15장. 컬렉션 프레임워크

Contents

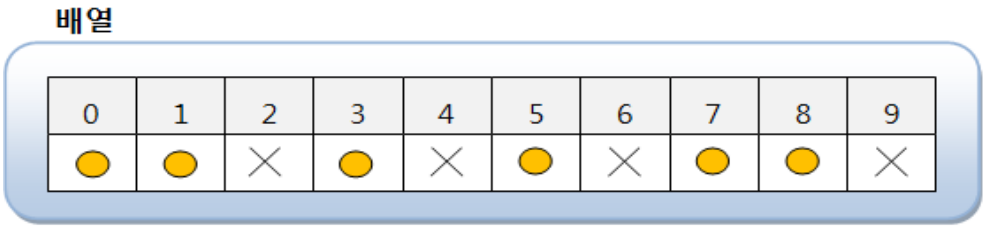
1절. 컬렉션 프레임워크 소개

❖ 컬렉션 프레임워크(Collection Framework)

- 컬렉션
  - 사전적 의미로 요소(객체)를 수집해 저장하는 것

- 배열의 문제점
  - 저장할 수 있는 객체 수가 배열을 생성할 때 결정
    → 불특정 다수의 객체를 저장하기에는 문제

  - 객체 삭제했을 때 해당 인덱스가 비게 됨
    → 낱알 빠진 옥수수 같은 배열
    → 객체를 저장하려면 어디가 비어있는지 확인해야



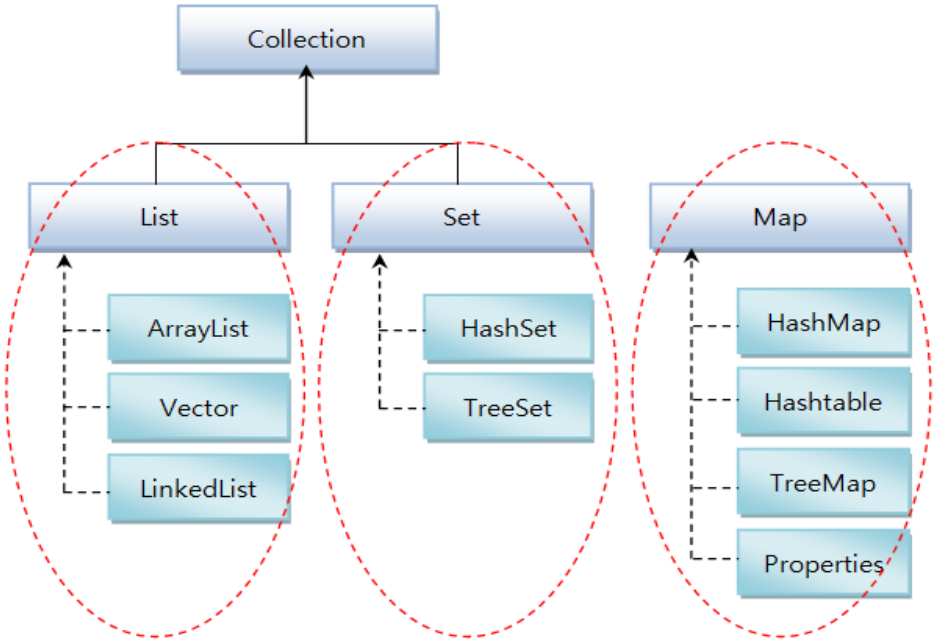
1절. 컬렉션 프레임워크 소개

❖ 컬렉션 프레임워크(Collection Framework)

- 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 제공되는 컬렉션 라이브러리
- java.util 패키지에 포함
- 인터페이스를 통해서 정형화된 방법으로 다양한 컬렉션 클래스 이용

1절. 컬렉션 프레임워크 소개

❖ 컬렉션 프레임워크의 주요 인터페이스



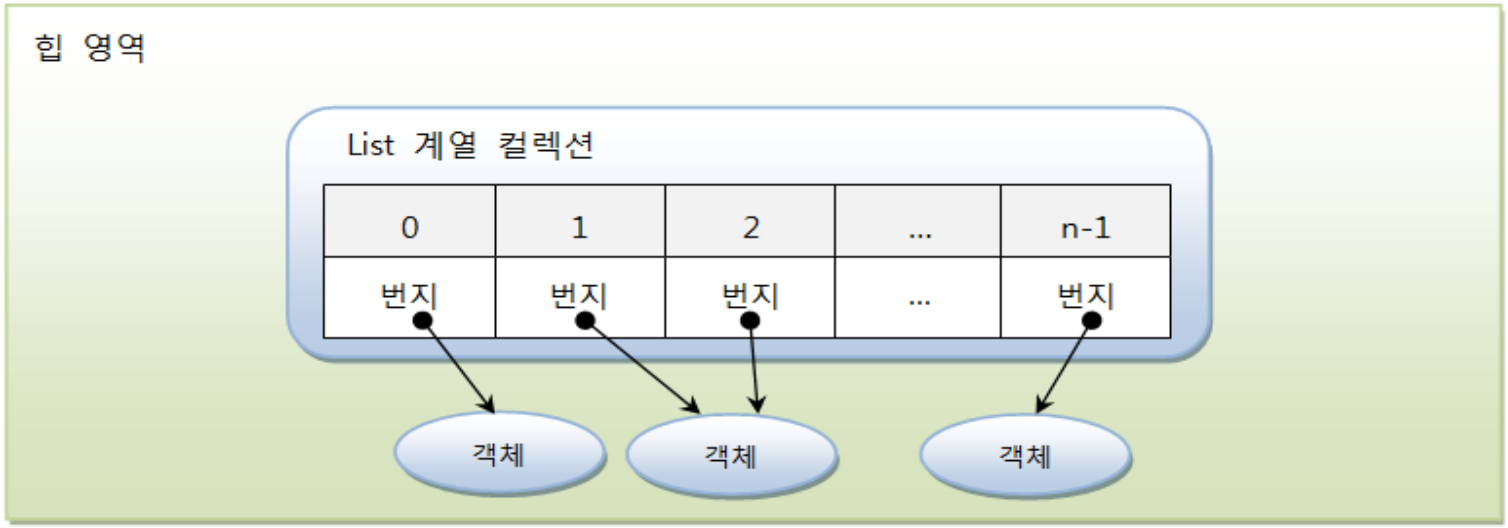
인터페이스 분류		특징	구현 클래스
Collection	List 계열	- 순서를 유지하고 저장 - 중복 저장 가능	ArrayList, Vector, LinkedList
	Set 계열	- 순서를 유지하지 않고 저장 - 중복 저장 안됨	HashSet, TreeSet
Map 계열		- 키와 값의 쌍으로 저장 - 키는 중복 저장 안됨	HashMap, Hashtable, TreeMap, Properties

2절. List 컬렉션

❖ List 컬렉션의 특징 및 주요 메소드

- 특징
  - 인덱스로 관리
  - 중복해서 객체 저장 가능

- 구현 클래스
  - ArrayList
  - Vector
  - LinkedList



2절. List 컬렉션

❖ List 컬렉션의 특징 및 주요 메소드

- 주요 메소드

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨끝에 추가
	void add(int index, E element)	주어진 인덱스에 객체를 추가
	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
	isEmpty()	컬렉션이 비어 있는지 조사
	int size()	저장되어있는 전체 객체수를 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

2절. List 컬렉션

❖ List 컬렉션의 특징 및 주요 메소드

```
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
```

2절. List 컬렉션

❖ List 컬렉션의 특징 및 주요 메소드

```
List<String> list = …;
for(int i=0; i<list.size(); i++) {
    String str = list.get(i);
}
```

저장된 총 객체 수만큼 루핑

i 인덱스에 저장된 String 객체를 가져옴

String 객체를 하나씩 가져옴

```
for(String str : list) {
}
```

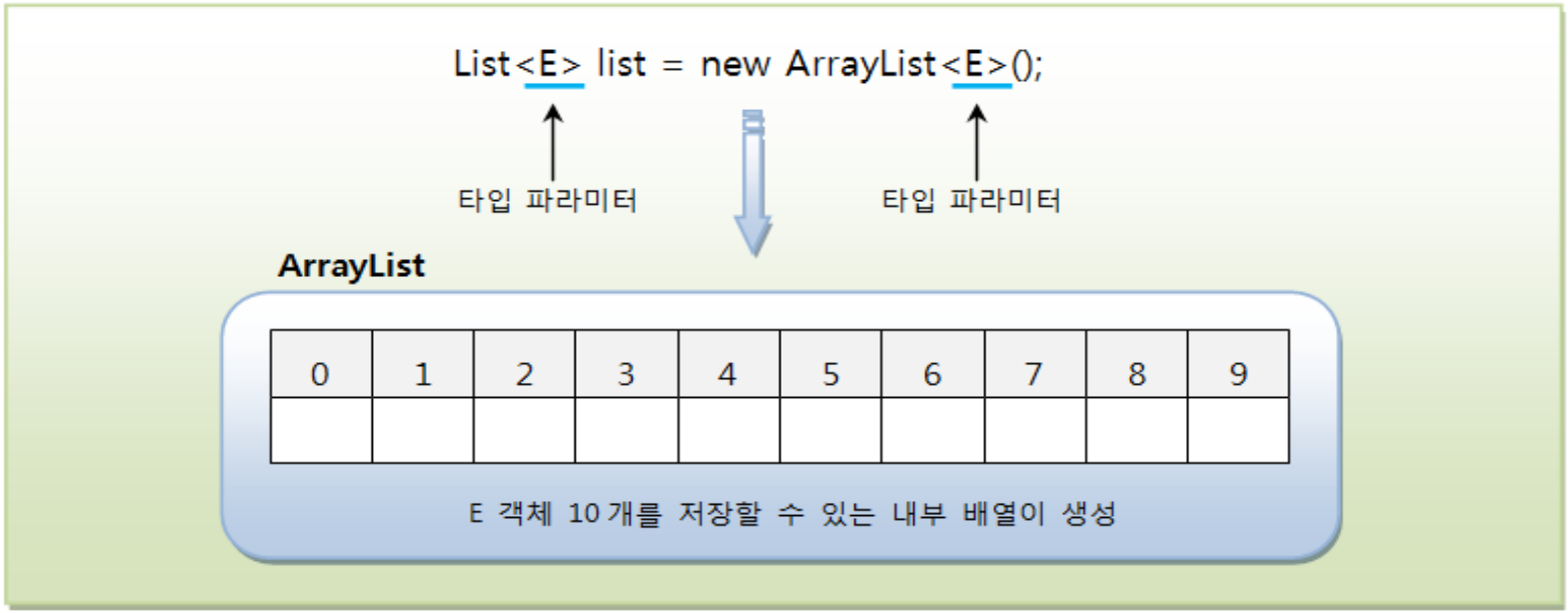
저장된 총 객체 수만큼 루핑

2절. List 컬렉션

❖ ArrayList

- 저장 용량(capacity)
  - 초기 용량 : 10 (따로 지정 가능)
  - 저장 용량을 초과한 객체들이 들어오면 자동적으로 늘어남. 고정도 가능

```
List<String> list = new ArrayList<String>();
```



