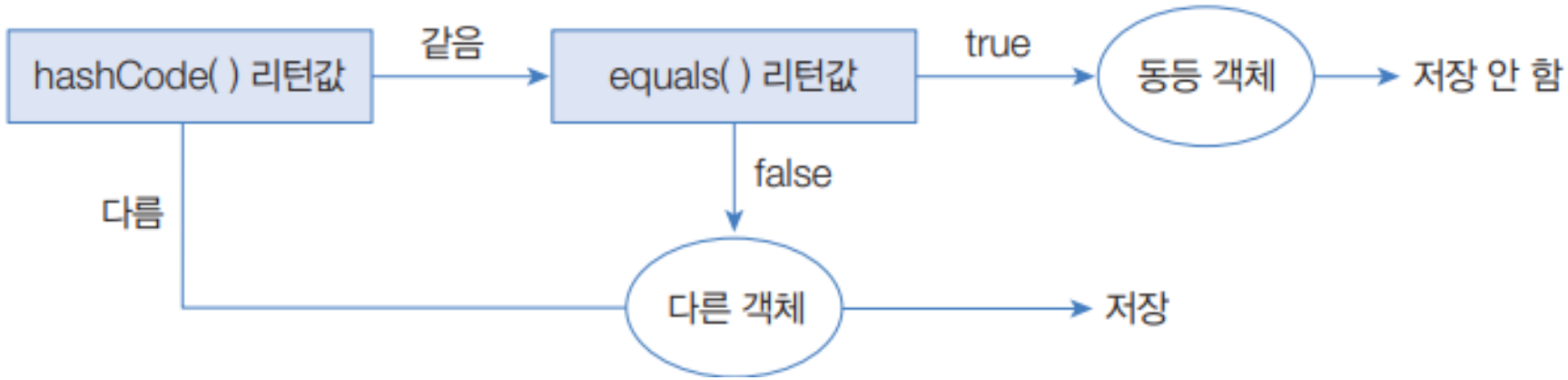
```
package ch15.sec03.exam01;

import java.util.*;

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

    //객체 저장
    set.add("Java");
    set.add("JDBC");
    set.add("Servlet/JSP");
    set.add("Java");     //<-- 중복 객체이므로 저장하지 않음
    set.add("iBATIS");

    //저장된 객체 수 출력
    int size = set.size();
    System.out.println("총 객체 수: " + size);
  }
}
```

```
총 객체 수: 4
```

HashSet

- 동등 객체를 중복 저장하지 않음
- 다른 객체라도 hashCode() 메소드의 리턴값이 같고, equals() 메소드가 true를 리턴하면 동일한 객체라고 판단하고 중복 저장하지 않음

```
Set<E> set = new HashSet<E>();    //E에 지정된 타입의 객체만 저장
Set<E> set = new HashSet<>();     //E에 지정된 타입의 객체만 저장
Set set = new HashSet();          //모든 타입의 객체를 저장
```



Member.java

```
package ch15.sec03.exam02;

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

  // 생성자 생략 ..

  //hashCode 재정의
  @Override
  public int hashCode() {
    return name.hashCode() + age;
  }

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

HashSetExample.java

```
package ch15.sec03.exam02;

import java.util.*;

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

    //Member 객체 저장
    set.add(new Member("홍길동", 30));
    set.add(new Member("홍길동", 30));

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

```
총 객체 수: 1
```

HashSet

- iterator() 메소드: 반복자를 얻어 Set 컬렉션의 객체를 하나씩 가져옴

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

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

```
while(iterator.hasNext()) {
  E e = iterator.next();
}
```

HashSetExample.java

```
package ch15.sec03.exam03;

import java.util.*;

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

    //객체 추가
    set.add("Java");
    set.add("JDBC");
    set.add("JSP");
    set.add("Spring");
```

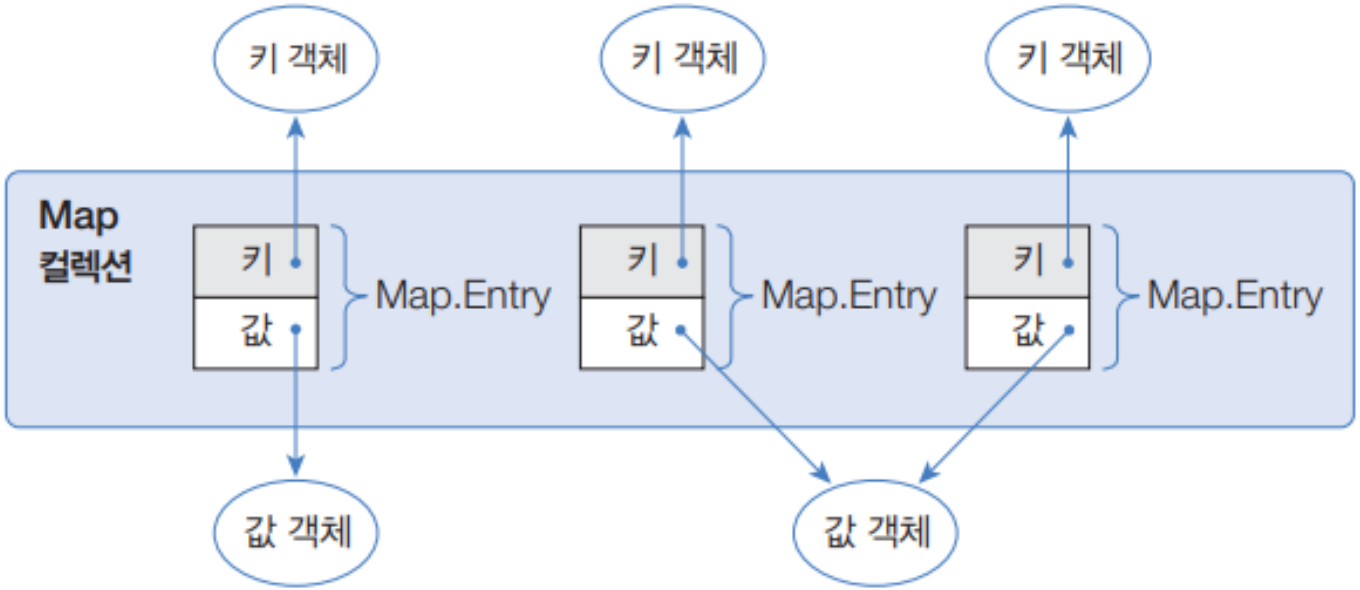
HashSetExample.java

```
    //객체를 하나씩 가져와서 처리
    Iterator<String> iterator = set.iterator();
    while(iterator.hasNext()) {
      //객체를 하나 가져오기
      String element = iterator.next();
      System.out.println( element);
      if(element.equals("JSP")) {
        //가져온 객체를 컬렉션에서 제거
        iterator.remove();
      }
    }
    System.out.println();

    //객체 제거
    set.remove("JDBC");

    //객체를 하나씩 가져와서 처리
    for(String element : set) {
      System.out.println(element);
    }
  }
}
```

```
Java
JSP
JDBC
Spring

Java
Spring
```

