10장. 제네릭

10.1. 제네릭

- 제네릭(Generic) 이란 타입을 파라미터화하여 컴파일시 구체적인 타입이 결정되도록 하는 것
- 제네릭을 사용함으로써 타입변환을 제거할 수 있으며, 컴파일시 타입을 사전에 체크할 수 있음

제네릭 개요

제네릭 문법

Exam_01

❖ 제네릭은 자바5부터 시작되었으며, (1.5버젼)
객체 생성시 객체의 타입을 미리 정의하는 것을 의미함

```
List list = new ArrayList( );
list.add ("Human");
String str = (String) list.get(0);  // 강제타입변환
```

```
List <String> list = new ArrayList<String>( );
list.add ("Human");
String str = list.get(0);       // 변환타입이 생략됨
```

❖ list 객체에 String만 ADD 하여야 컴파일시 에러가
발생하지 않음으로 컴파일 시점에 사전 체크 가능함.

제네릭 사용사례

Exam_02

❖ class 선언시 클래스의 Type을 선언함으로써
데이터 정합성 유지 가능

클래스 선언

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

T➔ String

```
Box<String> box
 = new Box<String>;
box.set("Human");
String str = box.get(0);
```

❖ String이 아닌
숫자가 들어올 경우
컴파일 에러 발생

T➔ Integet

객체 생성

```
Box<Integer> box
 = new Box<Integer>;
box.set(6);
Integer val = box.get(0);
```

❖ Integer이 아닌
문자가 들어올 경우
컴파일 에러 발생

10.2. 제네릭 활용

- 제네릭은 Multi Type으로 파라미터를 사용해서 선언할 수 있음

-

제네릭 활용

Multi Type 파라미터 사용

❖ 클래스 및 인터페이스에서 내부적으로 사용할 변수 및 메서드의 리턴타입에 대해서 멀티 타입으로 사용 가능
❖ 아래의 예제는 Product Class 하나로 여러종류의 객체를 표현할 수 있는 Generic 예제.

Product 클래스

```
public class Product <T, M> {
  private T type;
  private M model;

  public T getType( ) { return this.type; }
  public M getModel( ) { return this.model; }

  public void setType(T type) {
    this.type = type;
  }
  public void setModel(M model) {
    this.model = model;
  }
}
```

실행 클래스

```
public class ProductExam  {
  public static void main(String[ ] args) {
    Product<Tv, String> prod1 = new Product<Tv, String>( ) ;
    prod1.setType(new Tv("삼성"));        // TV 객체로 등록
    prod1.setModel("Smart TV");           // 모델명 String 등록
    Tv tv = prod1.getType( );
    String str1 = prod1.getModel( );

    Product<Car, String> prod2 = new Product<Car, String>( ) ;
    prod2.setType(new Car("현대"));       // Car 객체로 등록
    prod2.setModel("Sports car");         // 모델명 String 등록
    Car car = prod2.getType( );
    String str2 = prod2.getModel( );
  }
}
```

11장. 컬렉션

11.1.1. Collection Framework의 발생 배경

- 배열의 한계점을 극복하기 위해 Collection Framework 발생함.

-

배열 사용의 한계점

❖ 내가 가지고 있는 제품을 Product 객체배열로 관리한다고 가정하면 아래와 같이 할 수 있음

```
Product[ ] prod = new Product[10];
// 객체 추가
prod[0] = new Product("TV");
prod[1] = new Product("HandSet");
prod[2] = new Product("Car");

Prod[1] = null;                    // 객체 삭제

// 객체 조회
for (int i=0 ; i<prod.length ; i++) {
try { System.out.println(prod[i].name); }
catch (Exception e) { System.out.println(e); }
}
```

0	1	2	3	4	5	6	7	8	9
TV	X	Car	X	X	X	X	X	X	X

1. 처음부터 배열의 크기를 지정해야 하나?
➔ 필요할 때마다 늘리고 싶은데..
➔ X 부분은 메모리 낭비임.

2. 중복되는 데이터를 관리할 수 있나?
➔ 기존의 Data가 있는지 없는지 모두 확인해야 함.
➔ 꼭 순서가 있어야 하는지?