2절. List 컬렉션

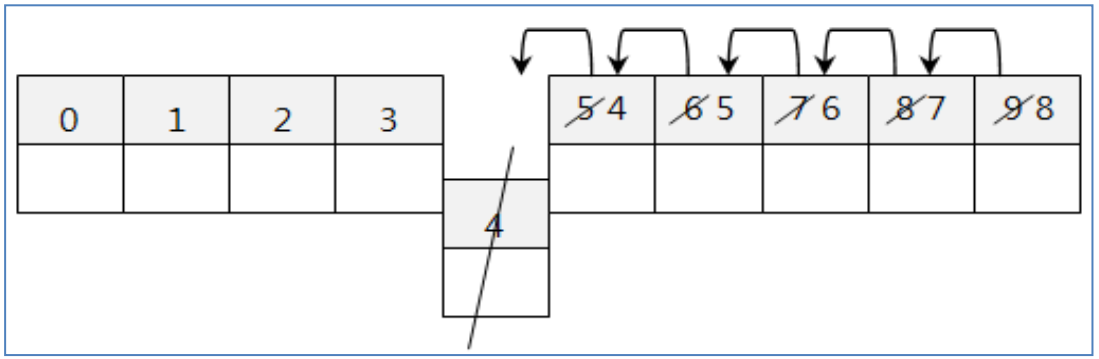
❖ ArrayList

```
list<String> list = new ArrayList<String>();      //컬렉션 생성
list.add("홍길동");                                 //컬렉션에 객체를 추가
String name =  list.get(0);                        //컬렉션에서 객체 검색, 홍길동을 바로 얻음
```

2절. List 컬렉션

❖ ArrayList

- 객체 제거
  - 바로 뒤 인덱스부터 마지막 인덱스까지 모두 앞으로 1씩 당겨짐



2절. List 컬렉션

❖ String 객체를 저장하는 ArrayList: ArrayListExample.java

```
import java.util.*;

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

        list.add("Java");
        list.add("JDBC");
        list.add("Servlet/JSP");
        list.add(2, "Database");
        list.add("iBATIS");

        int size = list.size();
        System.out.println("총 객체수: " + size);
        System.out.println();

        String skill = list.get(2);
        System.out.println("2: " + skill);
        System.out.println();
```

2절. List 컬렉션

❖ String 객체를 저장하는 ArrayList: ArrayListExample.java

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

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

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

2절. List 컬렉션

❖ Arrays.asList() 메서드: ArrayListExample.java

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

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

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

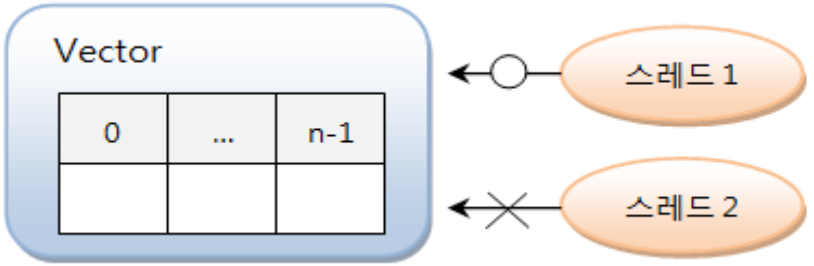
2절. List 컬렉션

❖ Vector

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

- 특징
  - Vector는 스레드 동기화(synchronization)

    복수의 스레드가 동시에 Vector에 접근해 객체를 추가, 삭제하더라도 스레드에 안전(thread safe)



2절. List 컬렉션

❖ **게시물 정보 객체**: ArrayListExample.java

```
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;
    }
}
```

2절. List 컬렉션

❖ Board **객체를 저장하는** Vector: VectorExample.java

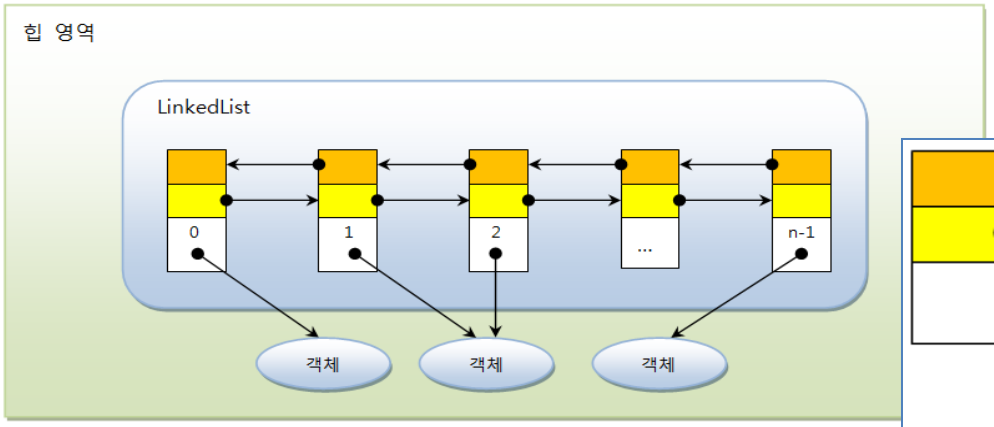
```
import java.util.List;
import java.util.Vector;

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

        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"));

        list.remove(2);
        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);
        }
    }
}
```

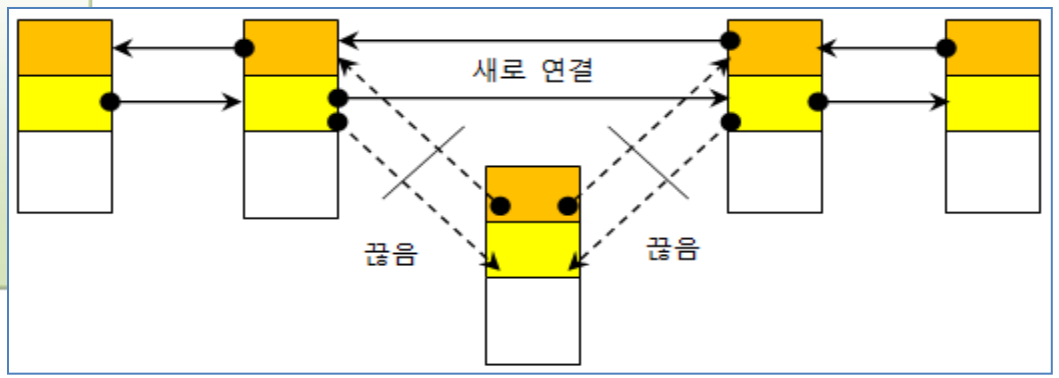
2절. List 컬렉션

❖ LinkedList

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

- 특징
  - 인접 참조를 링크해서 체인처럼 관리
  - 특정 인덱스에서 객체를 제거하거나 추가하게 되면 바로 앞뒤 링크만 변경
  - 빈번한 객체 삭제와 삽입이 일어나는 곳에서는 ArrayList보다 좋은 성능





2절. List 컬렉션

❖ ArrayList와 LinkedList의 실행 성능 비교: ArrayListExample.java

```
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");
```

2절. List 컬렉션

❖ ArrayList와 LinkedList의 실행 성능 비교: ArrayListExample.java

```
        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");
    }
}
```

