

12장. 제네릭 & 컬렉션

한림대학교 소프트웨어융합대학 양은샘.



이것이 자바다
신용권의 Java 프로그래밍 정복

저자 신용권
출판 한빛미디어 | 2015.1.5
페이지수 1,224 | 사이즈 183*235mm
판매가 **서적 27,000원**
구매이벤트 IT독자 설문이벤트 외 6건



혼자 공부하는 자바
JAVA 8 & 11 지원/무료 동영상 강의 제공

저자 신용권
출판 한빛미디어 | 2019.6.10.
페이지수 708 | 사이즈 188*257mm
판매가 **서적 21,600원**

12장. 제네릭 & 컬렉션

- ❖ 안녕하세요? 여러분!
- ❖ 오늘은 자바의 제네릭과 컬렉션 단원을 학습 합니다.
- ❖ 이번 장에서는
 - 제네릭을 왜 사용하는지, 사용할 때의 이점은 무엇인지 배웁니다.
 - 또한 여러가지 형태로 자료를 저장하는 컬렉션의 다양한 기능과 사용법에 대해 알아보도록 하겠습니다.
 - 제네릭과 컬렉션은 데이터를 다루어야 하는 작업환경에서 다양한 처리를 할 수 있는 장점을 제공합니다.
- ❖ 지난 시간에 학습한 내용을 리뷰한 후 학습을 시작하도록 하겠습니다.

지난 시간 Review

1절. Process와 Thread

2절. 작업 스레드 생성과 실행

3절. 스레드 스케줄링

4절. 스레드 동기화

5절. 스레드 상태

학습 목차

1절. 제네릭(Generic)

2절. 컬렉션(Collection)

3절. List 컬렉션

4절. Set 컬렉션

5절. Map 컬렉션

6절. 검색 기능을 강화시킨 컬렉션

7절. LIFO와 FIFO 컬렉션

8절. 동기화된(synchronized) 컬렉션

9절. 병렬 처리를 위한 컬렉션

학습 목표

- ❖ 제네릭의 의미와 사용법을 알고 적용할 수 있다.
- ❖ 배열이 가지는 불편함을 해결하기 위해 제공되는 컬렉션 프레임워크의 종류에 대해 안다.
- ❖ List 컬렉션, Set 컬렉션, Map 컬렉션의 차이점을 알고 응용할 수 있다.
- ❖ Stack 클래스와 Queue 인터페이스의 차이점을 알고 응용할 수 있다.
- ❖ 문제 상황에서 제네릭과 컬렉션의 필요성 여부를 판단할 수 있으며, 필요할 경우 제네릭과 컬렉션을 사용하여 해당 문제를 해결할 수 있다.

1절. 제네릭(Generic)

❖ 제네릭(Generic) 타입이란?

- 타입을 파라미터로 가지는 클래스와 인터페이스
- 선언 시 클래스 또는 인터페이스 이름 뒤에 "<>" 부호 붙임
- "<>" 사이에는 타입 파라미터가 위치 함

❖ 제네릭 타입 사용 여부에 따른 비교

- 클래스를 선언할 때 타입 파라미터 사용
- 컴파일 시 타입 파라미터가 구체적인 클래스로 변경 됨

```
public class Box<T> {  
    private T t;  
    public T get() { return t; }  
    public void set(T t) { this.t = t; }  
}
```

```
public class Box<String> {  
    private String t;  
    public void set(String t) { this.t = t; }  
    public String get() { return t; }  
}
```

```
Box<String> box = new Box<String>();  
box.set("hello");  
String str = box.get();
```

```
public class Box<Integer> {  
    private Integer t;  
    public void set(Integer t) { this.t = t; }  
    public Integer get() { return t; }  
}
```

```
Box<Integer> box = new Box<Integer>();  
box.set(6);  
int value = box.get();
```

제네릭 사용 코드의 이점

❖ 제네릭을 사용하는 코드의 이점

- 컴파일 시 강한 타입 체크가 가능해서 에러를 사전에 방지할 수 있음
- 컬렉션, 람다식(함수적 인터페이스), 스트림, NIO에서 널리 사용
- 제네릭을 모르면 API 문서 해석이 어려움
- 자바5부터 새로 추가됨

❖ 제네릭 타입을 사용하지 않은 경우

- Object 타입 사용 -> 빈번한 타입 변환 발생 -> 프로그램 성능 저하

```
public class Box {  
    private Object object;  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

```
Box box = new Box();  
box.set("hello");           //String 타입을 Object 타입으로 자동 타입 변환해서 저장  
String str = (String) box.get(); //Object 타입을 String 타입으로 강제 타입 변환해서 얻음
```

멀티 타입 파라미터

- ❖ 두 개 이상의 타입 파라미터 사용 가능
 - 각 타입 파라미터는 콤마로 구분
 - `class<K, V, ...> { ... }`
 - `interface<K, V, ...> { ... }`

```
public class Tv {  
}  
  
public class Product<T, M> {  
    private T kind;  
    private M model;  
  
    public T getKind() { return this.kind; }  
    public M getModel() { return this.model; }  
  
    public void setKind(T kind) { this.kind = kind; }  
    public void setModel(M model) { this.model = model; }  
}  
  
public class ProductExample {  
    public static void main(String[] args) {  
        Product<Tv, String> product1 = new Product<Tv, String>();  
        product1.setKind(new Tv());  
        product1.setModel("스마트Tv");  
  
        Tv tv = product1.getKind();  
        String tvModel = product1.getModel();  
    }  
}
```


제네릭 메소드

❖ 제네릭 메소드 선언 방법

- 매개변수 타입과 리턴 타입으로 타입 파라미터를 갖는 메소드
- 타입 파라미터를 리턴 타입과 매개변수에 사용
- 리턴 타입 앞에 "<>" 기호를 추가하고 타입 파라미터들을 나열

```
public <타입파라미터,...> 리턴타입 메소드명(매개변수,...) { ... }
```

```
public <T> Box<T> boxing(T t) { ... }
```

❖ 제네릭 메소드를 호출하는 두 가지 방법

```
리턴타입 변수 = <구체적타입> 메소드명(매개값); //명시적으로 구체적 타입 지정
```

```
리턴타입 변수 = 메소드명(매개값); //매개값을 보고 구체적 타입을 추정
```

```
Box<Integer> box = <Integer>boxing(100); //타입 파라미터를 명시적으로 Integer 로 지정
```

```
Box<Integer> box = boxing(100); //타입 파라미터를 Integer 으로 추정
```

제네릭 메소드 예

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey() { return key; }
    public V getValue() { return value; }
}

public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        boolean keyCompare = p1.getKey().equals(p2.getKey());
        boolean valueCompare = p1.getValue().equals(p2.getValue());

        return keyCompare && valueCompare;
    }
}
```

```
public class CompareMethodExample {
    public static void main(String[] args) {
        Pair<Integer, String> p1 = new Pair<Integer, String>(1, "사과");
        Pair<Integer, String> p2 = new Pair<Integer, String>(1, "사과");

        boolean result1 = Util.<Integer, String>compare(p1, p2);
        if(result1) {
            System.out.println("논리적으로 동등한 객체입니다.");
        } else {
            System.out.println("논리적으로 동등하지 않는 객체입니다.");
        }

        Pair<String, String> p3 = new Pair<String, String>("user1", "홍길동");
        Pair<String, String> p4 = new Pair<String, String>("user2", "홍길동");

        boolean result2 = Util.compare(p3, p4);
        if(result2) {
            System.out.println("논리적으로 동등한 객체입니다.");
        } else {
            System.out.println("논리적으로 동등하지 않는 객체입니다.");
        }
    }
}
```

제한된 타입 파라미터

- ❖ 타입 파라미터에 지정되는 구체적인 타입을 제한할 필요가 있을 때 사용
 - 상속 및 구현 관계를 이용해 타입을 제한하는 경우
 - 상위 타입은 클래스 뿐만 아니라 인터페이스도 가능

```
public <T extends 상위타입> 리턴타입 메소드(매개변수, ...) { ... }
```