Map 컬렉션

- 키와 값으로 구성된 엔트리 객체를 저장
- 키는 중복 저장할 수 없지만 값은 중복 저장할 수 있음. 기존에 저장된 키와 동일한 키로 값을 저장하면 새로운 값으로 대치

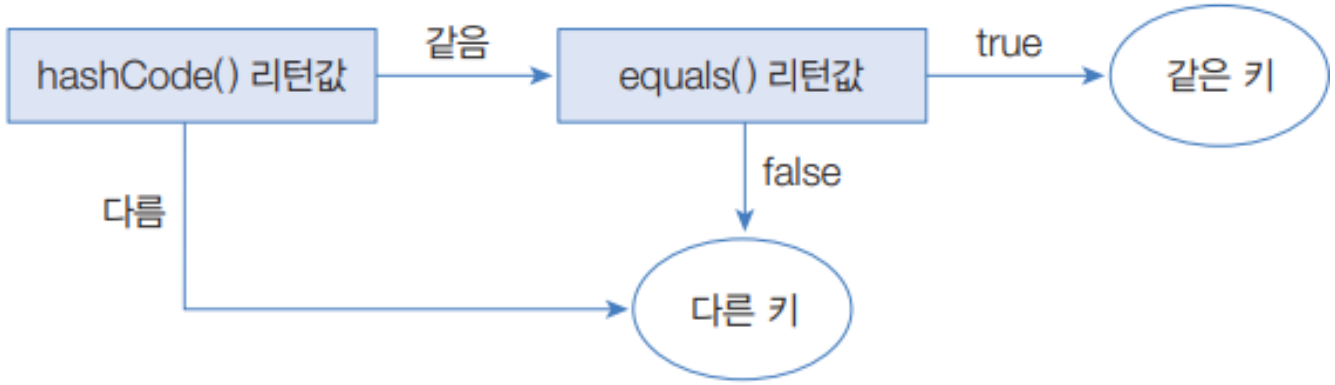


Map 컬렉션 주요 메소드

기능	메소드	설명
객체 추가	V put(K key, V value)	주어진 키와 값을 추가, 저장이 되면 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 키의 값을 리턴
	boolean isEmpty( )	컬렉션이 비어있는지 여부
	Set<K> keySet( )	모든 키를 Set 객체에 담아서 리턴
	int size( )	저장된 키의 총 수를 리턴
	Collection<V> values( )	저장된 모든 값 Collection에 담아서 리턴
객체 삭제	void clear( )	모든 Map.Entry(키와 값)를 삭제
	V remove(Object key)	주어진 키와 일치하는 Map.Entry 삭제, 삭제가 되면 값을 리턴

HashMap

- 키로 사용할 객체가 hashCode() 메소드의 리턴값이 같고 equals() 메소드가 true를 리턴할 경우 동일 키로 보고 중복 저장을 허용하지 않음




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

키 타입 값 타입 키 타입 값 타입

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

```
Map map = new HashMap();
```

HashMapExample.java

```
package ch15.sec04.exam01;

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

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

    //객체 저장
    map.put("신용권", 85);
    map.put("홍길동", 90);
    map.put("동장군", 80);
    map.put("홍길동", 95);
    System.out.println("총 Entry 수: " + map.size());
    System.out.println();
```

```
총 Entry 수: 3
```

HashMapExample.java

```
//키로 값 얻기
String key = "홍길동";
int value = map.get(key);
System.out.println(key + ": " + value);
System.out.println();

//키 Set 컬렉션을 얻고, 반복해서 키와 값을 얻기
Set<String> keySet = map.keySet();
Iterator<String> keyIterator = keySet.iterator();
while (keyIterator.hasNext()) {
  String k = keyIterator.next();
  Integer v = map.get(k);
  System.out.println(k + " : " + v);
}
System.out.println();
```

```
홍길동: 95

홍길동 : 95
신용권 : 85
동장군 : 80
```

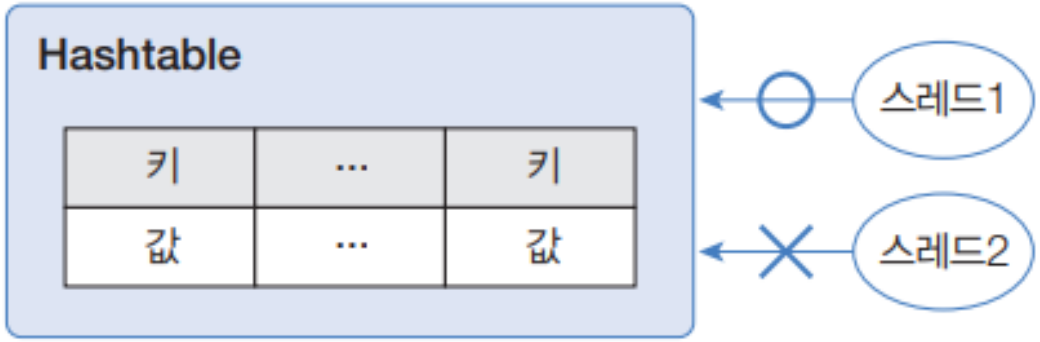
HashMapExample.java

```
    //엔트리 Set 컬렉션을 얻고, 반복해서 키와 값을 얻기
    Set<Entry<String, Integer>> entrySet = map.entrySet();
    Iterator<Entry<String, Integer>> entryIterator = entrySet.iterator();
    while (entryIterator.hasNext()) {
      Entry<String, Integer> entry = entryIterator.next();
      String k = entry.getKey();
      Integer v = entry.getValue();
      System.out.println(k + " : " + v);
    }
    System.out.println();

    //키로 엔트리 삭제
    map.remove("홍길동");

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