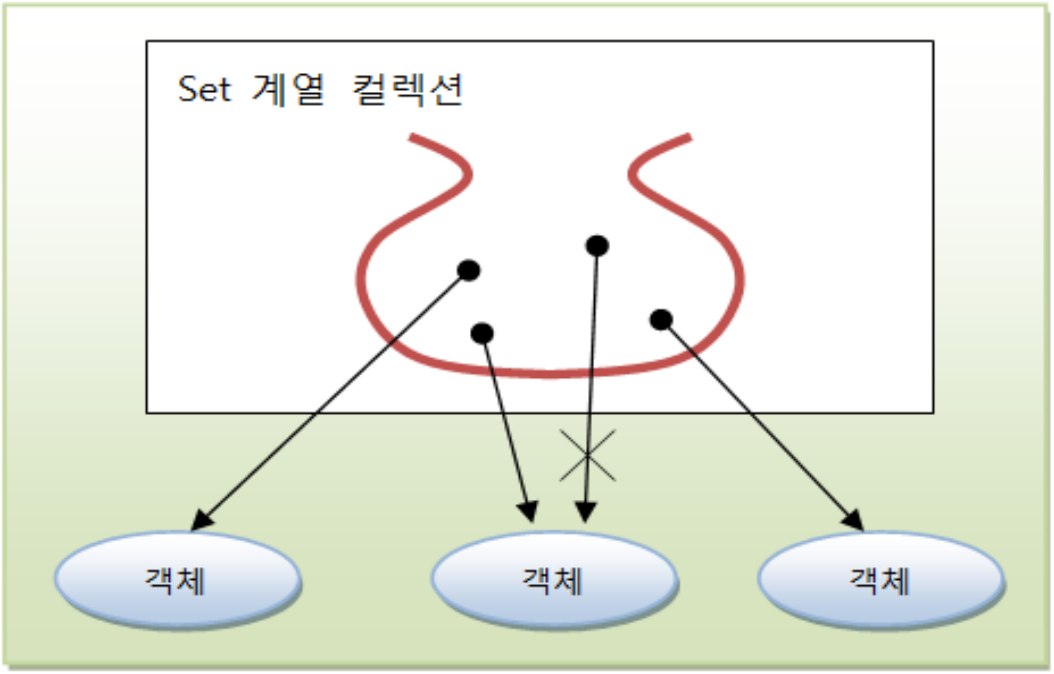
구분	순차적으로 추가/삭제	중간에 추가/삭제	검색
ArrayList	빠르다	느리다	빠르다
LinkedList	느리다	빠르다	느리다

3절. Set 컬렉션

❖ Set 컬렉션의 특징 및 주요 메소드

- 특징
  - 수학의 집합에 비유
  - 저장 순서가 유지되지 않음
  - 객체를 중복 저장 불가
  - 하나의 null만 저장 가능

- 구현 클래스
  - HashSet, LinkedHashSet, TreeSet



3절. Set 컬렉션

❖ Set 컬렉션의 특징 및 주요 메소드

- 주요 메소드

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

3절. Set 컬렉션

❖ Set 컬렉션의 특징 및 주요 메소드

- 전체 객체 대상으로 한 번씩 반복해 가져오는 반복자(Iterator) 제공
  - 인덱스로 객체를 검색해서 가져오는 메소드 없음

```
Set<String> set = …;
Iterator<String> 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( )	Set 컬렉션에서 객체를 제거한다.

3절. Set 컬렉션

❖ Set 컬렉션의 특징 및 주요 메소드

- 전체 객체 대상으로 한 번씩 반복해 가져오는 반복자(Iterator) 제공

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

저장된 객체 수만큼 루핑한다.

- 향상된 for 문으로 대체 가능

```
Set<String> set = …;
for(String str : set) {
}
```

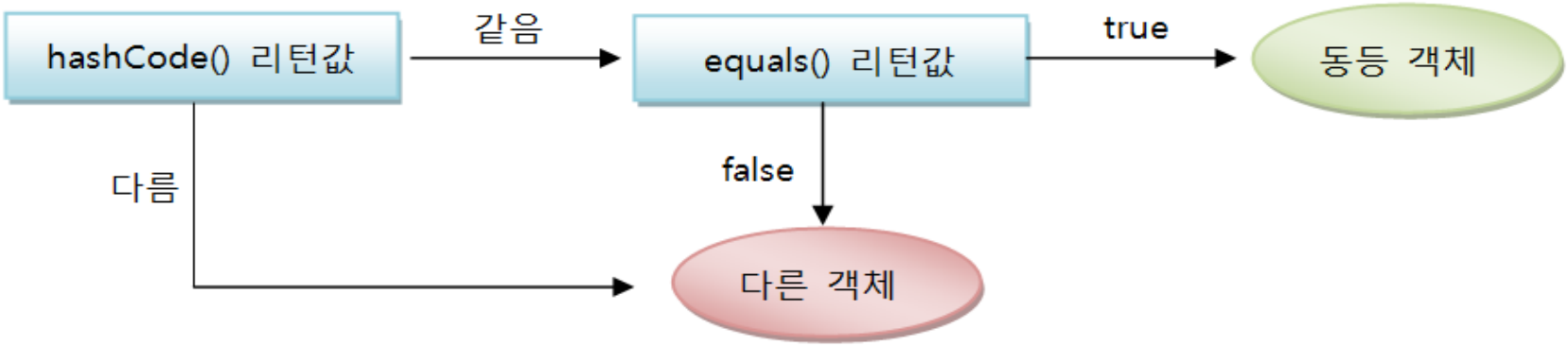
저장된 객체 수만큼 루핑한다.

3절. Set 컬렉션

❖ HashSet (p.736~739)

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

- 특징
  - 동일 객체 및 동등 객체는 중복 저장하지 않음
  - 동등 객체 판단 방법



3절. Set 컬렉션

❖ String 객체를 중복 없이 저장하는 HashSet: HashSetExample1.java

```
import java.util.*;

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

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

        int size = set.size();
        System.out.println("총 객체수: " + size);

        Iterator<String> iterator = set.iterator();
        while(iterator.hasNext()) {
            String element = iterator.next();
            System.out.println("\t" + element);
        }
```

3절. Set 컬렉션

❖ String 객체를 중복 없이 저장하는 HashSet: HashSetExample1.java

```
        set.remove("JDBC");
        set.remove("iBATIS");

        System.out.println("총 객체수: " + set.size());

        for(String element : set) {
            System.out.println("\t" + element);
        }

        set.clear();
        if(set.isEmpty()) { System.out.println("비어 있음"); }
    }
}
```

3절. Set 컬렉션

❖ hashCode()와 equals() 메서드 재정의: Member.java

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

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

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

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

3절. Set 컬렉션

❖ Member **객체를 중복없이 저장하는** HashSet: HashSetExample2.java

```
import java.util.*;

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

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

        System.out.println("총 객체수 : " + set.size());
    }
}
```

4절. Map 컬렉션

❖ Map 컬렉션의 특징 및 주요 메소드

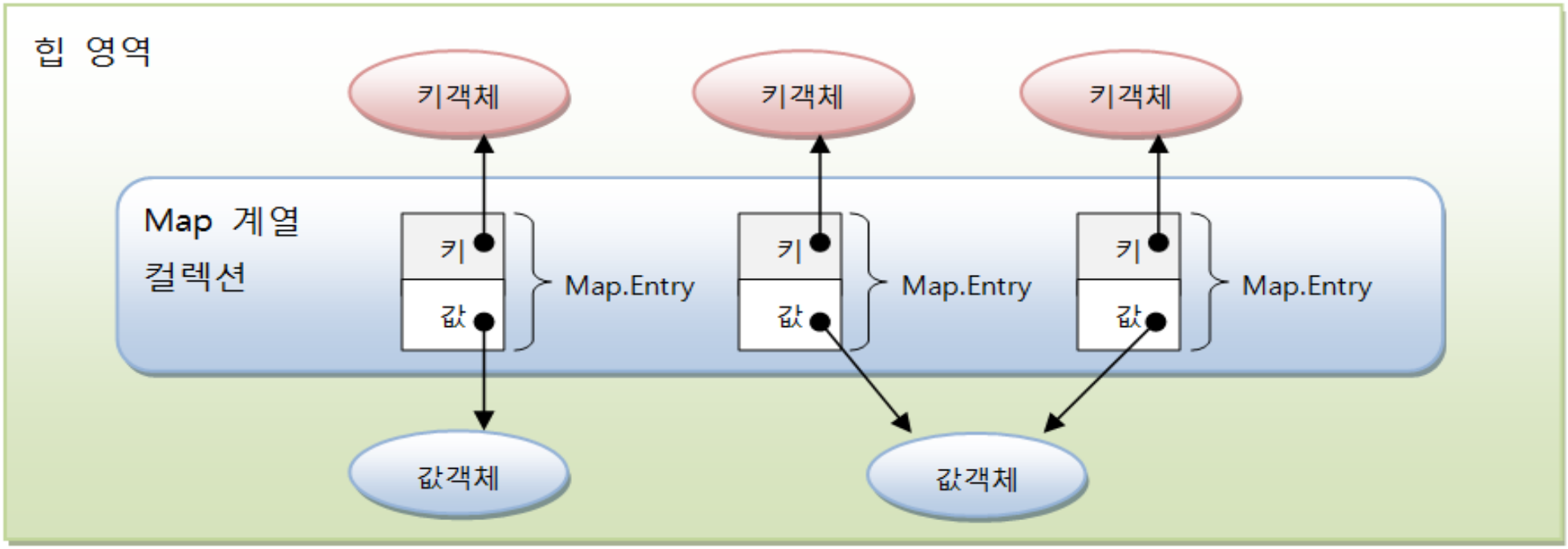
- 특징
  - 키(key)와 값(value)으로 구성된 Map.Entry 객체를 저장하는 구조
  - 키와 값은 모두 객체
  - 키는 중복될 수 없지만 값은 중복 저장 가능

- 구현 클래스
  - HashMap, Hashtable, LinkedHashMap, Properties, TreeMap

4절. Map 컬렉션

❖ Map 컬렉션의 특징 및 주요 메소드

- 키(key)와 값(value)으로 구성된 Entry 객체를 저장하는 구조
- 키, 값 모두 객체



4절. Map 컬렉션

❖ Map 컬렉션의 특징 및 주요 메소드

- 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 Set 에 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 Collection 에 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

4절. Map 컬렉션

❖ HashMap

- 기본 사용방법

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

4절. Map 컬렉션

❖ HashMap

- 순회 방법 – 키 집합으로 순회하기

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

4절. Map 컬렉션

❖ HashMap

- 순회 방법 – Entry 집합으로 순회하기