- ❖ 타입 파라미터를 대체할 구체적인 타입
 - 상위, 하위 타입이거나 또는 구현 클래스만 지정 가능
- ❖ 와일드카드 타입의 세가지 형태
 - 제네릭타입<?> Unbounded Wildcards (제한 없음)
 - 제네릭타입<? extends 상위타입> Upper Bounded Wildcards (상위 클래스 제한)
 - 제네릭타입<? super 하위타입> Lower Bounded Wildcards (하위 클래스 제한)

제한된 타입 파라미터 예

```
public class Util {
    public static <T extends Number> int compare(T t1, T t2) {
        double v1 = t1.doubleValue();
        //System.out.println(t1.getClass().getName());

        double v2 = t2.doubleValue();
        //System.out.println(t2.getClass().getName());

        return Double.compare(v1, v2);
    }
}

public class BoundedTypeParameterExample {
    public static void main(String[] args) {
        //String str = Util.compare("a", "b"); (x)

        int result1 = Util.compare(10, 20);
        System.out.println(result1);

        int result2 = Util.compare(4.5, 3);
        System.out.println(result2);
    }
}
```

제네릭 타입의 상속과 구현

❖ 제네릭 타입을 부모 클래스로 사용할 경우

- 타입 파라미터는 자식 클래스에도 기술해야 함

```
public class ChildProduct<T, M> extends Product<T, M> { ... }
```

- 추가적인 타입 파라미터도 가질 수 있음

```
public class ChildProduct<T, M, C> extends Product<T, M> { ... }
```

❖ 제네릭 인터페이스를 구현할 경우

- 제네릭 인터페이스를 구현한 클래스도 제네릭 타입

2절. 컬렉션(Collection)

❖ 배열의 문제점

- 저장할 수 있는 객체 수가 배열을 생성할 때 결정 됨
 - 불특정 다수의 객체를 저장할 때 문제가 됨
- 객체를 삭제했을 때 해당 인덱스가 비게 됨
 - 객체를 저장하려면 어디가 비어 있는지 확인해야 함

배열

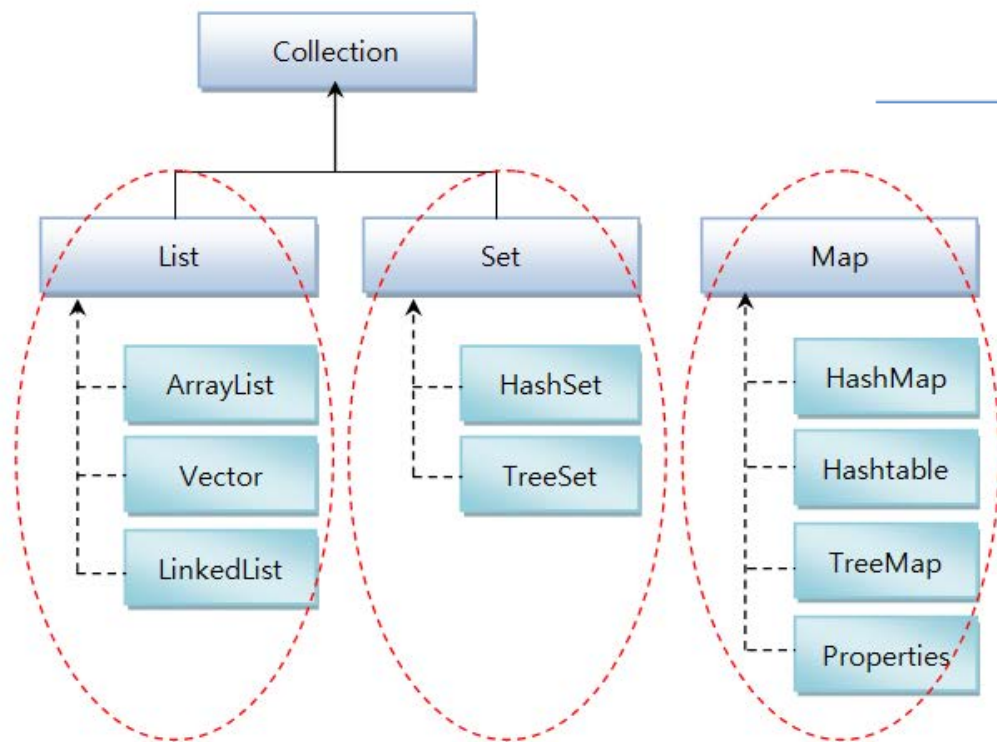
0	1	2	3	4	5	6	7	8	9
●	●	×	●	×	●	×	●	●	×

❖ 컬렉션 프레임워크(Collection Framework)

- 프레임워크 : 사용 방법을 정해놓은 라이브러리
- 자료구조를 사용해서 객체들을 효율적으로 관리할 수 있도록 인터페이스와 구현 클래스 제공
- java.util 패키지에서 제공 (추가, 삭제, 검색이 용이함)
- 주요 인터페이스로 List, Set, Map이 있음

컬렉션 프레임워크의 주요 인터페이스

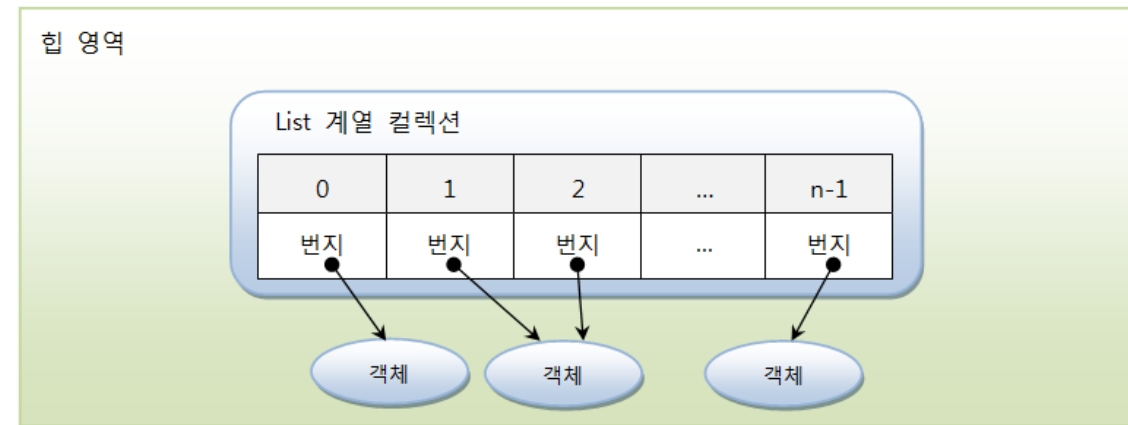
인터페이스 분류		특징	구현 클래스
Collection	List 계열	- 순서를 유지하고 저장 - 중복 저장 가능	ArrayList, Vector, LinkedList
	Set 계열	- 순서를 유지하지 않고 저장 - 중복 저장 안됨	HashSet, TreeSet
Map 계열		- 키와 값의 쌍으로 저장 - 키는 중복 저장 안됨	HashMap, Hashtable, TreeMap, Properties



3절. List 컬렉션

❖ List 컬렉션

- 객체를 인덱스로 관리
- 객체 저장 시 자동 인덱스가 부여
- 중복해서 객체 저장이 가능
- 객체 자체를 저장하는 것이 아닌 객체 번지를 참조
- 추가, 삭제, 검색 위한 다양한 메소드 제공