```
홍길동 : 95
신용권 : 85
동장군 : 80

총 Entry 수: 2
```

Hashtable

- 동기화된 메소드로 구성되어 있어 멀티 스레드가 동시에 Hashtable의 메소드들을 실행 불가
- 멀티 스레드 환경에서도 안전하게 객체를 추가, 삭제할 수 있다.




```
Map<String, Integer> map = new Hashtable<String, Integer>();
Map<String, Integer> map = new Hashtable<>();
```

Properties

- Properties는 Hashtable의 자식 클래스. 키와 값을 String 타입으로 제한한 컬렉션
- 주로 **확장자가 .properties인 프로퍼티 파일**을 읽을 때 사용
- 프로퍼티 파일은 **키와 값이 = 기호로 연결된** 텍스트 파일
- Properties 객체를 생성하고, load() 메소드로 프로퍼티 파일의 내용을 메모리로 로드

```
driver=oracle.jdbc.OracleDirver
url=jdbc:oracle:thin:@localhost:1521:orcl
username=scott
password=tiger
admin=\uD64D\uAE38\uB3D9
```

```
Properties properties  = new Properties();
properties.load(Xxx.class.getResourceAsStream("database.properties"));
```

database.properties

```
driver=oracle.jdbc.OracleDirver
url=jdbc:oracle:thin:@localhost:1521:orcl
username=scott
password=tiger
admin=\uD64D\uAE38\uB3D9
```

PropertiesExample.java

```
package ch15.sec04.exam03;

import java.util.Properties;

public class PropertiesExample {
  public static void main(String[] args) throws Exception {
    //Properties 컬렉션 생성
    Properties properties = new Properties();

    //PropertiesExample.class와 동일한 ClassPath에 있는 database.properties 파일 로드
    properties.load(PropertiesExample.class.getResourceAsStream("database.properties"));

    //주어진 키에 대한 값 읽기
    String driver = properties.getProperty("driver");
    String url = properties.getProperty("url");
    String username = properties.getProperty("username");
    String password = properties.getProperty("password");
    String admin = properties.getProperty("admin");
```

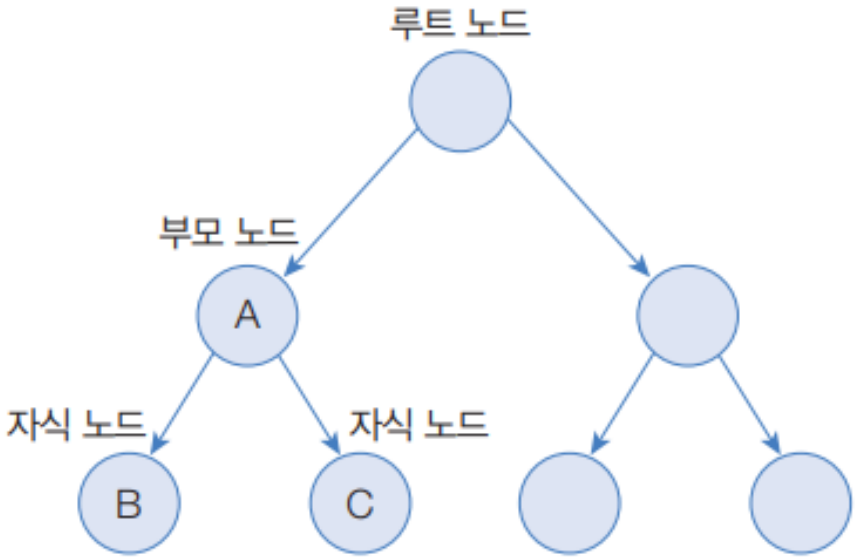
PropertiesExample.java

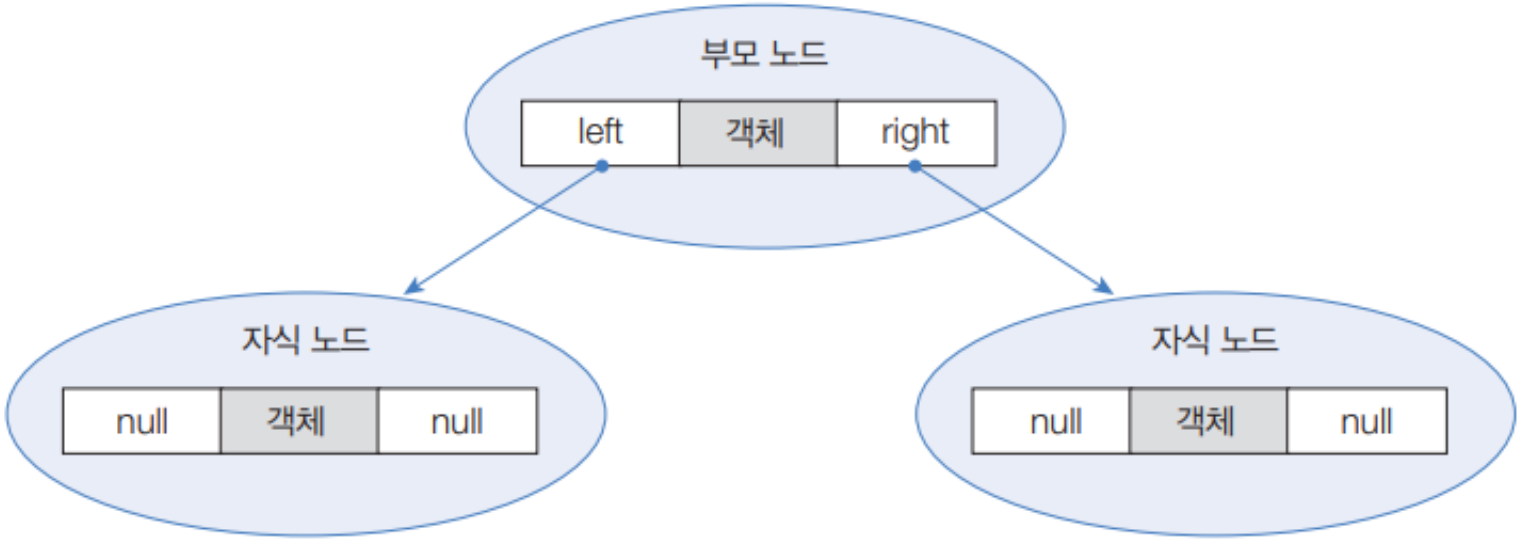
```
    //값 출력
    System.out.println("driver : " + driver);
    System.out.println("url : " + url);
    System.out.println("username : " + username);
    System.out.println("password : " + password);
    System.out.println("admin : " + admin);
  }
}
```

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

TreeSet

- 이진 트리를 기반으로 한 Set 컬렉션
- 여러 개의 노드가 트리 형태로 연결된 구조.
  루트 노드에서 시작해 각 노드에 최대 2개의 노드를 연결할 수 있음
- TreeSet에 객체를 저장하면 부모 노드의 객체와 비교해서 낮은 것은 왼쪽 자식 노드에, 높은 것은 오른쪽 자식 노드에 저장





TreeSet 컬렉션을 생성하는 방법

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

- Set 타입 변수에 대입해도 되지만 TreeSet 타입으로 대입한 이유는
  검색 관련 메소드가 TreeSet에만 정의되어 있기 때문

TreeSet 컬렉션 메소드

리턴 타입	메소드	설명
E	first( )	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 바로 위의 객체를 리턴
E	pollFirst( )	제일 낮은 객체를 꺼내오고 컬렉션에서 제거함
E	pollLast( )	제일 높은 객체를 꺼내오고 컬렉션에서 제거함
Iterator<E>	descendingIterator( )	내림차순으로 정렬된 Iterator를 리턴
NavigableSet<E>	descendingSet( )	내림차순으로 정렬된 NavigableSet을 리턴

TreeSet 컬렉션 메서드

리턴 타입	메소드	설명
NavigableSet<E>	headSet(E toElement, boolean inclusive)	주어진 객체보다 낮은 객체들을 NavigableSet으로 리턴. 주어진 객체 포함 여부는 두 번째 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

TreeSetExample.java