```
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();
}
```

4절. Map 컬렉션

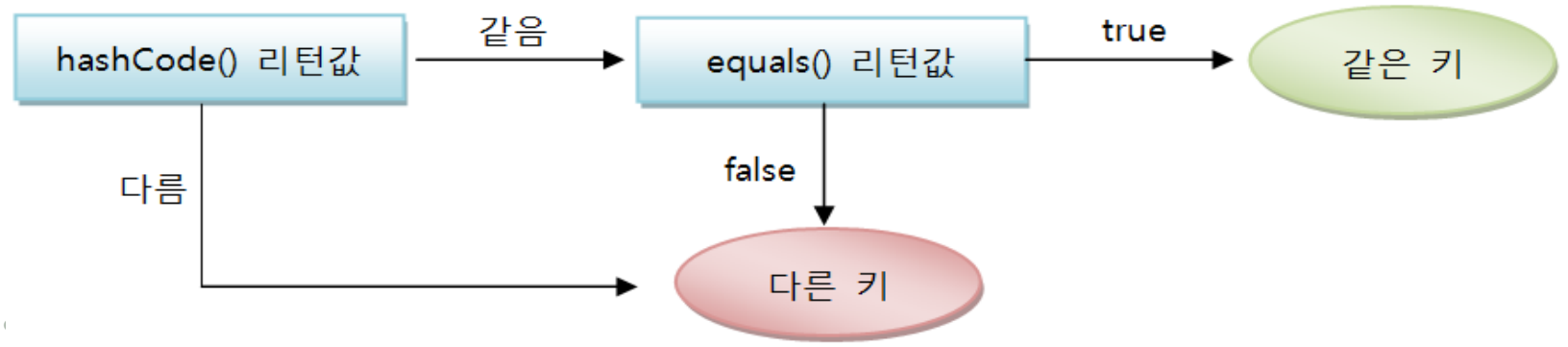
❖ HashMap

- 특징

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

키 타입 — 값 타입 — 키 타입 — 값 타입

- 키 객체는 hashCode()와 equals() 를 재정의해 동등 객체가 될 조건을 정해야



String은 문자열이 같을 경우 동등 객체가 될 수 있도록
hashCode()와 equals() 메소드가 재정의되어 있기 때문

4절. Map 컬렉션

❖ **이름을 키로 접수를 값으로 저장하기**: HashMapExample1.java

```
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);
        map.put("홍길동", 95);
        System.out.println("총 Entry 수: " + map.size());

        //객체 찾기
        System.out.println("\t홍길동 : " + map.get("홍길동"));
        System.out.println();
```

4절. Map 컬렉션

❖ **이름을 키로 접수를 값으로 저장하기**: HashMapExample1.java

```
//객체를 하나씩 처리
Set<String> keySet = map.keySet();
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
map.remove("홍길동");
System.out.println("총 Entry 수: " + map.size());
```

4절. Map 컬렉션

❖ **이름을 키로 접수를 값으로 저장하기**: HashMapExample1.java

```
        //객체를 하나씩 처리
        Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
        Iterator<Map.Entry<String, Integer>> entryIterator = entrySet.iterator();
        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
        map.clear();
        System.out.println("총 Entry 수: " + map.size());
    }
}
```

4절. Map 컬렉션

❖ **키로 사용할 객체** – hashCode()**와** equals() **재정의**: Student.java

```
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) {
        if(obj instanceof Student) {
            Student student = (Student) obj;
            return (sno==student.sno) && (name.equals(student.name)) ;
        } else {
            return false;
        }
    }

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

4절. Map 컬렉션

❖ **학번과 이름이 동일한 경우 같은 키로 인식**: HashMapExample2 .java

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

public class HashMapExample2 {
    public static void main(String[] args) {
        Map<Student, Integer> map = new HashMap<Student, Integer>();

        map.put(new Student(1, "홍길동"), 95);
        map.put(new Student(1, "홍길동"), 95);

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

4절. Map 컬렉션

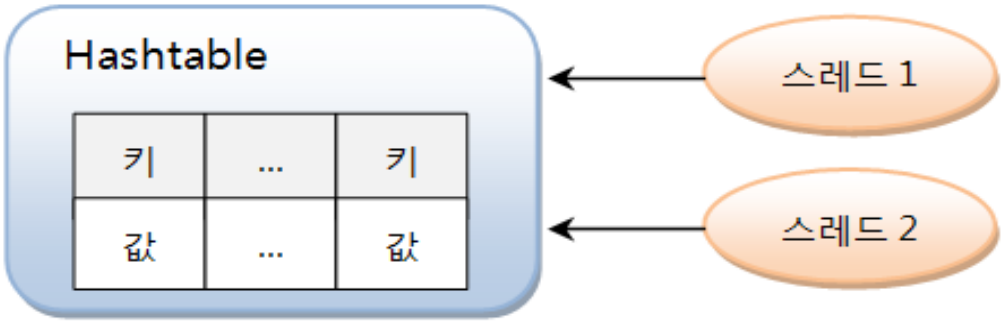
❖ Hashtable

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

키 타입 값 타입 키 타입 값 타입

- 특징
  - 키 객체 만드는 법은 HashMap과 동일
  - Hashtable은 스레드 동기화(synchronization)가 된 상태

    복수의 스레드가 동시에 Hashtable에 접근해서 객체를 추가, 삭제하더라도 스레드에 안전(thread safe)



스레드 동기화 적용됨

4절. Map 컬렉션

❖ **아이디와 비밀번호 검사하기**: HashtableExample .java

```
import java.util.*;

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

        map.put("spring", "12");
        map.put("summer", "123");
        map.put("fall", "1234");
        map.put("winter", "12345");

        Scanner scanner = new Scanner(System.in);
```

4절. Map 컬렉션

❖ **아이디와 비밀번호 검사하기**: HashtableExample .java

```
        while(true) {
            System.out.println("아이디와 비밀번호를 입력해주세요");
            System.out.print("아이디: ");
            String id = scanner.nextLine();

            System.out.print("비밀번호: ");
            String password = scanner.nextLine();
            System.out.println();

            if(map.containsKey(id)) {
                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");
            }
        }
    }
}
```

4절. Map 컬렉션

❖ Properties

- 특징
  - 키와 값을 String 타입으로 제한한 Map 컬렉션
  - Properties는 프로퍼티(~.properties) 파일을 읽어 들일 때 주로 사용

- 프로퍼티(~.properties) 파일
  - 옵션 정보, 데이터베이스 연결 정보, 국제화(다국어) 정보를 기록
    텍스트 파일로 활용
  - 애플리케이션에서 주로 변경이 잦은 문자열을 저장
    유지 보수를 편리하게 만들어 줌

  - 키와 값이 = 기호로 연결되어 있는 텍스트 파일
    ISO 8859-1 문자셋으로 저장
    한글은 유니코드(Unicode)로 변환되어 저장

4절. Map 컬렉션

❖ **키=값으로 구성된 프로퍼티**: database.properties

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

4절. Map 컬렉션

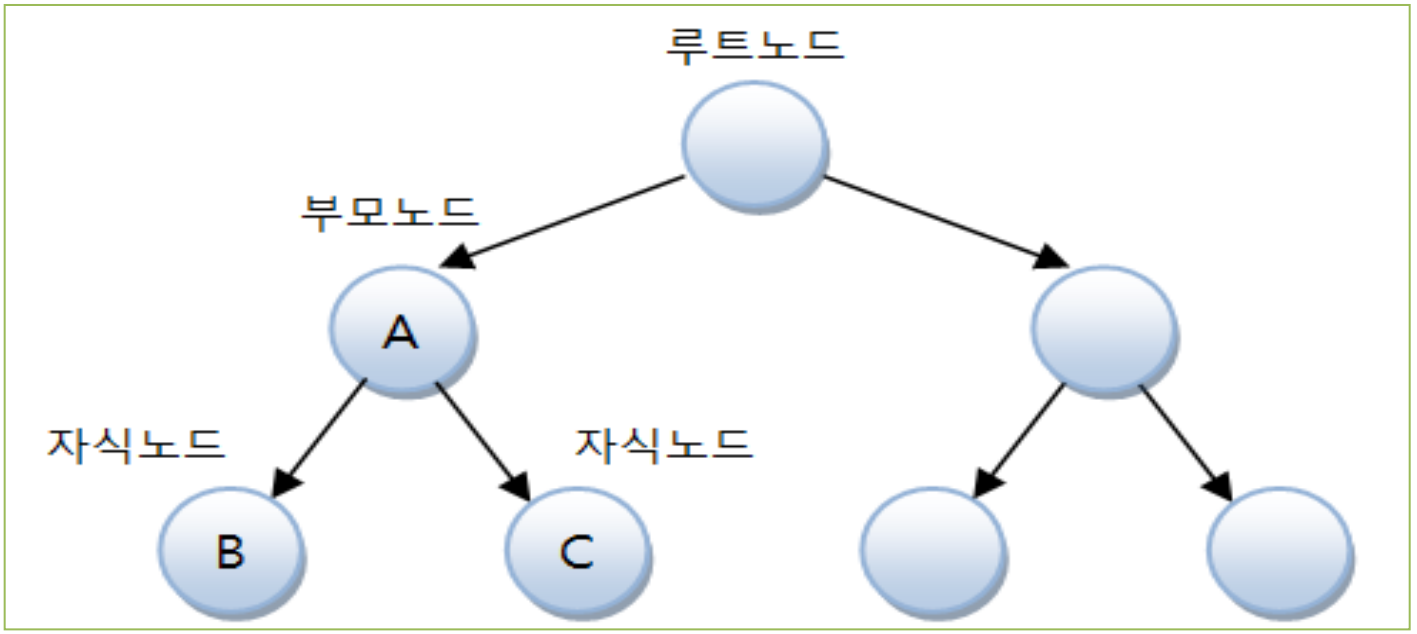
❖ **키=값으로 구성된 프로퍼티**: PropertiesExample.java