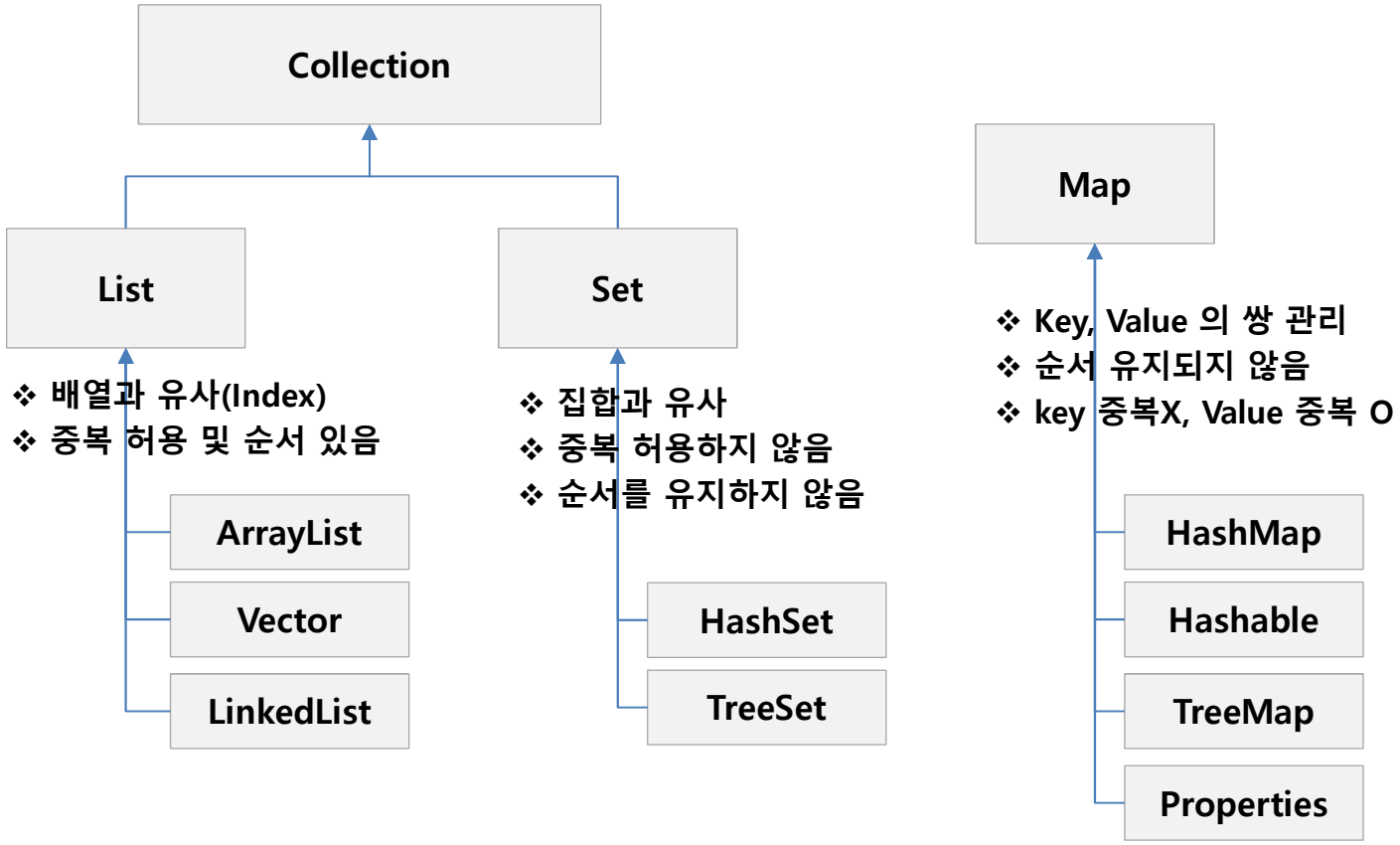
3. 0~9까지의 인덱스가 아니라. 전자제품, 자동차 등의 속성값으로 관리 가능한지?
➔ Product['Car'] = "그랜져"
➔ Product["TV"] = "X-Canvas"

11.1.2. Collection

- 컬렉션(Collection) 이란 객체(요소)를 수집해서 저장하는 것.
- java.util 패키지에 포함되고 있으며, 인터페이스를 통해 다양한 컬랙션 클래스 이용가능

컬렉션 개요

❖ 컬렉션은 저장, 삭제, 검색, 정렬 등을 편하기 다루게 해주는 프레임워크 (JAVA.UTIL.*)



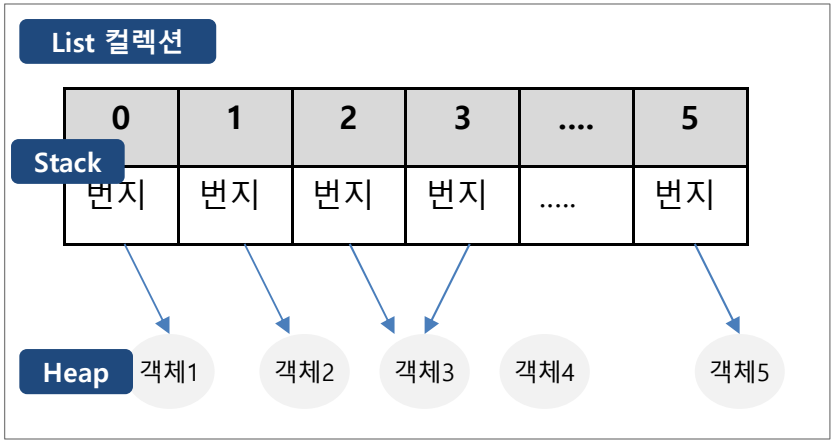
11.2.1. List Collection

- List 컬렉션은 인덱스로 관리되어 순서가 있으며, 중복해서 객체 저장됨.
- 구현 클래스 : ArrayList, Vector, LinkedList가 있음.