❖ 구현 클래스

- ArrayList
- Vector
- LinkedList

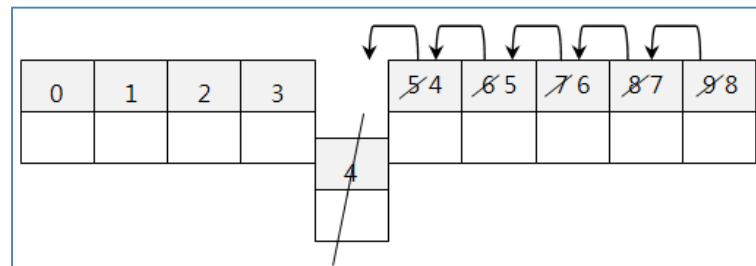
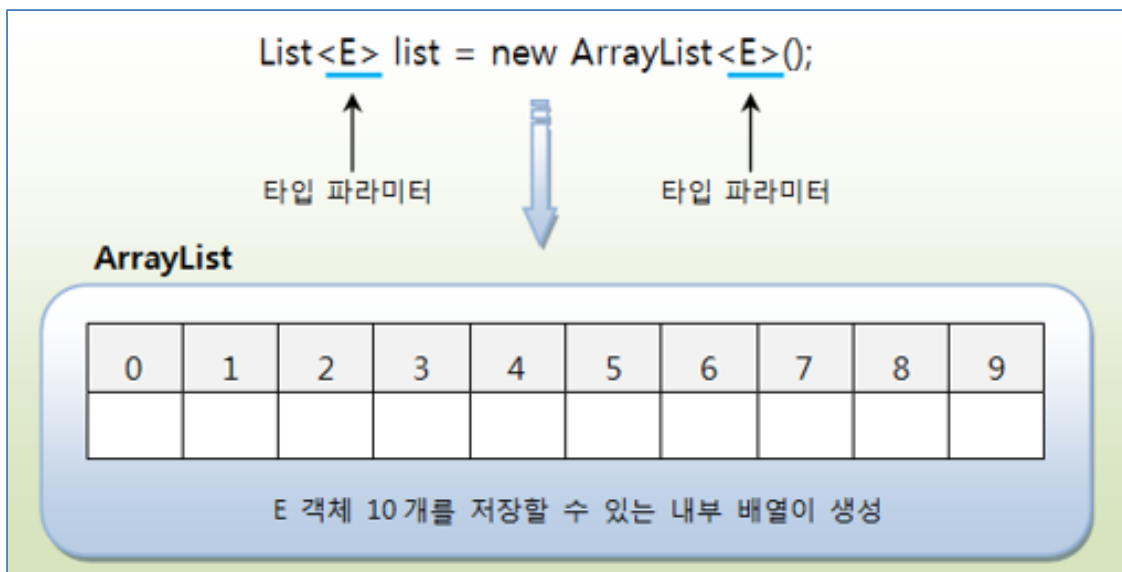
❖ 주요 메소드

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

ArrayList

❖ ArrayList

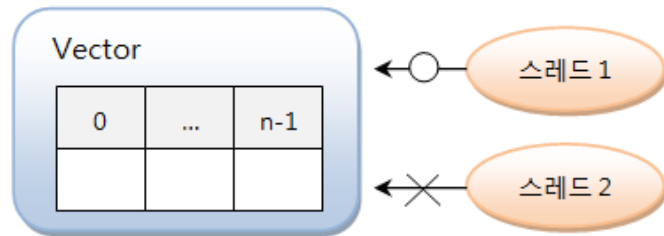
- List 인터페이스의 대표적 구현 클래스
- 초기 저장 용량(capacity) : 10 (따로 지정 가능)
- ArrayList에 객체 추가하면 0번 인덱스부터 차례로 저장
- 저장 용량을 초과한 객체들이 들어오면 자동적으로 늘어남. 고정도 가능.
- 객체 제거 시 바로 뒤 인덱스부터 마지막 인덱스까지 모두 앞으로 1씩 당겨 짐.



Vector

❖ Vector

- 멀티 스레드 환경에서 안전하게 객체 추가 삭제 가능
 - 동기화된 메소드로 구성 되어,
 - 하나의 스레드가 메소드 실행을 완료해야만,
 - 다른 스레드가 메소드를 실행할 수 있음.



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

Vector의 E 타입 파라미터를 생략하면
왼쪽 List에 지정된 타입을 따라 감

```
03 public class Board {  
04     String subject;  
05     String content;  
06     String writer;  
07  
08     public Board(String subject, String content, String writer) {  
09         this.subject = subject;  
10         this.content = content;  
11         this.writer = writer;  
12     }  
13 }  
05 public class VectorExample {  
06     public static void main(String[] args) {  
07         List<Board> list = new Vector<Board>();  
08  
09         list.add(new Board("제목1", "내용1", "글쓴이1"));  
10         list.add(new Board("제목2", "내용2", "글쓴이2"));  
11         list.add(new Board("제목3", "내용3", "글쓴이3"));  
12         list.add(new Board("제목4", "내용4", "글쓴이4"));  
13         list.add(new Board("제목5", "내용5", "글쓴이5"));  
14  
15         list.remove(2);  
16         list.remove(3);  
17  
18         for(int i=0; i<list.size(); i++) {  
19             Board board = list.get(i);  
20             System.out.println(board.subject + "\t" + board.content + "\t" + board.writer);  
21         }  
22     }  
23 }
```

Board 객체를 저장

2번 인덱스 객체(제목3) 삭제(뒤의 인덱스는 1씩 앞으로 당겨짐)

3번 인덱스 객체(제목5) 삭제

실행결과		
제목1	내용1	글쓴이1
제목2	내용2	글쓴이2
제목4	내용4	글쓴이4

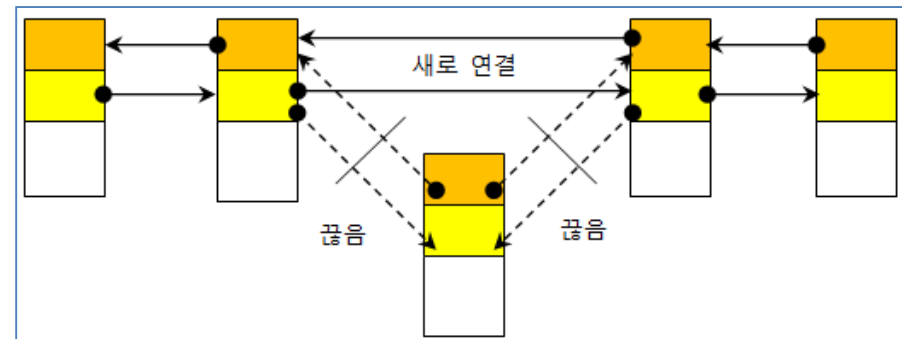
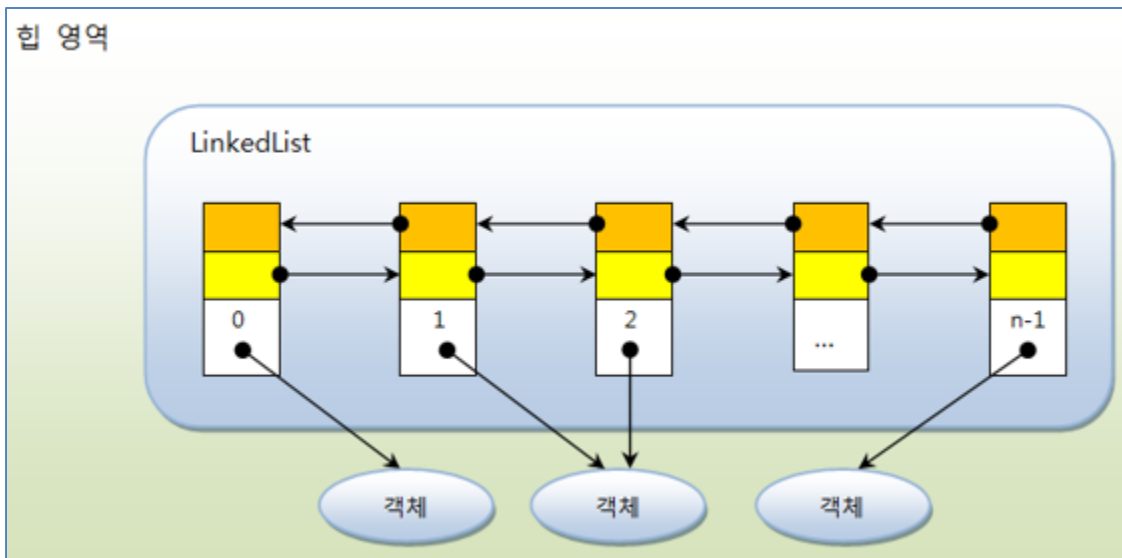
LinkedList 컬렉션 예

