Chapter 15. 컬렉션 자료구조

15.1 컬렉션 프레임워크

> 자바는 널리 알려져 있는 자료구조를 바탕으로 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 관련된 인터페이스와 클래스들을 java.util 패키지에 포함시켜 놓았다. 이들을 총칭해서 **컬렉션 프레임워크 (Collection Framework)** 라고 부른다.

인터페이스 분류	특징	구현 클래스
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

> List와 Set은 객체를 추가, 삭제, 검색하는 방법에 있어서 공통점이 있기 때문에 공통된 메소드만 따로 모아 **Collection 인터페이스**로 정의해 두고 이것을 상속하고 있다. Map은 키와 값을 하나의 쌍으로 묶어서 관리하는 구조로 되어 있어 List 및 Set과는 사용 방법이 다르다.

15.2 List 컬렉션

> List 컬렉션은 객체를 **인덱스로 관리**하기 때문에 객체를 저장하면 인덱스가 부여되고 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공한다. ArrayList, Vector, LinkedList 등이 있다.

ArrayList

- List 컬렉션에서 가장 많이 사용하는 컬렉션.
- **내부 배열**에 객체가 저장됨. 일반 배열과의 차이점은 **제한 없이** 객체를 추가할 수 있다는 점.
- 객체 자체가 아닌 객체의 번지를 저장.
- 중복 저장할 수 있는데, 이때는 동일한 번지가 저장됨. `null`도 저장 가능.
- 특정 인덱스의 객체를 제거하면 바로 뒤 인덱스부터 마지막 인덱스까지 모두 앞으로 1씩 당겨짐. 마찬가지로 특정 인덱스에 객체를 삽입하면 해당 인덱스부터 마지막 인덱스까지 모두 1씩 밀려남. 따라서 **빈번한 객체 삭제와 삽입이 일어나는 곳**에서는 ArrayList를 사용하는 것은 바람직하지 않고 대신 **LinkedList** 사용하는 것이 좋음.

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();

        //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);

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

Vector

- ArrayList와 동일한 내부 구조.
- 차이점은 동기화된(synchronized) 메소드로 구성되어 있기 때문에 멀티 스레드가 동시에 Vector() 메소드를 실행할 수 없음. **멀티 스레드 환경**에서 안전하게 객체 추가, 삭제 가능.

VectorExample.java

```
package ch15.sec02.exam02;

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

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

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

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

```
                }
            }
        };

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

        //작업 스레드들이 모두 종료될때까지 메인 스레드를 기다리게함
        try {
            threadA.join();
            threadB.join();
        } catch(Exception e) {
        }

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

결과

```
총 객체 수: 2000
```

> 이를 ArrayList로 실행한다면 실행 결과는 2000이 나오지 않거나 에러가 발생할 수 있다. 두 스레드가 동시에 add() 메소드를 호출할 수 있어 경합이 발생하거나 결국 하나만 저장되기 때문이다. 반면 Vector의 add()는 동기화 메소드이므로 한 번에 하나의 스레드만 실행할 수 있어 경합이 발생하지 않는다.

LinkedList

- ArrayList와 동일한 사용 방법, 완전히 다른 내부 구조.
- ArrayList는 내부 배열에 객체 저장, LinkedList는 인접 객체를 체인처럼 연결해서 관리함.
- LinkedList는 특정 위치에 객체를 삽입하거나 삭제하면 **바로 앞뒤 링크만 변경**하면 되므로 **빈번한 객체 삭제와 삽입**이 일어나는 곳에서는 ArrayList보다 좋은 성능 발휘.

15.3 Set 컬렉션

> List 컬렉션은 저장 순서를 유지하지만, Set 컬렉션은 **저장 순서가 유지되지 않는다.** 또한 객체를 **중복해서 저장할 수 없고**, 하나의 null만 저장할 수 있다. 수학의 집합에도 비유된다. HastSet, LinkedHashSet, TreeSet 등이 있다. 인덱스로 관리하지 않기 때문에 인덱스를 매개값으로 갖는 메소드가 없다.

HashSet

- Set 컬렉션 중 가장 많이 사용.

- 중복 저장하지 않으므로 다른 객체라도 `hashCode()` 메소드의 리턴값이 같고, `equals()` 메소드가 true 를 리턴하면 동일한 객체라고 판단, 저장하지 않음.
- 같은 문자열을 갖는 `String` 객체는 동등한 객체로 간주.

Member.java

```
package ch15.sec03.exam02;

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

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

    //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());
    }
}
```

결과

```
총 객체 수 : 1
```

> Set 컬렉션으로 객체를 한 개씩 반복해 가져오는 방법
>
> 1. 향상된 for문 사용
> 2. Set 컬렉션의 iterator() 메소드로 반복자 얻어 객체 하나씩 가져오기

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");

        //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);
```

```
        }
    }
}
```

15.4 Map 컬렉션

> Map 컬렉션은 **키(key)와 값(value)으로 구성된 엔트리(Entry) 객체를 저장**한다. 키는 중복 저장할 수 없지만 값은 중복 저장할 수 있다. 기존에 저장된 키와 동일한 키로 값을 저장하면 기존의 값은 없어지고 새로운 값으로 대치된다. HashMap, Hashtable, LinkedHashMap, Properties, TreeMap 등이 있다. 키로 객체들을 관리하기 때문에 키를 매개값으로 갖는 메소드가 많다.

HashMap

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

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();

        //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();

        //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();
    }
}
```

Hashtable

- HashMap과 동일한 내부 구조.
- 차이점은 동기화된(synchronized) 메소드로 구성되어 있기 때문에 멀티 스레드가 동시에 Hashtable의 메소드를 실행할 수 없다는 것. **멀티 스레드 환경**에서도 안전하게 객체 추가, 삭제 가능.