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

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

5절. 검색 기능을 강화시킨 컬렉션

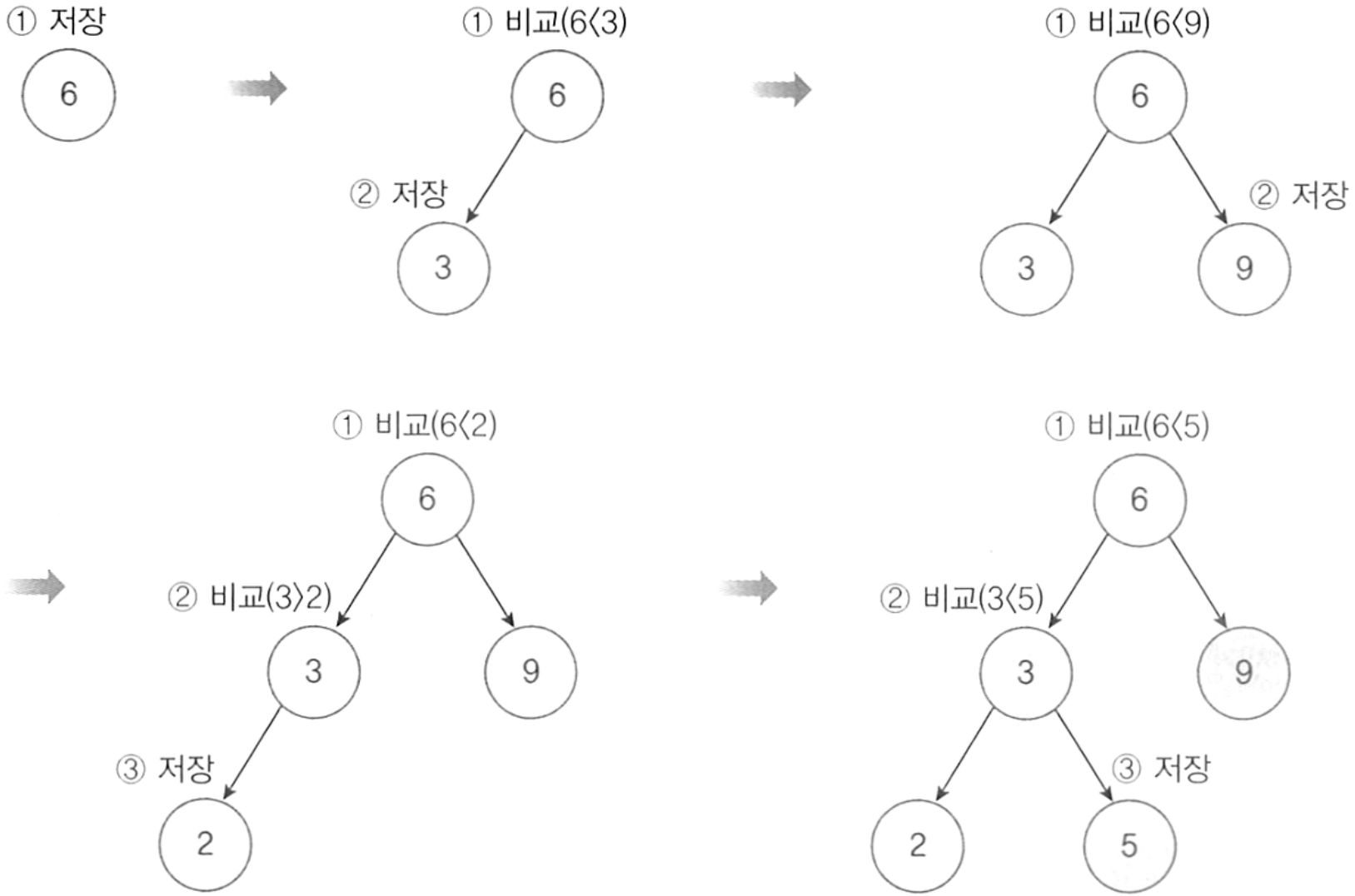
❖ 검색 기능을 강화시킨 컬렉션 (계층 구조 활용)

- TreeSet, TreeMap → 이진트리(binary tree) 사용하기 때문에 검색 속도 향상



5절. 검색 기능을 강화시킨 컬렉션

❖ 검색 기능을 강화시킨 컬렉션 (계층 구조 활용)

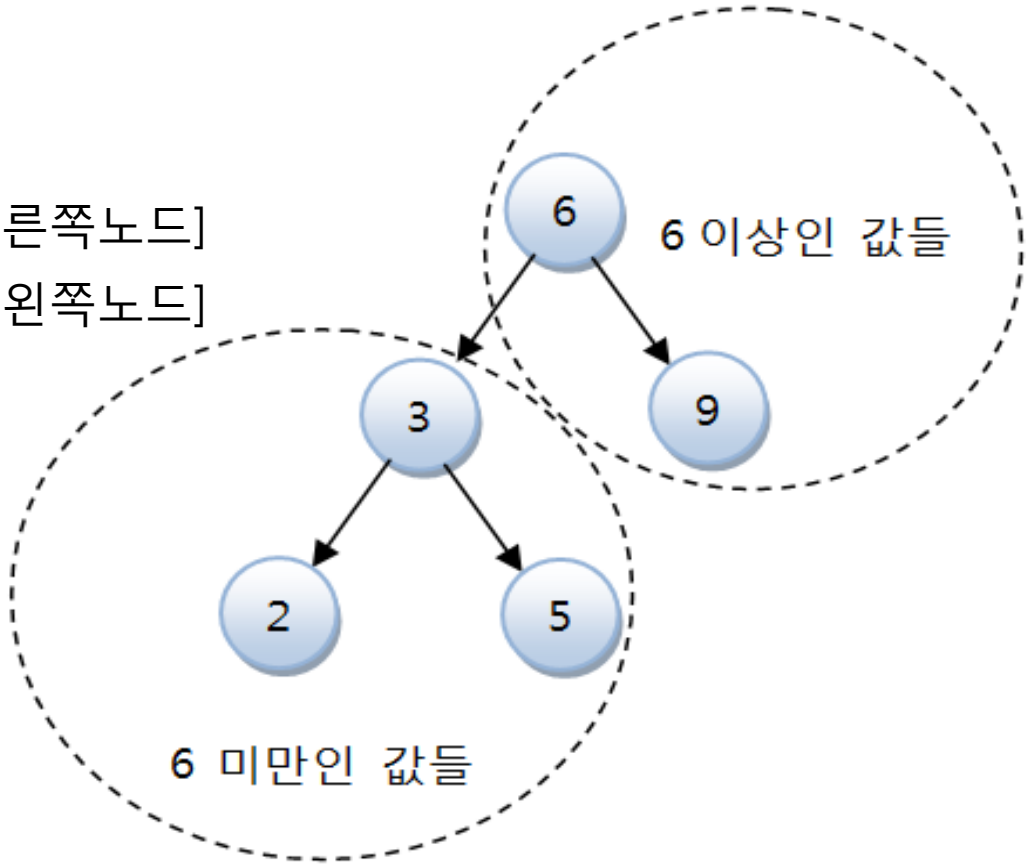


5절. 검색 기능을 강화시킨 컬렉션

❖ 이진 트리 구조

- 부모 노드와 자식 노드로 구성
  - 왼쪽 자식 노드: 부모 보다 적은 값
  - 오른쪽 자식 노드: 부모 보다 큰 값

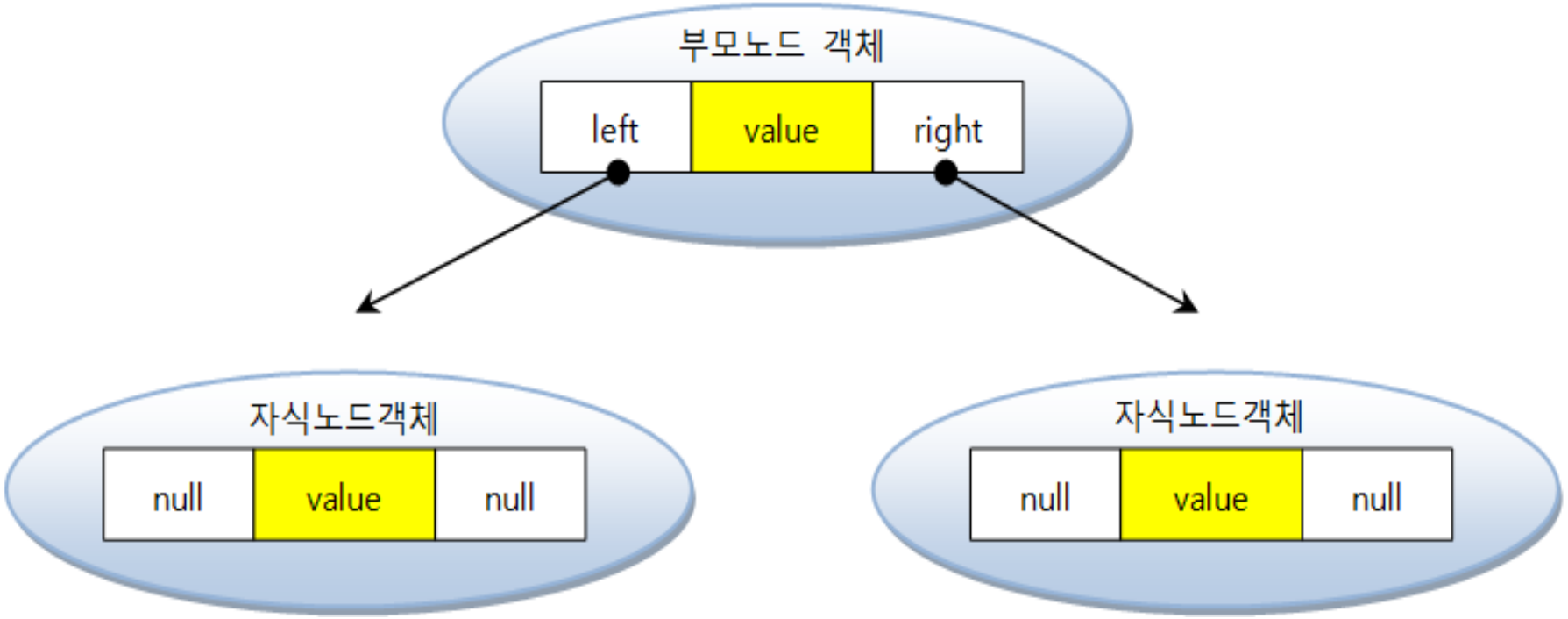
- 정렬 쉬움
  - 올림 차순: [왼쪽노드→부모노드→오른쪽노드]
  - 내림 차순: [오른쪽노드→부모노드→왼쪽노드]