❖ LinkedList

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

LinkedList의 E 타입 파라미터를 생각하면
← 왼쪽 List에 지정된 타입을 따라 감

- 인접 참조를 링크해서 체인처럼 관리
- 특정 인덱스에서 객체를 제거 또는 추가하게 되면 바로 앞뒤 링크만 변경
- 객체 삭제와 삽입이 빈번히 일어나는 곳에서는 ArrayList보다 성능이 좋음



ArrayList LinkedList 성능 비교

```
03 import java.util.*;
04
05 public class LinkedListExample {
06     public static void main(String[] args) {
07         List<String> list1 = new ArrayList<String>();
08         List<String> list2 = new LinkedList<String>();
09
10         long startTime;
11         long endTime;
12
```

```
13         startTime = System.nanoTime();
14         for(int i=0; i<10000; i++) {
15             list1.add(0, String.valueOf(i));
16         }
17         endTime = System.nanoTime();
18         System.out.println("ArrayList 걸린시간: " + (endTime-startTime) + " ns");
19
20         startTime = System.nanoTime();
21         for(int i=0; i<10000; i++) {
22             list2.add(0, String.valueOf(i));
23         }
24         endTime = System.nanoTime();
25         System.out.println("LinkedList 걸린시간: " + (endTime-startTime) + " ns");
26     }
27 }
```

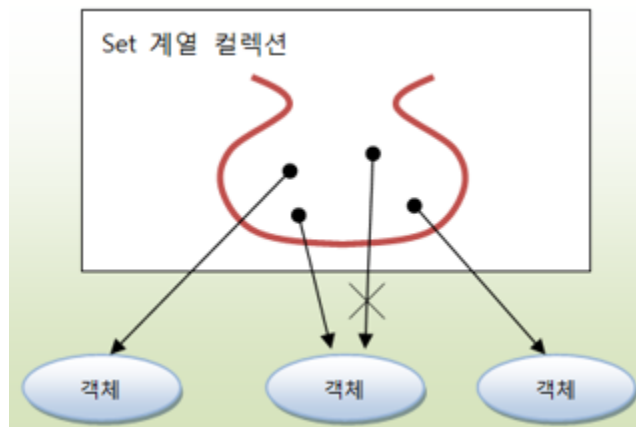
실행결과

ArrayList 걸린시간: 20248953 ns
LinkedList 걸린시간: 4279517 ns

4절. Set 컬렉션

❖ Set

- 수학의 집합과 같음
- 저장 순서가 유지되지 않음
- 객체의 중복 저장이 불가 함
- 하나의 null만 저장 가능



❖ 구현 클래스

- HashSet, LinkedHashSet, TreeSet

❖ 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
	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
객체 검색	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

Set : 검색

❖ 검색

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

❖ 반복자(iterator) 이용

- Iterator 인터페이스 메소드

리턴 타입	메소드	설명
boolean	hasNext()	가져올 객체가 있으면 true를 리턴하고 없으면 false를 리턴합니다.
E	next()	컬렉션에서 하나의 객체를 가져옵니다.
void	remove()	Set 컬렉션에서 객체를 제거합니다.

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

} 저장된 객체 수만큼 루핑

❖ 향상된 for문 이용

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

} 저장된 객체 수만큼 루핑

HashSet

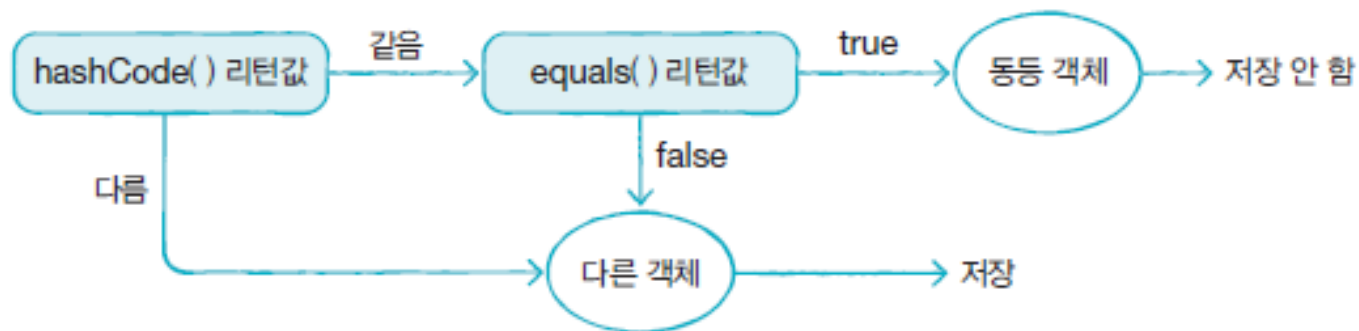
❖ HashSet

- 객체를 순서 없이 저장하되 동일 객체는 중복 저장하지 않음

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

HashSet의 E 타입 파라미터를 생각하면
← 왼쪽 Set에 지정된 타입을 따라 감

- 객체 저장 전 객체의 hashCode()로 해시코드를 얻어내고 이미 저장된 객체의 해시코드와 비교
- 동등 객체 판단 방법



객체를 중복 없이 저장하는 HashSet 예

```
03 import java.util.*;
04
05 public class HashSetExample {
06     public static void main(String[] args) {
07         Set<String> set = new HashSet<String>();
08
09         set.add("Java");
10         set.add("JDBC");
11         set.add("Servlet/JSP");
12         set.add("Java");
13         set.add("iBATIS");
14
15         int size = set.size();
16         System.out.println("총 객체수: " + size);
17     }
```

"Java"는 한 번만 저장됨

저장된 객체 수 얻기

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

반복자 얻기

객체 수 만큼 루핑

1개의 객체를 가져옴

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

1개의 객체 삭제

1개의 객체 삭제

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

저장된 객체 수 얻기

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

반복자 얻기

객체 수 만큼 루핑

```
34 set.clear();
35 if(set.isEmpty()) { System.out.println("비어 있음"); }
36 }
37 }
```

모든 객체를 제거하고 비움

실행결과

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

Member 객체를 중복 없이 저장 : hashCode(), equals() 재정의

```
03 public class Member {
04     public String name;
05     public int age;
06
07     public Member(String name, int age) {
08         this.name = name;
09         this.age = age;
10     }
11
12     @Override
13     public boolean equals(Object obj) { ← name과 age 값이 같으면 true를 리턴
14         if(obj instanceof Member) {
15             Member member = (Member) obj;
16             return member.name.equals(name) && (member.age==age) ;
17         } else {
18             return false;
19         }
20     }
21
22     @Override
23     public int hashCode() { ← name과 age 값이 같으면 동일한 hashCode를 리턴
24         return name.hashCode() + age;
25     } ← String의 hashCode() 이용
26 }
```