Properties

- Hashtable의 자식 클래스.
- 키와 값을 `String` 타입으로 제한한 컬렉션.
- 주로 확장자가 *.properties*인 프로퍼티 파일 (키와 값이 = 로 연결되어 있는 텍스트 파일)을 읽을 때 사용.

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 파
일 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");

        //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);
    }
}
```

15.5 검색 기능을 강화시킨 컬렉션

TreeSet

> TreeSet은 **이진 트리(binary tree)** 를 기반으로 한 Set 컬렉션이다. TreeSet에 객체를 저장하면 부모 노드의 객체와 비교해서 낮은 것은 왼쪽 자식 노드에, 높은 것은 오른쪽 자식 노드에 자동으로 정렬하며 저장한다. TreeSet에만 정의되어 있는 검색 관련 메소드가 있어 TreeSet 타입 변수에 대입해야 한다.

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");
```

```
        //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");

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

결과

```
75 80 87 95 98

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

80 87
```

TreeMap

> TreeMap은 이진 트리를 기반으로 한 Map 컬렉션이다. TreeSet과의 차이점은 키와 값이 저장된 Entry를 저장한다는 점이다. TreeMap에 엔트리를 저장하면 키를 기준으로 자동 정렬되는데, 부모 키 값과 비교해서 낮은 것은 왼쪽, 높은 것은 오른쪽 자식 노드에 Entry 객체를 저장한다.

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);

        //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");

        //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());
        }
    }
}
```

결과

```
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

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
```

Comparable과 Comparator

> 객체가 정렬되기 위해서는 Comparable 인터페이스를 구현하고 있어야 한다. Integer, Double, String 타입은 모두 구현하고 있기 때문에 상관 없지만 사용자 정의 객체를 저장할 때에는 반드시 Comparable 인터페이스의 `compareTo()` 메소드를 재정의해야 한다. 리턴 값은 다음과 같다.

> - 주어진 객체와 같으면 **0**을 리턴
> - 주어진 객체보다 작으면 **음수**를 리턴
> - 주어진 객체보다 크면 **양수**를 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);
        }
    }
}
```

결과

```
감자바:25
박지원:31
홍길동:45
```

> 비교 기능이 없는 Comparable 비구현 객체를 저장하고 싶다면 TreeSet과 TreeMap을 생성할 때 비교자를 제공하면 된다. 비교자는 **Comparator** 인터페이스를 구현한 객체를 말하는데, Comparator 인터페이스에는 `compare()` 메소드가 정의되어 있다. 리턴 값은 다음과 같다.
>
> > - o1과 o2가 동등하다면 **0**을 리턴
> > - o1이 o2보다 앞에 오게 하려면 **음수**를 리턴
> > - o1이 o2보다 뒤에 오게 하려면 **양수**를 리턴

Fruit.java

```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

```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

```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);
        }
    }
}
```

결과

```
포도:3000
딸기:6000
수박:10000
```

15.6 LIFO와 FIFO 컬렉션

> **후입선출(LIFO[Last In First Out])** 은 나중에 넣은 객체가 먼저 빠져나가고, **선입선출(FIFO[First In First Out])** 은 먼저 넣은 객체가 먼저 빠져나가는 구조를 말한다. 컬렉션 프레임워크는 LIFO 자료구조를 제공하는 **스택(Stack)** 클래스와 FIFO 자료구조를 제공하는 **큐(Queue)** 인터페이스를 제공하고 있다.

Stack

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");
        }
    }
}
```

결과

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

Queue

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"));

        //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;
            }
        }
    }
}
```

결과

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

15.7 동기화된 컬렉션

> 동기화된 메소드로 구성되어 있지 않은 ArrayList, HashSet, HashMap을 멀티 스레드 환경에서 사용하고 싶을 때 컬렉션 프레임워크의 비동기화된 메소드를 동기화된 메소드로 래핑하는 Collections의 `synchronizedXXX()` 메소드를 사용한다.

SynchronizedMapExample.java

```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);
                }
            }
        };

        //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();

        //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();
    }
}
```

결과

```
총 객체 수: 2000
```

15.8 수정할 수 없는 컬렉션

> 수정할 수 없는(unmodifiable) 컬렉션이란 요소를 추가, 삭제할 수 없는 컬렉션을 말한다. 컬렉션 생성 시 저장된 요소를 변경하고 싶지 않을 때 유용하다.
>
> 1. List, Set, Map 인터페이스의 정적 메소드인 `of()`로 생성
> 2. List, Set, Map 인터페이스의 정적 메소드인 `copyOf()` 을 이용해 기존 컬렉션을 복사
> 3. 배열로부터 수정할 수 없는 List 컬렉션 생성

ImmutableExample.java

```java
package ch15.sec08;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

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)

        //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);

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