```
package ch15.sec05.exam01;

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

public class TreeSetExample {
  public static void main(String[] args) {
    //TreeSet 컬렉션 생성
    TreeSet<Integer> scores = new TreeSet< >();

    //Integer 객체 저장
    scores.add(87);
    scores.add(98);
    scores.add(75);
    scores.add(95);
    scores.add(80);

    //정렬된 Integer 객체를 하나씩 가져오기
    for(Integer s : scores) {
      System.out.print(s + " ");
    }
    System.out.println("\n");
```

```
75 80 87 95 98
```

TreeSetExample.java

```
//특정 Integer 객체를 가져오기
System.out.println("가장 낮은 점수: " + scores.first());
System.out.println("가장 높은 점수: " + scores.last());
System.out.println("95점 아래 점수: " + scores.lower(95));
System.out.println("95점 위의 점수: " + scores.higher(95));
System.out.println("95점이거나 바로 아래 점수: " + scores.floor(95));
System.out.println("85점이거나 바로 위의 점수: " + scores.ceiling(85) + "\n");

//내림차순으로 정렬하기
NavigableSet<Integer> descendingScores = scores.descendingSet();
for(Integer s : descendingScores) {
  System.out.print(s + " ");
}
System.out.println("\n");

//범위 검색( 80 <= )
NavigableSet<Integer> rangeSet = scores.tailSet(80, true);
for(Integer s : rangeSet) {
  System.out.print(s + " ");
}
System.out.println("\n");
```

```
가장 낮은 점수: 75
가장 높은 점수: 98
95점 아래 점수: 87
95점 위의 점수: 98
95점이거나 바로 아래 점수: 95
85점이거나 바로 위의 점수: 87