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

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

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

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

실행결과	×
총 객체수 : 1	

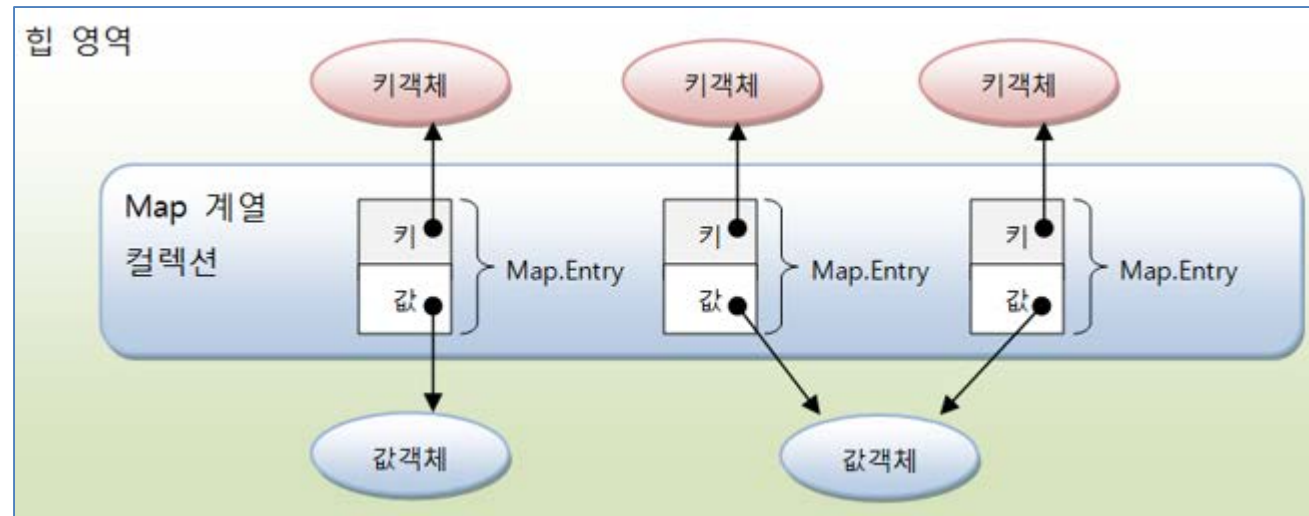
5절. Map 컬렉션

❖ Map 특징

- 키(key)와 값(value)으로 구성된 Map.Entry 객체를 저장하는 구조
- 키는 중복될 수 없지만, 값은 중복 저장 가능
- 기존 저장된 키와 동일한 키로 값을 저장하면 기존 값은 없어지고 새로운 값으로 대체
- 키와 값은 모두 객체

❖ 구현 클래스

- HashMap
- Hashtable
- LinkedHashMap
- Properties
- TreeMap



Map 인터페이스 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

Map : 검색

❖ 데이터 한 세트의 경우

- 키(key) 타입이 String, 값(value) 타입이 Integer인 Map 컬렉션 생성
- put()으로 키(key)와 값(value) 저장
- get(), remove() 사용

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

❖ 저장된 전체 객체 대상의 경우

- keySet()
 - keySet()으로 모든 키를 Set 컬렉션으로 얻은 뒤,
 - Iterator 반복자 통해 키(key)를 하나씩 얻고,
 - get()으로 값을 얻음
- entrySet()
 - entrySet()으로 모든 Map.Entry를 Set 컬렉션으로 얻은 뒤,
 - Iterator 반복자 통해 Map.Entry를 하나씩 얻고,
 - getKey()와 getValue()로 키(key)와 값(value) 얻음

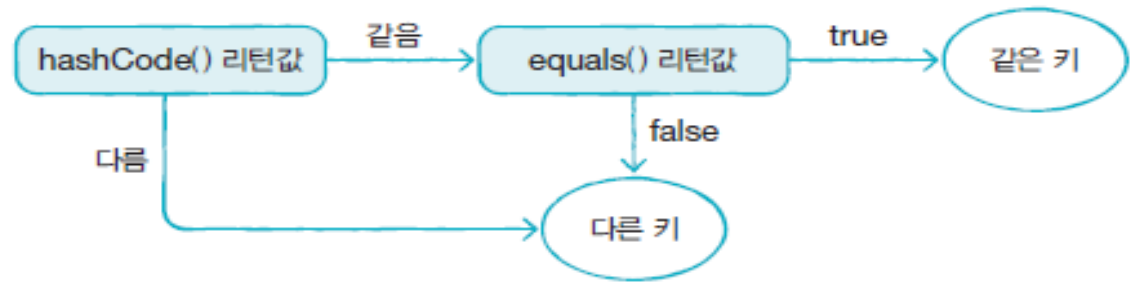
```
Map<K, V> map = ...;  
Set<K> keySet = map.keySet();  
Iterator<K> keyIterator = keySet.iterator();  
while(keyIterator.hasNext()) {  
    K key = keyIterator.next();  
    V value = map.get(key);  
}
```

```
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();  
}
```


HashMap

❖ HashMap은 대표적인 Map 컬렉션

- 키 객체는 hashCode()와 equals() 를 재정의해 동등 객체가 될 조건을 정해야 함.
- hashCode()의 리턴값이 같고, equals()가 true를 리턴해야 함.



❖ HashMap을 생성하려면

- 키 타입과 값 타입을 타입 파라미터로 주고 기본
- 키 타입은 String을 많이 사용
- String은 문자열이 같을 경우 동등 객체가 될 수 있도록 hashCode()와 equals()가 재정의되어 있기 때문

```
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<>();
```

HashMap의 K와 V 타입 파라미터를 생각하면
왼쪽 Map에 지정된 타입을 따라 감

HashMap 예

```
03 import java.util.HashMap;
04 import java.util.Iterator;
05 import java.util.Map;
06 import java.util.Set;
07
08 public class HashMapExample {
09     public static void main(String[] args) {
10         //Map 컬렉션 생성
11         Map<String, Integer> map = new HashMap<String, Integer>();
12
13         //객체 저장
14         map.put("신용권", 85);
15         map.put("홍길동", 90);
16         map.put("동장군", 80);
17         map.put("홍길동", 95);
18         System.out.println("총 Entry 수: " + map.size());
19
20         //객체 찾기
21         System.out.println("\t홍길동 : " + map.get("홍길동"));
22         System.out.println();
```

"홍길동" 키가 같기 때문에
제일 마지막에 저장한 값으로 대체

실행결과

총 Entry 수: 3
홍길동 : 95
홍길동 : 95
신용권 : 85
동장군 : 80
총 Entry 수: 2
신용권 : 85
동장군 : 80
총 Entry 수: 0

//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();
```

← Key Set 얻기

반복해서 키를 얻고
값을 Map에서 얻어냄

//객체 삭제

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

← 키로 Map.Entry를 제거

//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();
```

← Map.Entry Set 얻기

반복해서
Map.Entry를 얻고
키와 값을 얻어냄

//객체 전체 삭제

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

← 모든 Map.Entry 삭제

HashMap 예 : hashCode(), equals() 재정의

```
03 class Student {
04     public int sno;
05     public String name;
06
07     public Student(int sno, String name) {
08         this.sno = sno;
09         this.name = name;
10     }
11
12     public boolean equals(Object obj) { ← 학번과 이름이 같다면 true를 리턴
13         if(obj instanceof Student) {
14             Student student = (Student) obj;
15             return (sno==student.sno) && (name.equals(student.name));
16         } else {
17             return false;
18         }
19     }
20
21     public int hashCode() { ← 학번과 이름이 같다면 동일한 값을 리턴
22         return sno + name.hashCode();
23     }
24 }
```

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

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

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

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

실행결과

총 Entry 수: 1

Hashtable

❖ HashMap과 동일한 내부 구조

- 키로 사용할 객체를 hashCode()와 equals()로 재정의하여 동등 객체가 될 조건을 정해야 함.