5절. 검색 기능을 강화시킨 컬렉션

❖ TreeSet

- 특징
  - 이진 트리(binary tree)를 기반으로 한 Set 컬렉션
  - 왼쪽과 오른쪽 자식 노드를 참조하기 위한 두 개의 변수로 구성



5절. 검색 기능을 강화시킨 컬렉션

❖ TreeSet

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

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeSet

- 검색관련 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

5절. 검색 기능을 강화시킨 컬렉션

❖ **특정 객체 찾기**: TreeSetExample1.java

```
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);
```

5절. 검색 기능을 강화시킨 컬렉션

❖ **특정 객체 찾기**: TreeSetExample1.java

```
        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() + ")");
        }
    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeSet

- 정렬 관련 메소드

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

```
NavigableSet<E> descendingSet = treeSet.descendingSet();
NavigableSet<E> ascendingSet = descendingSet.descendingSet();
```

5절. 검색 기능을 강화시킨 컬렉션

❖ **객체정렬하기**: TreeSetExample2.java

```
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> descendingSet = scores.descendingSet();
        for(Integer score : descendingSet) {
            System.out.print(score + " ");
        }
        System.out.println();

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

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

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeSet

- 범위 검색 관련 메소드

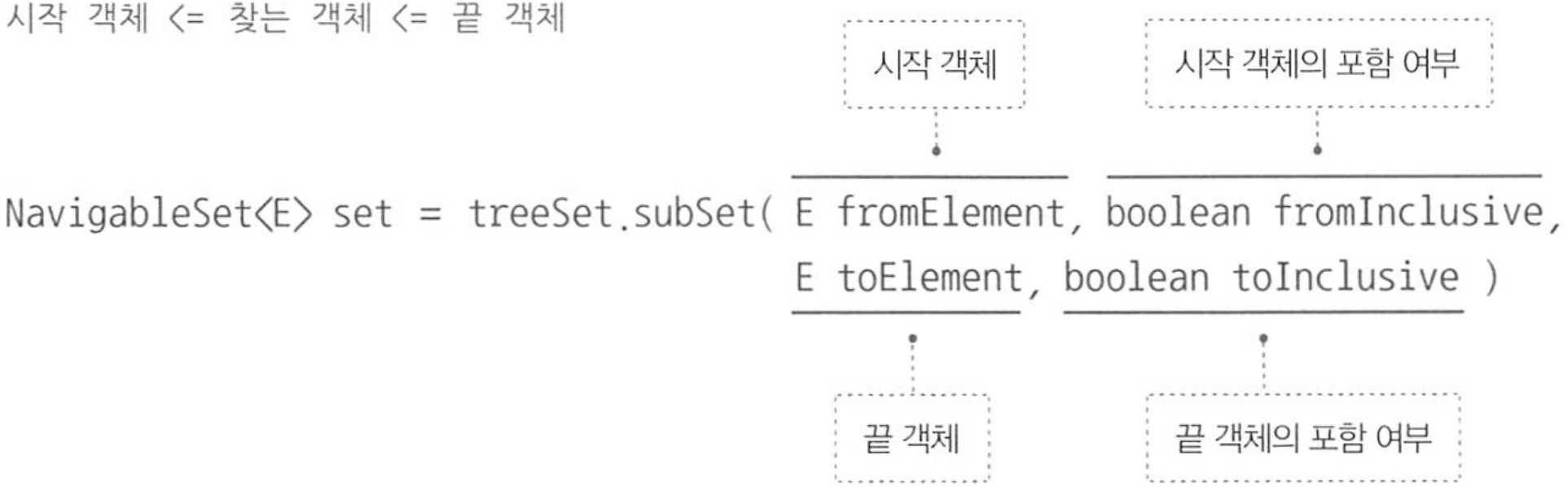
리턴 타입	메소드	설명
NavigableSet<E>	headSet(E toElement, boolean inclusive)	주어진 객체보다 낮은 객체들을 NavigableSet으로 리턴. 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableSet<E>	tailSet(E fromElement, boolean inclusive)	주어진 객체보다 높은 객체들을 NavigableSet으로 리턴. 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableSet<E>	subSet(E fromElement, boolean 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

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeSet

- 범위 검색 관련 메소드

```
시작 객체 < 찾는 객체 < 끝 객체
시작 객체 <= 찾는 객체 <= 끝 객체
```



5절. 검색 기능을 강화시킨 컬렉션

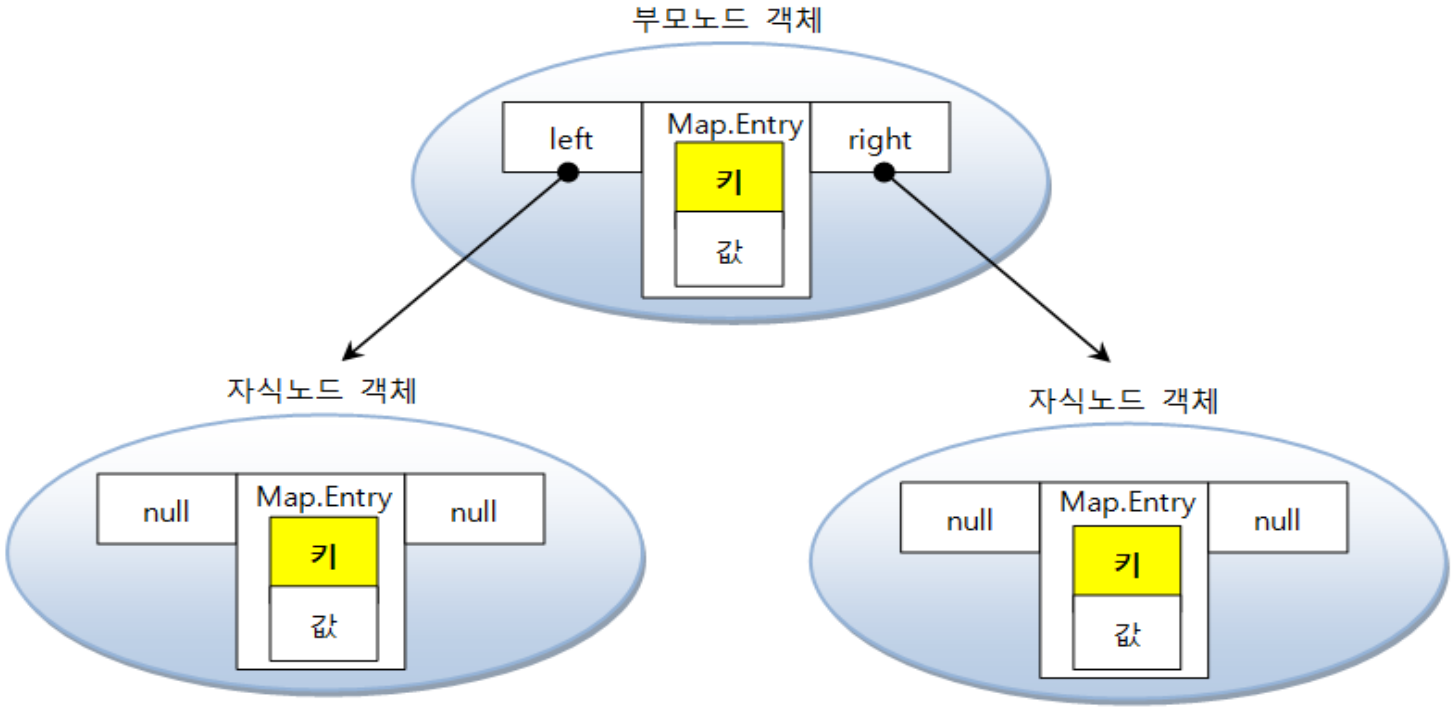
❖ **영어 단어를 정렬하고, 범위 검색해보기**: TreeSetExample3.java

```
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]");
        NavigableSet<String> rangeSet = treeSet.subSet("c", true, "f", true);
        for(String word : rangeSet) {
            System.out.println(word);
        }

    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeMap

- 특징
  - 이진 트리(binary tree) 를 기반으로 한 Map 컬렉션
  - 키와 값이 저장된 Map.Entry를 저장
  - 왼쪽과 오른쪽 자식 노드를 참조하기 위한 두 개의 변수로 구성



5절. 검색 기능을 강화시킨 컬렉션

❖ TreeMap

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


```
TreeMap<String, Integer> treeMap = new TreeMap<String, Integer>();
```

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeMap

- 검색 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

5절. 검색 기능을 강화시킨 컬렉션

❖ **특정** Map.Entry **찾기**: TreeMapExample1.java

```
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");
```

5절. 검색 기능을 강화시킨 컬렉션

❖ **특정** Map.Entry **찾기**: TreeMapExample1.java