98 95 87 80 75

80 87 95 98
```

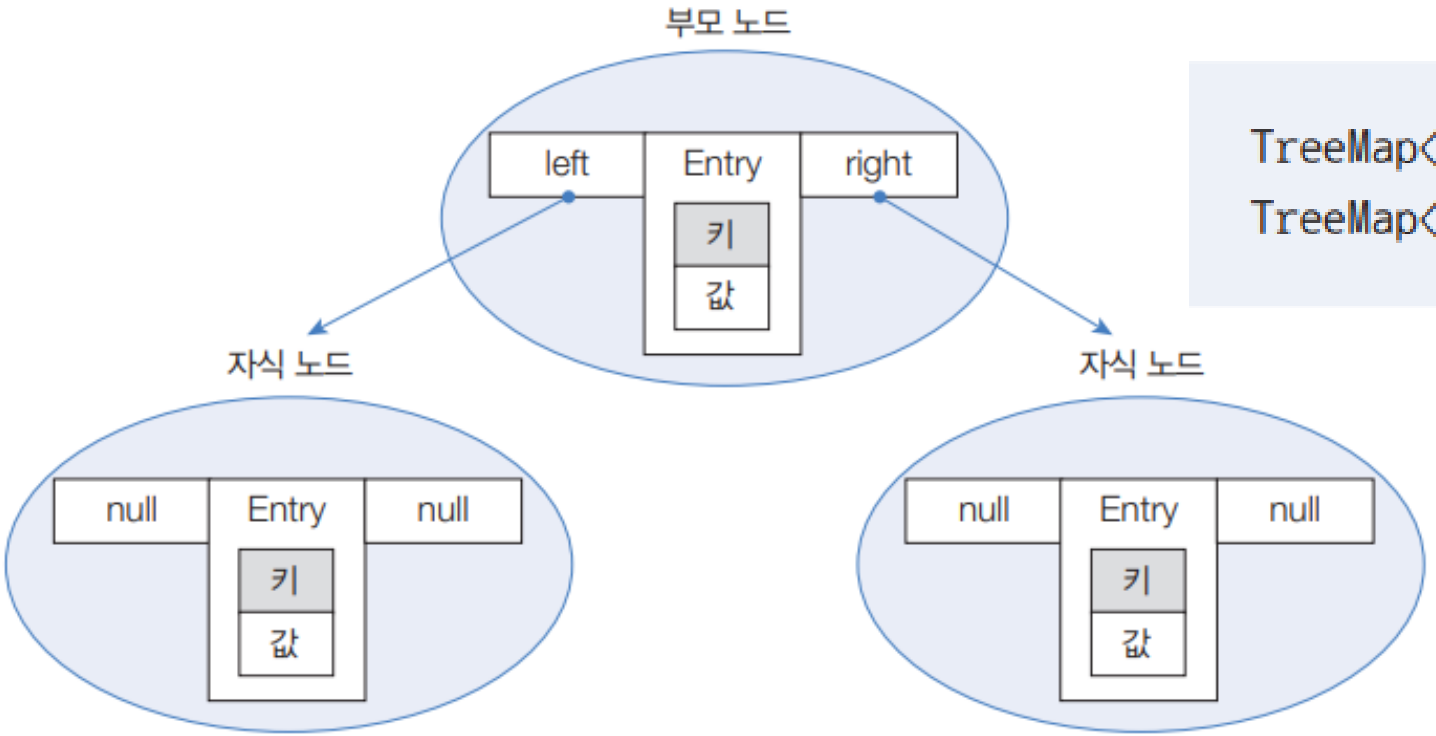
TreeSetExample.java

```
    //범위 검색( 80 <= score < 90 )
    rangeSet = scores.subSet(80, true, 90, false);
    for(Integer s : rangeSet) {
      System.out.print(s + " ");
    }
  }
}
```

```
80 87
```

TreeMap

- 이진 트리를 기반으로 한 Map 컬렉션. 키와 값이 저장된 엔트리 저장
- 부모 키 값과 비교해서 낮은 것은 왼쪽, 높은 것은 오른쪽 자식 노드에 Entry 객체를 저장




```
TreeMap<K, V> treeMap = new TreeMap<K, V>();
TreeMap<K, V> treeMap = new TreeMap<>();
```

검색 기능을 강화시킨 컬렉션

TreeMap 메서드

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
NavigableSet<K>	descendingKeySet( )	내림차순으로 정렬된 키의 NavigableSet을 리턴
NavigableMap<K,V>	descendingMap( )	내림차순으로 정렬된 Map.Entry의 NavigableMap을 리턴

TreeMap 메서드

리턴 타입	메소드	설명
NavigableMap<K,V>	headMap(K toKey, boolean inclusive)	주어진 키보다 낮은 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 NavigableSet으로 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

TreeMapExample.java

```
package ch15.sec05.exam02;

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

public class TreeMapExample {
  public static void main(String[] args) {
    //TreeMap 컬렉션 생성
    TreeMap<String,Integer> treeMap = new TreeMap< >();

    //엔트리 저장
    treeMap.put("apple", 10);
    treeMap.put("forever", 60);
    treeMap.put("description", 40);
    treeMap.put("ever", 50);
    treeMap.put("zoo", 80);
    treeMap.put("base", 20);
    treeMap.put("guess", 70);
    treeMap.put("cherry", 30);
```

TreeMapExample.java

```
//정렬된 엔트리를 하나씩 가져오기
Set<Entry<String, Integer>> entrySet = treeMap.entrySet();
for(Entry<String, Integer> entry : entrySet) {
  System.out.println(entry.getKey() + "-" + entry.getValue());
}
System.out.println();

//특정 키에 대한 값 가져오기
Entry<String,Integer> entry = null;
entry = treeMap.firstEntry();
System.out.println("제일 앞 단어: " + entry.getKey() + "-" + entry.getValue());
entry = treeMap.lastEntry();
System.out.println("제일 뒤 단어: " + entry.getKey() + "-" + entry.getValue());
entry = treeMap.lowerEntry("ever");
System.out.println("ever 앞 단어: " + entry.getKey() + "-" + entry.getValue() + "\n");
```

```
apple-10
base-20
cherry-30
description-40
ever-50
forever-60
guess-70
zoo-80

제일 앞 단어: apple-10
제일 뒤 단어: zoo-80
ever 앞 단어: description-40
```

TreeMapExample.java

```
    //내림차순으로 정렬하기
    NavigableMap<String,Integer> descendingMap = treeMap.descendingMap();
    Set<Entry<String,Integer>> descendingEntrySet = descendingMap.entrySet();
    for(Entry<String,Integer> e : descendingEntrySet) {
      System.out.println(e.getKey() + "-" + e.getValue());
    }
    System.out.println();

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

```
zoo-80
guess-70
forever-60
ever-50
description-40
cherry-30
base-20
apple-10

[c~h 사이의 단어 검색]
cherry-30
description-40
ever-50
forever-60
guess-70
```

검색 기능을 강화시킨 컬렉션

Comparable과 Comparator

- TreeSet에 저장되는 객체와 TreeMap에 저장되는 키 객체를 정렬
- Comparable 인터페이스에는 compareTo() 메소드가 정의. 사용자 정의 클래스에서 이 메소드를 재정의해서 비교 결과를 정수 값으로 리턴

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

Person.java

```
package ch15.sec05.exam03;

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

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

  @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 ch15.sec05.exam03;

import java.util.TreeSet;

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

    //객체 저장
    treeSet.add(new Person("홍길동", 45));
    treeSet.add(new Person("감자바", 25));
    treeSet.add(new Person("박지원", 31));

    //객체를 하나씩 가져오기
    for(Person person : treeSet) {
      System.out.println(person.name + ":" + person.age);
    }
  }
}
```

```
감자바:25
박지원:31
홍길동:45
```