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

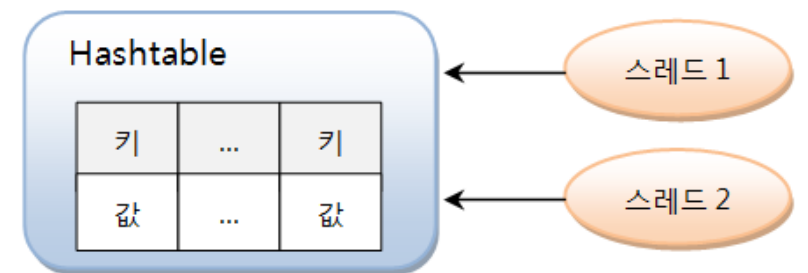
키 타입 값 타입 키 타입 값 타입

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

Hashtable의 K와 V 타입
파라미터를 생각하면 왼쪽
Map에 지정된 타입을 따라 감

❖ 동기화된 메소드로 구성

- 멀티 스레드가 동시에 Hashtable 메소드 실행할 수 없으며,
- 하나의 스레드가 실행을 완료해야만 다른 스레드 실행할 수 있음.
- 복수의 스레드가 동시에 Hashtable에 접근해서 객체를 추가, 삭제하더라도 안전(thread safe) 함.



스레드 동기화 적용됨

Hashtable 예

```
03 import java.util.*;
04
05 public class HashtableExample {
06     public static void main(String[] args) {
07         Map<String, String> map = new Hashtable<String, String>();
08
09         map.put("spring", "12");
10         map.put("summer", "123");
11         map.put("fall", "1234");
12         map.put("winter", "12345");
13
14         Scanner scanner = new Scanner(System.in);
15
16         while(true) {
17             System.out.println("아이디와 비밀번호를 입력해주세요.");
18             System.out.print("아이디: ");
19             String id = scanner.nextLine();
20         }
```

아이디와 비밀번호를 미리 저장

키보드로부터 입력된
내용을 받기 위해 생성

키보드로 입력한 아이디를 읽음

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

if(map.containsKey(id)) {
    if(map.get(id).equals(password)) {
        System.out.println("로그인되었습니다.");
        break;
    } else {
        System.out.println("비밀번호가 일치하지 않습니다.");
    }
} else {
    System.out.println("입력하신 아이디가 존재하지 않습니다");
}
```

키보드로 입력한
비밀번호를 읽음

아이디인 키가 존재하는지 확인

비밀번호를 비교

실행결과

```
아이디와 비밀번호를 입력해주세요
아이디: summer
비밀번호: 123

로그인되었습니다.
```

Properties

❖ 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

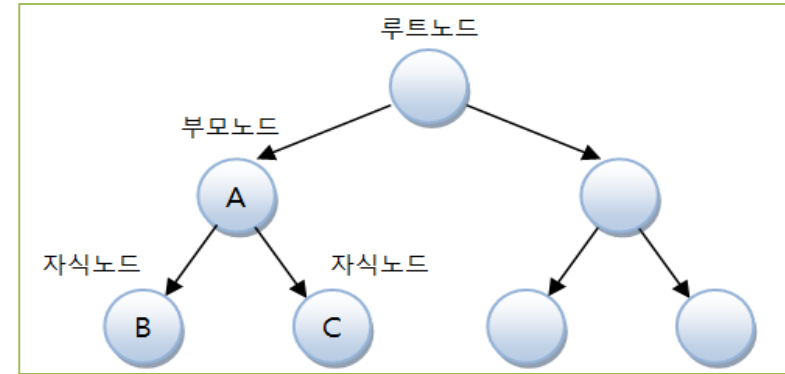
6절. 검색 기능을 강화시킨 컬렉션

❖ 검색 기능을 강화시킨 컬렉션 (계층 구조 활용)

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

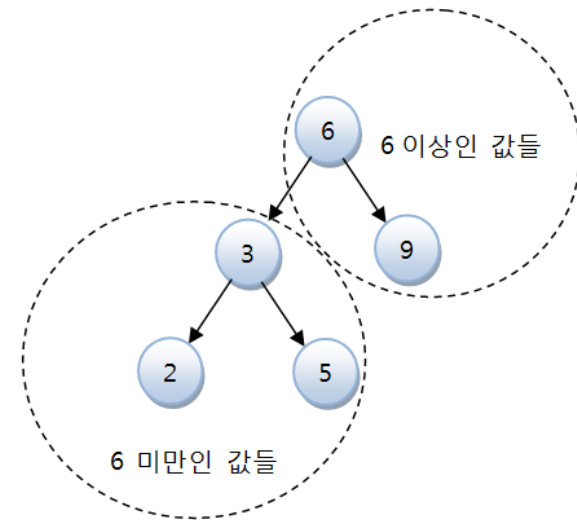
❖ 이진 트리 구조

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



❖ 정렬 방법

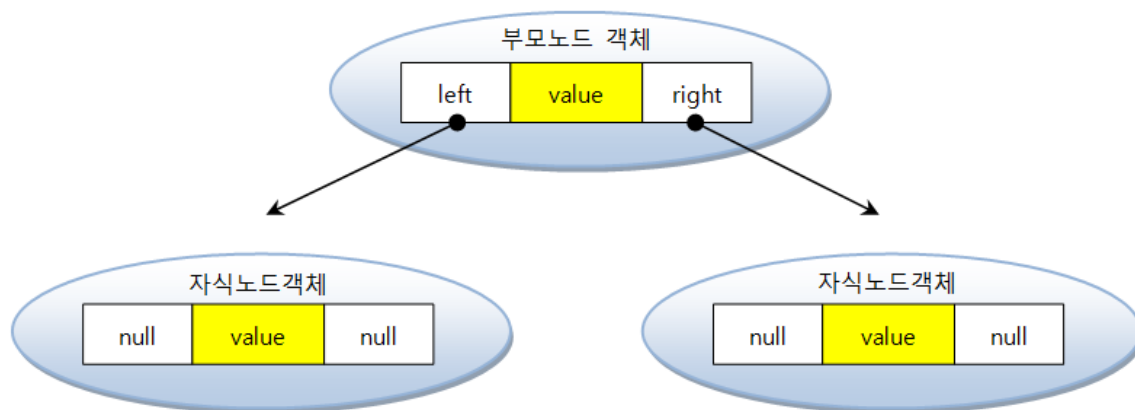
- 올림 차순: [왼쪽노드→부모노드→오른쪽노드]
- 내림 차순: [오른쪽노드→부모노드→왼쪽노드]



TreeSet

❖ TreeSet 특징

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



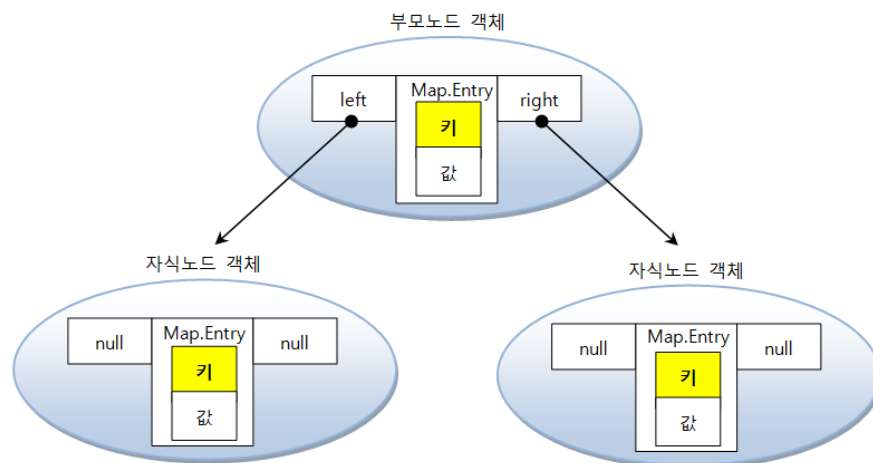
❖ 주요 메소드

- 특정 객체를 찾는 메소드 : `first()`, `last()`, `lower()`, `higher()`, ...
- 정렬 메소드 : `descendingIterator()`, `descendingSet()`
- 범위 검색 메소드 : `headSet()`, `tailSet`, `subSet()`

TreeMap

❖ 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



❖ 주요 메소드

- 단일 노드 객체를 찾는 메소드 : `firstEntry()`, `lastEntry()`, `lowerEntry()`, `higherEntry()`, ...
- 정렬 메소드 : `descendingKeySet()`, `descendingMap()`
- 범위 검색 메소드 : `headMap()`, `tailMap()`, `subMap()`

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 정렬을 위해 java.lang.Comparable을 구현한 객체를 요구
 - Integer, Double, String은 모두 Comparable 인터페이스 구현
 - Comparable을 구현하고 있지 않을 경우에는 저장하는 순간 ClassCastException 발생

7절. LIFO와 FIFO 컬렉션

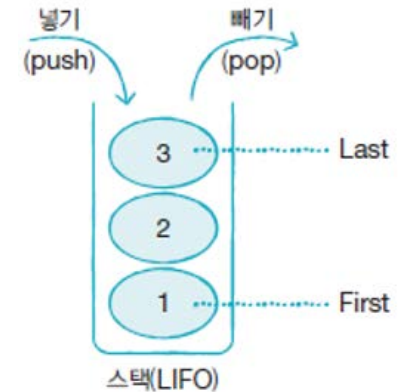
❖ 후입선출 (LIFO : Last In First Out) : 나중에 넣은 객체가 먼저 빠져나가는 자료구조

■ Stack 클래스

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

Stack의 E 타입 파라미터를 생각하면
← 왼쪽 Stack에 지정된 타입을 따라 감

리턴 타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣습니다.
E	peek()	스택의 맨 위 객체를 가져옵니다. 객체를 스택에서 제거하지 않습니다.
E	pop()	스택의 맨 위 객체를 가져옵니다. 객체를 스택에서 제거합니다.



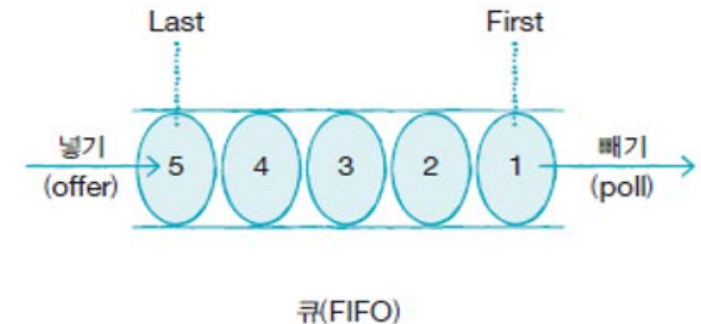
❖ 선입선출 (FIFO : First In First Out) : 먼저 넣은 객체가 먼저 빠져나가는 자료구조

■ Queue 인터페이스

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

LinkedList의 E 타입 파라미터를 생각하면
← 왼쪽 Queue에 지정된 타입을 따라 감

리턴 타입	메소드	설명
boolean	offer(E e)	주어진 객체를 넣습니다.
E	peek()	객체 하나를 가져옵니다. 객체를 큐에서 제거하지 않습니다.
E	poll()	객체 하나를 가져옵니다. 객체를 큐에서 제거합니다.



Stack 클래스 예