```
        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() + ")");
        }
    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeMap

- 정렬 메서드

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

```
NavigableMap<K,V> descendingMap = treeMap.descendingMap();
NavigableMap<K,V> ascendingMap = descendingMap.descendingMap();
```

5절. 검색 기능을 강화시킨 컬렉션

❖ **객체 정렬하기**: TreeMapExample2.java

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

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();
```

5절. 검색 기능을 강화시킨 컬렉션

❖ **객체 정렬하기**: TreeMapExample2.java

```
        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() + " ");
        }
    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ TreeMap

- 범위 검색 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

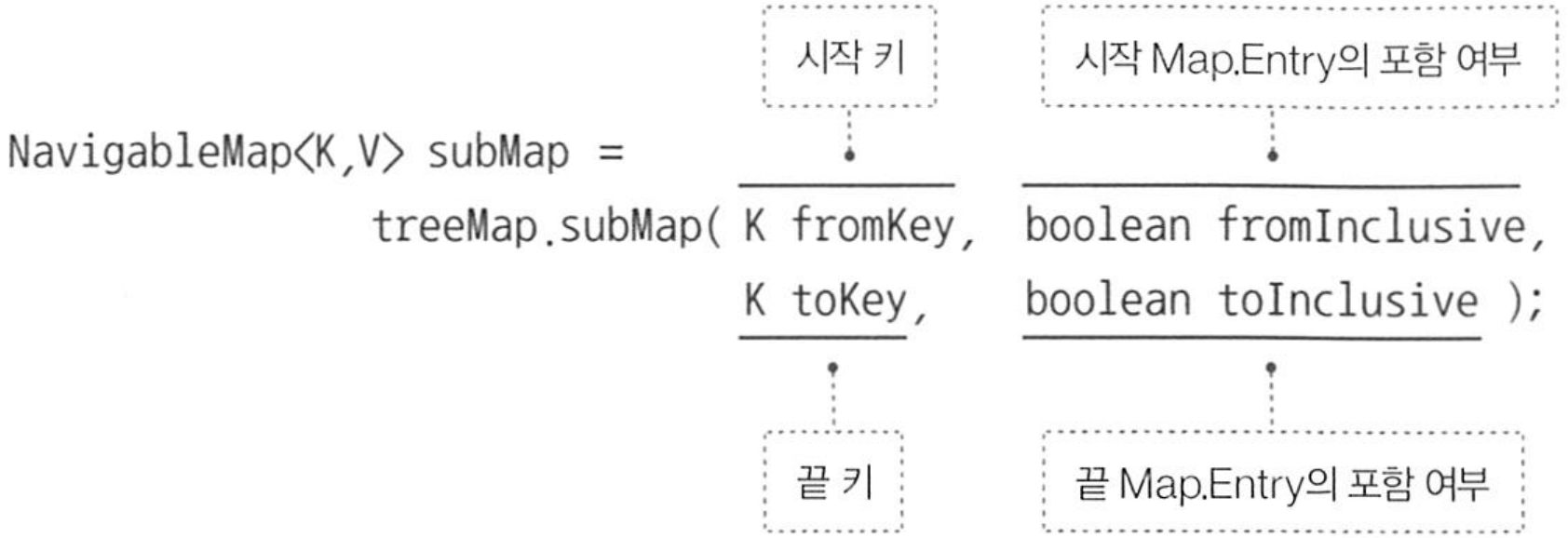
5절. 검색 기능을 강화시킨 컬렉션

❖ TreeMap

- 범위 검색 메서드

  시작 Map.Entry < 찾는 Map.Entry < 끝 Map.Entry

  시작 Map.Entry <= 찾는 Map.Entry <= 끝 Map.Entry



5절. 검색 기능을 강화시킨 컬렉션

❖ **키로 정렬하고, 범위 검색해보기**: TreeMapExample3.java

```
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");
        }

    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ Comparable과 Comparator

- TreeSet과 TreeMap의 자동 정렬
  - TreeSet의 객체와 TreeMap의 키는 저장과 동시에 자동 오름차순 정렬
  - 숫자(Integer, Double)타입일 경우에는 값으로 정렬
  - 문자열(String) 타입일 경우에는 유니코드로 정렬

  - TreeSet과 TreeMap은 정렬 위해 java.lang.Comparable을 구현 객체를 요구

    Integer, Double, String은 모두 Comparable 인터페이스 구현

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

Comparable을 구현하고 있지 않을 경우에는 저장하는 순간 ClassCastException 발생

5절. 검색 기능을 강화시킨 컬렉션

❖ Compareable 구현 클래스: Person.java

```
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;
    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ **사용자 정의 객체를 나이순으로 정렬하기**: ComparableExample.java

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

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

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

        Iterator<Person> iterator = treeSet.iterator();
        while(iterator.hasNext()) {
            Person person = iterator.next();
            System.out.println(person.name + ":" + person.age);
        }
    }
}
```

5절. 검색 기능을 강화시킨 컬렉션

❖ Comparable과 Comparator

- 생성자의 매개값으로 정렬자(Comaparator)를 제공하면 Comparable 비구현 객체도 정렬

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

오름차순 또는 내림차순 정렬자

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

- Comparator 인터페이스

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

5절. 검색 기능을 강화시킨 컬렉션

❖ Comparable을 **구현하지 않은** : ComparableExample.java

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

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

5절. 검색 기능을 강화시킨 컬렉션

❖ Fruit의 내림 차순 정렬자: DescendingComparator.java

```
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;
    }
}


/*public class DescendingComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Fruit fruit1 = (Fruit) o1;
        Fruit fruit2 = (Fruit) o2;
        if(fruit1.price < fruit2.price) return 1;
        else if(fruit1.price == fruit2.price) return 0;
        else return -1;
    }
}*/
```

5절. 검색 기능을 강화시킨 컬렉션

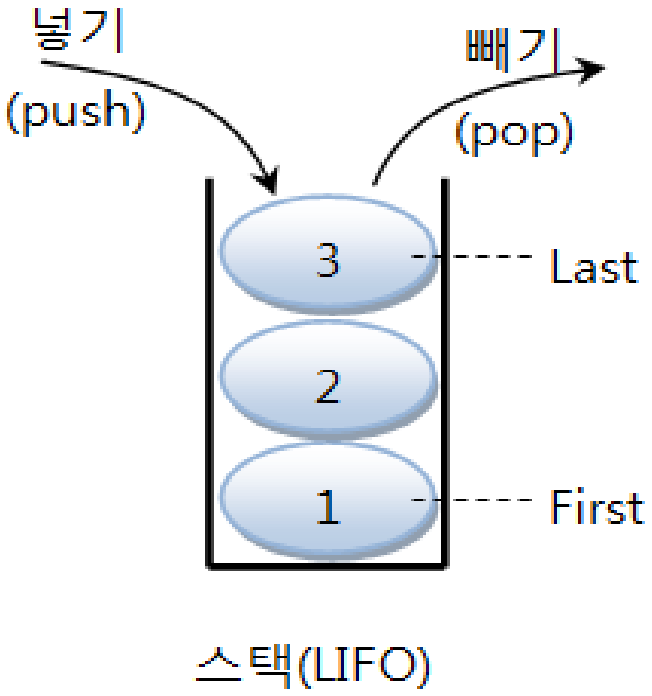
❖ **내림차순 정렬자를 사용하는** TestSet: ComparatorExample.java

```
import java.util.Iterator;
import java.util.TreeSet;
public class ComparatorExample {
    public static void main(String[] args) {
        /*
        TreeSet<Fruit> treeSet = new TreeSet<Fruit>();
        //Fruit이 Comparable을 구현하지 않았기 때문에 예외 발생
        treeSet.add(new Fruit("포도", 3000));
        treeSet.add(new Fruit("수박", 10000));
        treeSet.add(new Fruit("딸기", 6000));
        */
        TreeSet<Fruit> treeSet = new TreeSet<Fruit>(new DescendingComparator());
        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);
        }
    }
}
```

6절. LIFO와 FIFO 컬렉션

❖ Stack 클래스

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



스택(LIFO)

- 특징
  - 후입선출(LIFO: Last In First Out) 구조
  - 응용 예: JVM 스택 메모리

- 주요 메소드

리턴타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣는다.
E	peek()	스택의 맨위 객체를 가져온다. 객체를 스택에서 제거하지는 않는다.
E	pop()	스택의 맨위 객체를 가져온다. 객체를 스택에서 제거한다.

5절. 검색 기능을 강화시킨 컬렉션

❖ **동전 클래스**: Coin.java

```
public class Coin {
    private int value;

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

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

5절. 검색 기능을 강화시킨 컬렉션

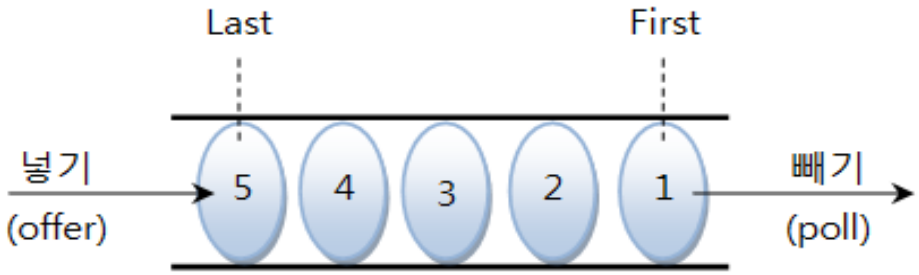
❖ Stack을 **이용한 동전 케이스**: StackExample .java

```
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");
        }

    }
}
```

6절. LIFO와 FIFO 컬렉션

❖ Queue 인터페이스

```
Queue queue = new LinkedList();
```



큐(FIFO)

- 특징
  - 선입선출(FIFO: First In First Out)
  - 응용 예: 작업 큐, 메시지 큐, …
  - 구현 클래스: LinkedList
- 주요 메소드

리턴타입	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.