Comparable과 Comparator

- 비교 기능이 없는 Comparable 비구현 객체를 저장하려면 비교자 Comparator를 제공
- 비교자는 compare() 메소드를 재정의해서 비교 결과를 정수 값으로 리턴

```
TreeSet<E> treeSet = new TreeSet<E>( new ComparatorImpl() );
                                     ↓
                                   비교자
                                     ↓
TreeMap<K,V> treeMap = new TreeMap<K,V>( new ComparatorImpl() );
```

리턴 타입	메소드	설명
int	compare(T o1, T o2)	o1과 o2가 동등하다면 0을 리턴 o1이 o2보다 앞에 오게 하려면 음수를 리턴 o1이 o2보다 뒤에 오게 하려면 양수를 리턴

Fruit.java

```
package ch15.sec05.exam04;

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

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

FruitComparator.java

```
package ch15.sec05.exam04;

import java.util.Comparator;

public class FruitComparator 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;
  }
}
```

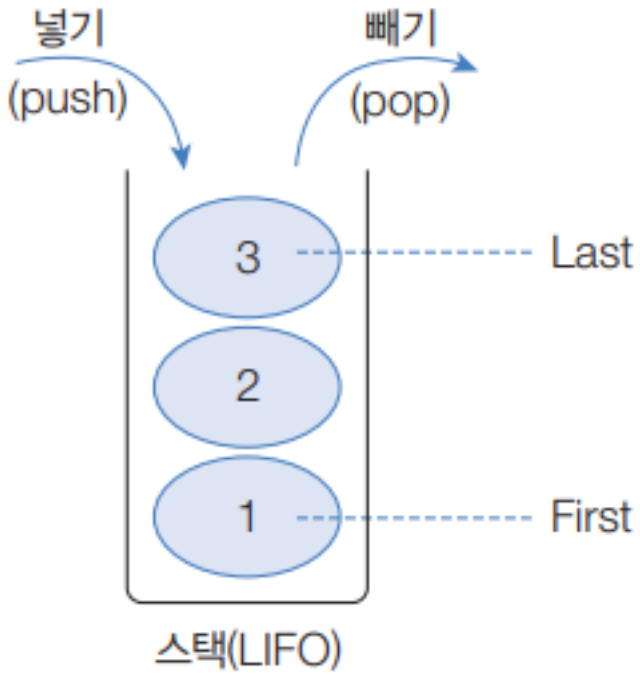
ComparatorExample.java

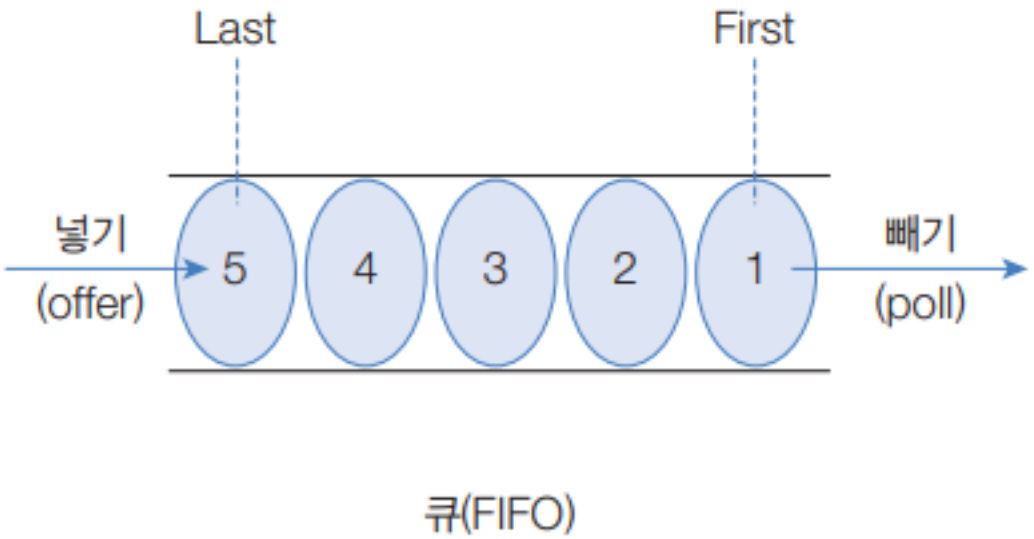
```
package ch15.sec05.exam04;

import java.util.TreeSet;

public class ComparatorExample {
  public static void main(String[] args) {
    //비교자를 제공한 TreeSet 컬렉션 생성
    TreeSet<Fruit> treeSet = new TreeSet<Fruit>(new FruitComparator());

    //객체 저장
    treeSet.add(new Fruit("포도", 3000));
    treeSet.add(new Fruit("수박", 10000));
    treeSet.add(new Fruit("딸기", 6000));

    //객체를 하나씩 가져오기
    for(Fruit fruit : treeSet) {
      System.out.println(fruit.name + ":" + fruit.price);
    }
  }
}
```

```
포도:3000
딸기:6000
수박:10000
```

후입선출과 선입선출

- 후입선출(LIFO): 나중에 넣은 객체가 먼저 빠져나가는 구조
- 선입선출(FIFO): 먼저 넣은 객체가 먼저 빠져나가는 구조
- 컬렉션 프레임워크는 LIFO 자료구조를 제공하는 스택 클래스와 FIFO 자료구조를 제공하는 큐 인터페이스를 제공





Stack

- Stack 클래스: LIFO 자료구조를 구현한 클래스

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

리턴 타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣는다.
E	pop( )	스택의 맨 위 객체를 빼낸다.

Coin.java

```
package ch15.sec06.exam01;

public class Coin {
  private int value;

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

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

StackExample.java

```
package ch15.sec06.exam01;

import java.util.Stack;

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

    //동전 넣기
    coinBox.push(new Coin(100));
    coinBox.push(new Coin(50));
    coinBox.push(new Coin(500));
    coinBox.push(new Coin(10));

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

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

Queue

- Queue 인터페이스: FIFO 자료구조에서 사용되는 메소드를 정의
- LinkedList: Queue 인터페이스를 구현한 대표적인 클래스

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

리턴 타입	메소드	설명
boolean	offer(E e)	주어진 객체를 큐에 넣는다.
E	poll( )	큐에서 객체를 빼낸다.

Message.java