```
03 public class Coin {
04     private int value;
05
06     public Coin(int value) {
07         this.value = value;
08     }
09
10     public int getValue() {
11         return value;
12     }
13 }
```



```
05 public class StackExample {
06     public static void main(String[] args) {
07         Stack<Coin> coinBox = new Stack<Coin>();
08
09         coinBox.push(new Coin(100));
10         coinBox.push(new Coin(50));
11         coinBox.push(new Coin(500));
12         coinBox.push(new Coin(10));
13
14         while(!coinBox.isEmpty()) {
15             Coin coin = coinBox.pop();
16             System.out.println("꺼내온 동전 : " + coin.getValue() + "원");
17         }
18     }
19 }
```

← 동전을 끼움

← 동전 케이스가 비었는지 확인

← 동전 케이스에서 제일 위의 동전을 꺼냄

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

Queue 인터페이스 예

```
03 public class Message {
04     public String command;
05     public String to;
06
07     public Message(String command, String to) {
08         this.command = command;
09         this.to = to;
10     }
11 }
```

```
03 import java.util.LinkedList;
04 import java.util.Queue;
05
06 public class QueueExample {
07     public static void main(String[] args) {
08         Queue<Message> messageQueue = new LinkedList<Message>();
09
10         messageQueue.offer(new Message("sendMail", "홍길동"));
11         messageQueue.offer(new Message("sendSMS", "신용권"));
12         messageQueue.offer(new Message("sendKakaotalk", "홍두깨"));
13
14         while(!messageQueue.isEmpty()) {
15             Message message = messageQueue.poll();
16             switch(message.command) {
17                 case "sendMail":
18                     System.out.println(message.to + "님에게 메일을 보냅니다.");
19                     break;
20                 case "sendSMS":
21                     System.out.println(message.to + "님에게 SMS를 보냅니다.");
22                     break;
23                 case "sendKakaotalk":
24                     System.out.println(message.to + "님에게 카카오톡을 보냅니다.");
25                     break;
26             }
27         }
28     }
29 }
```

← 메시지 저장

← 메시지 큐가 비었는지 확인

← 메시지 큐에서 1개의 메시지 꺼냄

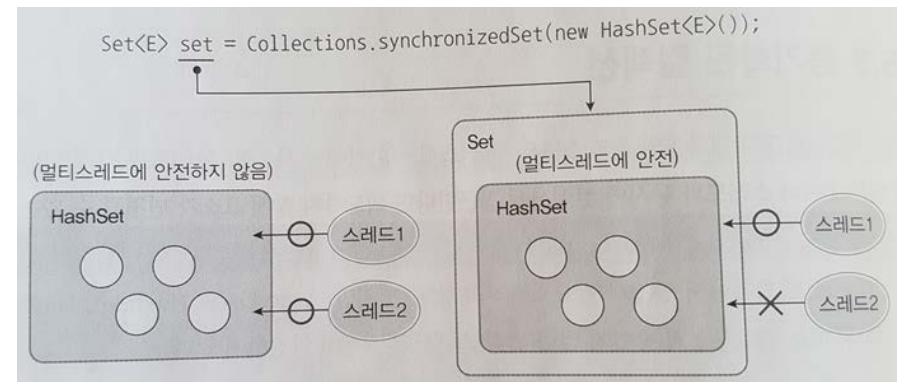
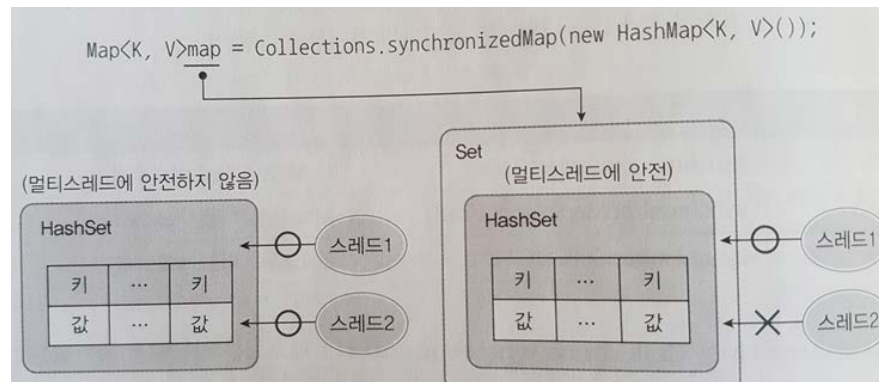
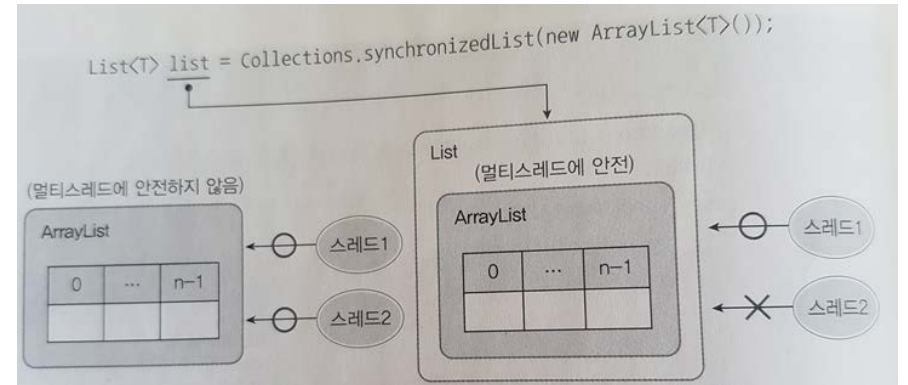
실행결과

홍길동님에게 메일을 보냅니다.
신용권님에게 SMS를 보냅니다.
홍두깨님에게 카카오톡을 보냅니다.

8절. 동기화된(synchronized) 컬렉션

- ❖ 비 동기화된 컬렉션을 동기화된 컬렉션으로 매핑
 - Collections의 synchronizedXXX() 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



9절. 병렬 처리를 위한 컬렉션

❖ 동기화(Synchronized) 컬렉션의 단점

- 하나의 스레드가 요소를 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
- 처리하는 요소가 포함된 부분(segment)만 잠금 사용
- 나머지 부분은 다른 스레드가 변경 가능

❖ 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

학습 정리 1

❖ 제네릭(Generic)

- 클래스와 인터페이스에 자료형 타입을 파라미터로 가질 수 있게 구성하여, 컴파일 시 강한 타입 체크를 가능하게 해 에러를 사전에 방지할 수 있는 기능이다.

❖ 컬렉션(Collection) 프레임워크

- 자료구조 사용하여 객체를 효율적으로 추가, 삭제, 검색할 수 있도록 인터페이스와 구현 클래스를 java.util 패키지에서 제공하는데 이들을 총칭하여 컬렉션 프레임워크라 한다.

❖ List 컬렉션

- List 컬렉션은 배열과 비슷하게 객체를 인덱스로 관리한다. 차이점은 저장용량이 자동으로 증가하며, 객체 저장 시 자동으로 인덱스가 부여된다. 또한 추가, 삭제, 검색을 위한 다양한 메소드가 제공된다.

❖ Set 컬렉션

- Set 컬렉션은 저장 순서가 유지되지 않으며, 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있다.

학습 정리 2

❖ Map 컬렉션

- Map 컬렉션은 키와 값으로 구성된 Map.Entry 객체를 저장하는 구조를 가지고 있으며, 여기서 키와 값은 모두 객체이다. 키는 중복 저장될 수 없지만 값은 중복 저장될 수 있다.