5절. 검색 기능을 강화시킨 컬렉션

❖ Message 클래스: Message.java

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

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

5절. 검색 기능을 강화시킨 컬렉션

❖ Queue를 이용한 메시지 큐: QueueExample.java

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

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

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

5절. 검색 기능을 강화시킨 컬렉션

❖ Queue를 **이용한 메시지 큐**: QueueExample.java

```
        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;
            }
        }
    }
}
```

7절. 동기화된(synchronized) 컬렉션

❖ 비 동기화된 컬렉션을 동기화된 컬렉션으로 래핑

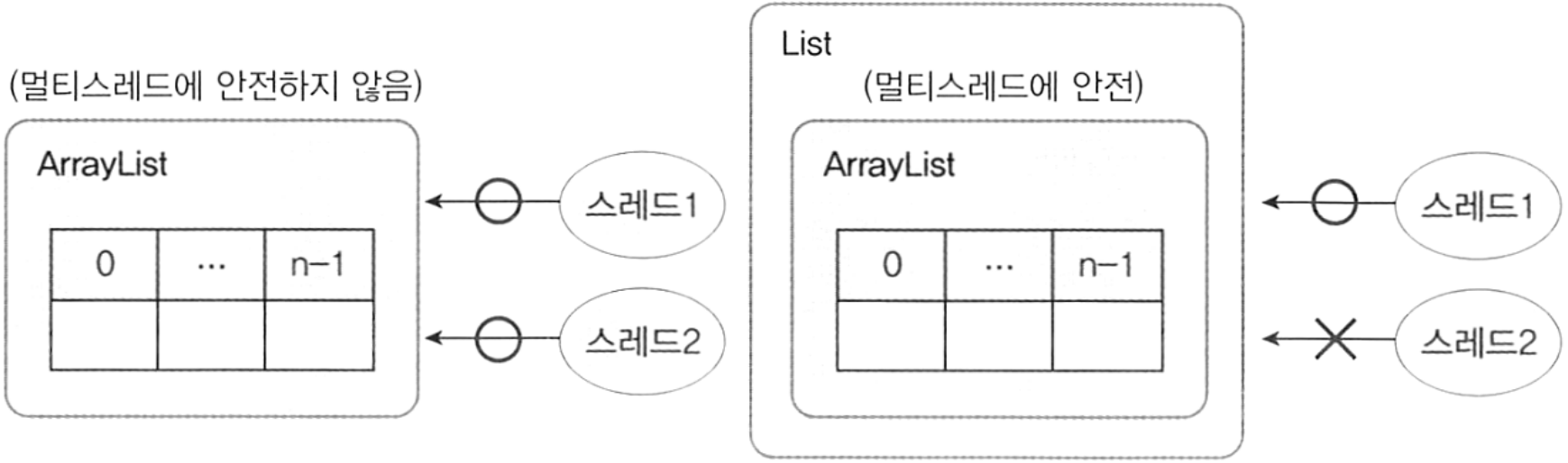
- Collections의 synchoronizedXXX() 메소드 제공

리턴 타입	메소드(매개 변수)	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

7절. 동기화된(synchronized) 컬렉션

❖ 비 동기화된 컬렉션을 동기화된 컬렉션으로 래핑

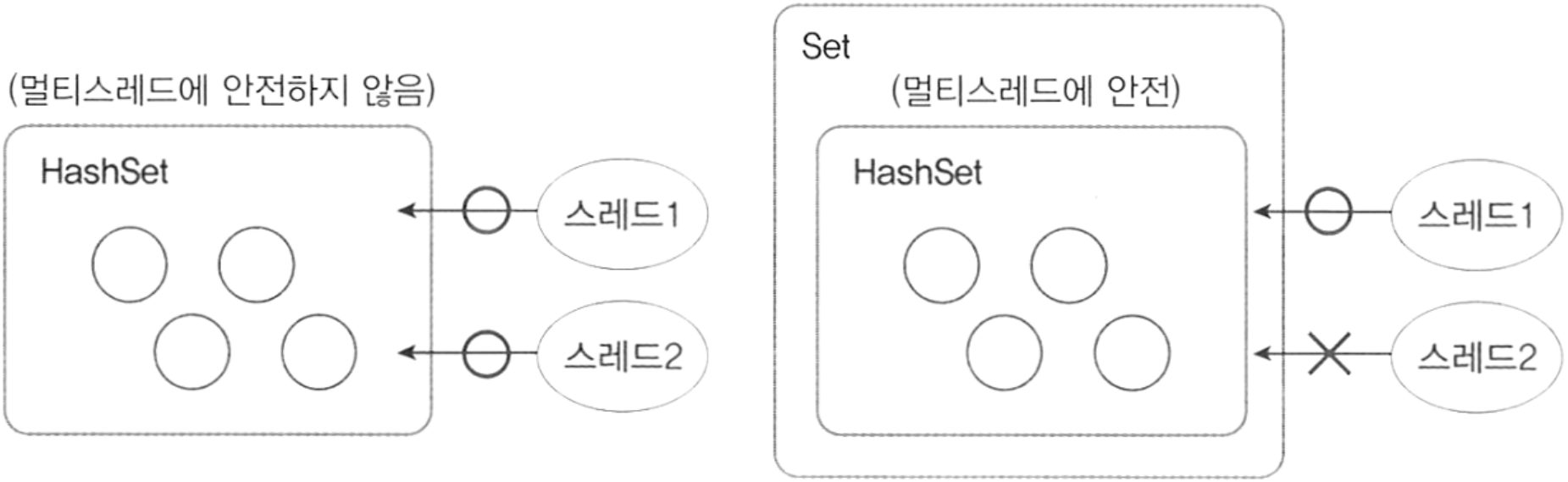
```
List<T> list = Collections.synchronizedList(new ArrayList<T>());
```



7절. 동기화된(synchronized) 컬렉션

❖ 비 동기화된 컬렉션을 동기화된 컬렉션으로 래핑

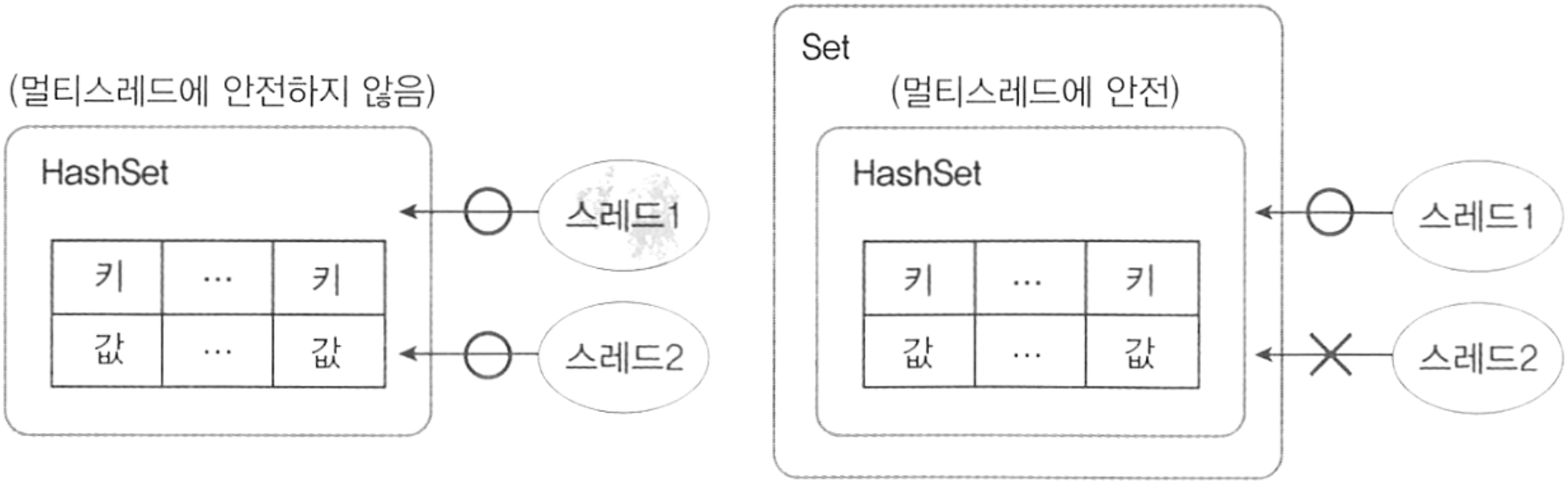
```
Set<E> set = Collections.synchronizedSet(new HashSet<E>());
```



7절. 동기화된(synchronized) 컬렉션

❖ 비 동기화된 컬렉션을 동기화된 컬렉션으로 래핑

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



8절. 병렬 처리를 위한 컬렉션

❖ 동기화(Synchronized) 컬렉션의 단점

- 하나의 스레드가 요소 처리할 때 전체 잠금 발생
  - 다른 스레드는 대기 상태
  - 멀티 스레드가 병렬적으로 컬렉션의 요소들을 빠르게 처리할 수 없음

❖ 컬렉션 요소를 병렬처리하기 위해 제공되는 컬렉션

- ConcurrentHashMap
  - 부분(segment) 잠금 사용

    처리하는 요소가 포함된 부분만 잠금
    나머지 부분은 다른 스레드가 변경 가능하게 → 부분 잠금

- ConcurrentLinkedQueue
  - 락-프리(lock-free) 알고리즘을 구현한 컬렉션

    잠금 사용하지 않음
    여러 개의 스레드가 동시에 접근하더라도 최소한 하나의 스레드가 성공하도록(안전하게 요소를 저장하거나 얻도록) 처리