```
package ch15.sec06.exam02;

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

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

QueueExample.java

```
package ch15.sec06.exam02;

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

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

    //메시지 넣기
    messageQueue.offer(new Message("sendMail", "홍길동"));
    messageQueue.offer(new Message("sendSMS", "신용권"));
    messageQueue.offer(new Message("sendKakaotalk", "감자바"));
```

LIFO와 FIFO 컬렉션

QueueExample.java

```
    //메시지를 하나씩 꺼내어 처리
    while(!messageQueue.isEmpty()) {
      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 카카오톡를 보냅니다.");
          break;
      }
    }
  }
}
```

```
홍길동님에게 메일을 보냅니다.
신용권님에게 SMS를 보냅니다.
감자바님에게 카카오톡를 보냅니다.
```

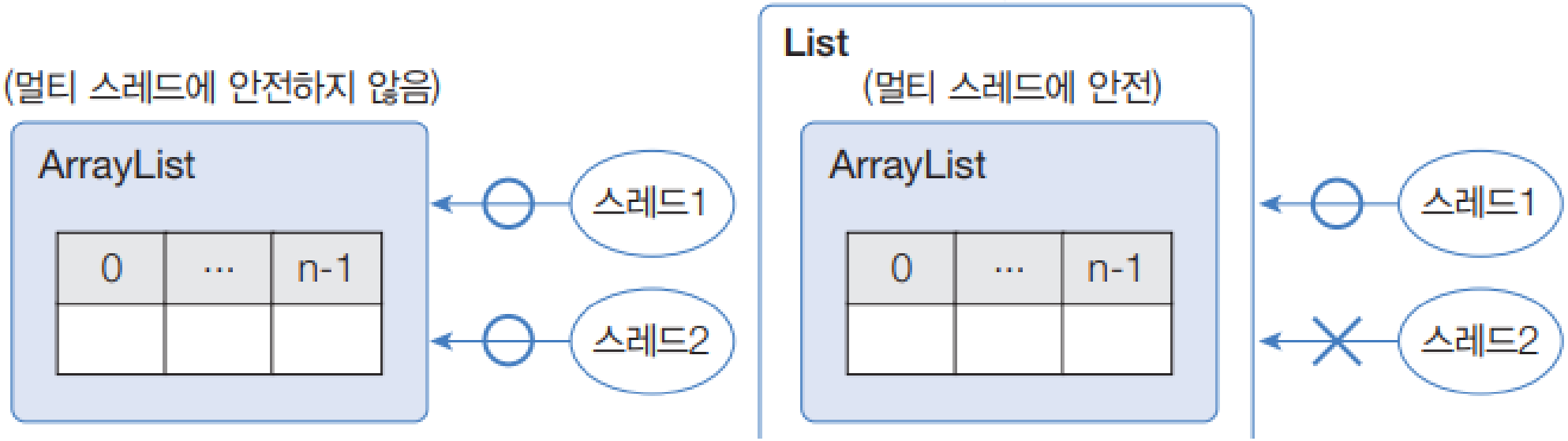
동기화된 컬렉션

- 동기화된 메소드로 구성된 Vector와 Hashtable는 멀티 스레드 환경에서 안전하게 요소를 처리
- Collections의 synchronizedXXX() 메소드:
  ArrayList, HashSet, HashMap 등 비동기화된 메소드를 동기화된 메소드로 래핑

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

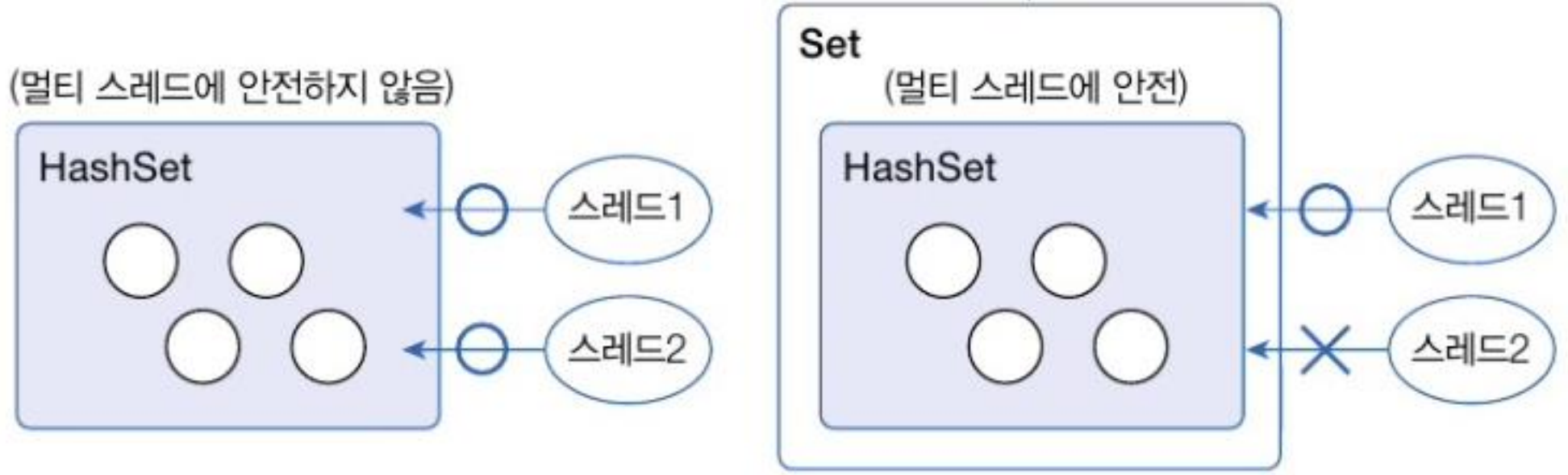
동기화된 List 컬렉션

```
List<T> list = Collections.synchronizedList(new ArrayList<T>());
```