❖ Stack : FILO을 구현한 클래스.

❖ Queue : FIFO 정의한 인터페이스.

학습 정리 3

- ❖ 자바의 컬렉션 프레임워크에 대한 설명으로 맞는 것에 O, 틀린 것에 X 하세요
 - List 컬렉션은 인덱스로 객체를 관리하며 중복 저장을 허용한다. ()
 - Set 컬렉션은 순서를 유지하지 않으며 중복 저장을 허용하지 않는다. ()
 - Map 컬렉션은 키와 값으로 구성된 Map.Entry를 저장한다. ()
 - List와 Set은 모두 하나의 null만 저장 가능하다. ()

- ❖ Stack과 Queue에 대한 설명으로 맞는 것에 O, 틀린 것에 X 하세요
 - Stack은 후입선출을 구현한 클래스이다. ()
 - Queue는 선입선출을 위한 인터페이스이다. ()
 - Stack의 push()는 객체를 넣을 때, pop()은 객체를 뺄 때 사용한다. ()
 - Queue의 poll()은 객체를 넣을 때, offer()은 객체를 뺄 때 사용한다. ()

학습 정리 4

- ❖ 싱글 스레드 환경에서 Board 객체를 저장 순서에 맞게 읽고 싶습니다. 가장 적합한 컬렉션을 생성하도록 밑줄 친 부분에 코드를 적성해보세요.

_____ 변수 = new _____
(타입) (생성자 호출)

- ❖ 싱글 스레드 환경에서 학번(String)을 키로, 점수(Integer)를 값으로 저장하고 싶습니다. 가장 적합한 컬렉션을 생성하도록 밑줄 친 부분에 코드를 작성해보세요.

_____ 변수 = new _____
(타입) (생성자 호출)

학습 정리 5

- ❖ HashSet에 Student 객체를 저장하려 합니다. 학번이 같으면 동일한 Student라고 가정하고 중복 저장이 되지 않도록 하고 싶습니다. Student의 해시코드는 학번이라 가정합니다. Student 클래스에서 재정의해야 하는 hashCode()와 equals() 메소드의 내용을 채워보세요.

```
03 import java.util.HashSet;
04 import java.util.Iterator;
05 import java.util.Set;
06
07 public class HashSetExample {
08     public static void main(String[] args) {
09         Set<Student> set = new HashSet<Student>();
10
11         set.add(new Student(1, "홍길동"));
12         set.add(new Student(2, "신용권"));
13         set.add(new Student(1, "조민우")); <----- 학번이 같으므로 저장되지 않음
14
15         Iterator<Student> iterator = set.iterator();
16         while(iterator.hasNext()) {
17             Student student = iterator.next();
18             System.out.println(student.studentNum + ":" + student.name);
19         }
20     }
21 }
```

실행결과

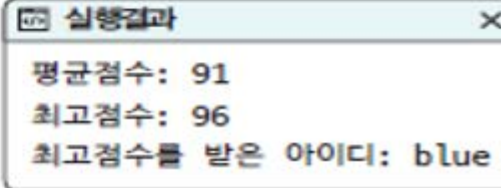
1: 홍길동
2: 신용권

```
03 public class Student {
04     public int studentNum;
05     public String name;
06
07     public Student (int studentNum, String name) {
08         this.studentNum = studentNum;
09         this.name = name;
10     }
11
12     @Override
13     public int hashCode() {
14         //코드 작성
15     }
16
17     @Override
18     public boolean equals(Object obj) {
19         //코드 작성
20     }
21 }
```

학습 정리 6

- ❖ HashMap에 아이디(String)와 점수(Integer)가 저장되어 있습니다. 실행 결과와 같이 평균 점수를 출력하고 최고 점수와 최고 점수를 받은 아이디를 출력해보세요

```
03 import java.util.HashMap;
04 import java.util.Map;
05 import java.util.Set;
06
07 public class MapExample {
08     public static void main(String[] args) {
09         Map<String,Integer> map = new HashMap<String,Integer>();
10         map.put("blue", 96);
11         map.put("hong", 86);
12         map.put("white", 92);
13
14         String name = null;    //최고 점수를 받은 아이디 저장
15         int maxScore = 0;      //최고 점수 저장
16         int totalScore = 0;    //점수 합계 저장
17
18         //작성 위치
19     }
20 }
```



실행결과

평균점수: 91
최고점수: 96
최고점수를 받은 아이디: blue

적용 확인 학습 & 응용 프로그래밍

- ❖ 다음 파일에 있는 문제들의 해답을 스스로 작성 해 보신 후 개념 & 적용 확인 학습 영상을 학습 하시기 바랍니다.
 - java_12장_제네릭_컬렉션_ex.pdf
- ❖ 퀴즈와 과제가 출제되었습니다.
 - 영상 수업을 학습하신 후 과제와 퀴즈를 수행 하시기 바랍니다.

Q & A

- ❖ “제네릭과 컬렉션”에 대한 학습이 모두 끝났습니다.
- ❖ 모든 내용을 이해 하셨나요?
- ❖ 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- ❖ 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.



- ❖ 퀴즈와 과제가 출제되었습니다. 마감시간에 늦지 않도록 주의해 주세요.
- ❖ 다음 시간에는 “파일입출력”을 공부하도록 하겠습니다.
- ❖ 수고하셨습니다.^^