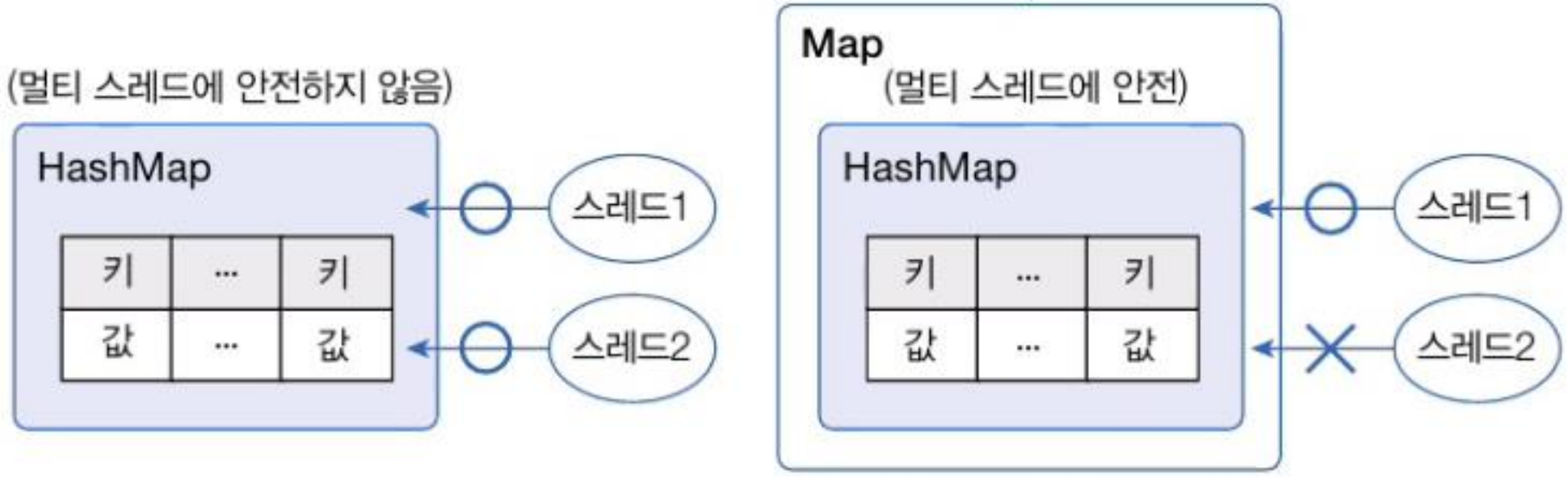
동기화된 Set 컬렉션

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



동기화된 Map 컬렉션

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



SynchronizedMapExample.java

```
package ch15.sec07;

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

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

    //작업 스레드 객체 생성
    Thread threadA = new Thread() {
      @Override
      public void run() {
        //객체 1000개 추가
        for(int i=1; i<=1000; i++) {
          map.put(i, "내용"+i);
        }
      }
    };
```

SynchronizedMapExample.java

```
//작업 스레드 객체 생성
Thread threadB = new Thread() {
  @Override
  public void run() {
    //객체 1000개 추가
    for(int i=1001; i<=2000; i++) {
      map.put(i, "내용"+i);
    }
  }
};

//작업 스레드 실행
threadA.start();
threadB.start();
```

SynchronizedMapExample.java

```
    //작업 스레드들이 모두 종료될 때까지 메인 스레드를 기다리게 함
    try {
      threadA.join();
      threadB.join();
    } catch(Exception e) {
    }

    //저장된 총 객체 수 얻기
    int size = map.size();
    System.out.println("총 객체 수: " + size);
    System.out.println();
  }
}
```

```
총 객체 수: 2000
```

수정할 수 없는 컬렉션

- 요소를 추가, 삭제할 수 없는 컬렉션. 컬렉션 생성 시 저장된 요소를 변경하고 싶지 않을 때 유용
- List, Set, Map 인터페이스의 정적 메소드인 of()로 생성
- List, Set, Map 인터페이스의 정적 메소드인 copyOf()을 이용해 기존 컬렉션을 복사
- 배열로부터 수정할 수 없는 List 컬렉션을 만듦

```
List<E> immutableList = List.of(E... elements);
Set<E> immutableSet = Set.of(E... elements);
Map<K,V> immutaleMap = Map.of( K k1, V v1,  K k2, V v2,  … );
```

```
List<E> immutableList = List.copyOf(Collection<E> coll);
Set<E> immutableSet = Set.copyOf(Collection<E> coll);
Map<K,V> immutaleMap = Map.copyOf(Map<K,V> map);
```

```
String[] arr = { "A", "B", "C" };
List<String> immutableList = Arrays.asList(arr);
```

ImmutableExample.java

```
package ch15.sec08;
…

public class ImmutableExample {
  public static void main(String[] args) {
    //List 불변 컬렉션 생성
    List<String> immutableList1 = List.of("A", "B", "C");
    //immutableList1.add("D"); (x)

    //Set 불변 컬렉션 생성
    Set<String> immutableSet1 = Set.of("A", "B", "C");
    //immutableSet1.remove("A"); (x)

    //Map 불변 컬렉션 생성
    Map<Integer, String> immutableMap1 = Map.of(
        1, "A",
        2, "B",
        3, "C"
        );
    //immutableMap1.put(4, "D"); (x)
```

ImmutableExample.java

```
//List 컬렉션을 불변 컬렉션으로 복사
List<String> list = new ArrayList< >();
list.add("A");
list.add("B");
list.add("C");
List<String> immutableList2 = List.copyOf(list);

//Set 컬렉션을 불변 컬렉션으로 복사
Set<String> set= new HashSet< >();
set.add("A");
set.add("B");
set.add("C");
Set<String> immutableSet2 = Set.copyOf(set);

//Map 컬렉션을 불변 컬렉션으로 복사
Map<Integer, String> map = new HashMap< >();
map.put(1, "A");
map.put(2, "B");
map.put(3, "C");
Map<Integer, String> immutableMap2 = Map.copyOf(map);
```

ImmutableExample.java

```
    //배열로부터 List 불변 컬렉션 생성
    String[] arr = { "A", "B", "C" };
    List<String> immutableList3 = Arrays.asList(arr);
  }
}
```