List 컬렉션 개요

List 컬렉션 개요

❖ list 인터페이스에서 정의된 Generic 타입의 객체를 중복을 허용하면서 순서를 가진 데이터 저장




```
List <String> list = new ArrayList<String> ( );
list.add("휴먼");       // list 맨 마지막에 객체 삽입
list.add(1, "교육");    // 지정된 인덱스에 객체삽입
String str1 = list.get(0);   // "휴먼"
list.remove(0);              // 지정된 위치의 객체 삭제
                             // index=0에는 "교육"이 있음
list.remove("교육");         // 동일한 객체 삭제
```

List 메서드 설명

❖ List 인터페이스는 객체의 추가, 검색, 삭제의 메서드를 포함하고 있음.

기능	메서드	설명
객체 추가	Boolean add (E e)	객체를 맨 끝에 추가
	Void add(int index, E e)	index위치에 객체추가
	E set(int index, E e)	Index위치에 객체변경
객체 검색	Boolean contains(Object o)	주어진 객체 포함여부
	E get(int index)	Index의 객체 리턴
	Boolean isEmpty( )	컬렉션 empty 여부
	Int size( )	컬렉션 길이
객체 삭제	Void clear( )	컬렉션 지움
	E remove(int index)	Index 위치 지움
	Boolean remove(Object o)	객체 삭제

❖ remove 후 index는 뒤에서 앞으로 한칸씩 이동.

11.2.2. List Collection 구현체 (ArrayList / LinkedList)

- ArrayList는 List의 구현클래스이며, 객체를 인덱스로 관리됨.
- LinkedList도 List 구현클래스이며 , 객체를 인접 참조를 링크하여 체인처럼 관리

ArrayList 및 LinkedList

ArrayList 개요

Exam_02

❖ index를 지정하여 선언가능 (default 10)
❖ 배열과 비슷하나 index가 자동적으로 늘어남

```
List <String> list1 = new ArrayList<String>( );
  // default 초기용량 10
List <String> list2 = new ArrayList<String>(100);
  // 100개의 index를 가진 arraylist
List <String> list3 = Arrays.asList("a", "b", "c", "d");
  // 컬렉션을 포함하여 객체 생성. 단 위의 경우 고정객체
```

❖ 객체 제거시 제거된 index부터 앞으로 1씩 당겨옴

```
List <String> list = new ArrayList<String>();
List.add("a"); List.add("b"); List.add("c"); List.add("d");
list.remove(1);      // 1위치인 "b" 제거.

 // list.get(0) ➔ a
 // list.get(1) ➔ c  (1씩 당겨와서 1의 위치에 있음)
 // list.get(2) ➔ d  (1씩 당겨와서 2의 위치에 있음)
```

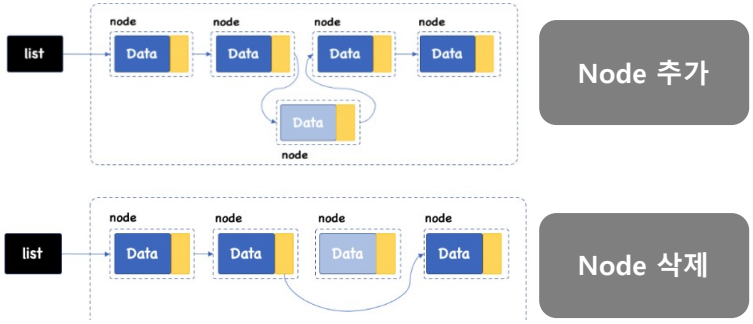
LinkedList 개요

Exam_03

❖ 불연속적인 데이터를 추가/삭제시 시간이 오래걸리는 배열의 단점을 보완 (추가/삭제시 재배열 필요없음)
❖ Node와 Node간의 Link를 가지고 있는 List임
❖ ArrayList와 사용법은 동일하나 객체의 삽입, 삭제가 빈번할 경우 처리속도가 빠름.

List 메서드 그대로 사용

```
LinkedList <String> list = new LinkedList<String>( );
list.add("Data1");
list.add(1, "Data2");       // 1의 위치에 반영
list.addFirst("Data3");     // list의 가장 앞에 반영
list.addLast("Data4");      // list의 가장 뒤에 반영
```



11.2.3. List Collection 구현체 (Vector)

- Vector는 ArrayList와 동일함.
- 단, Multi-Thread상에서 동시에 접속하는 것을 막아주고 있음.

Vector

11.3.1. Set Collection

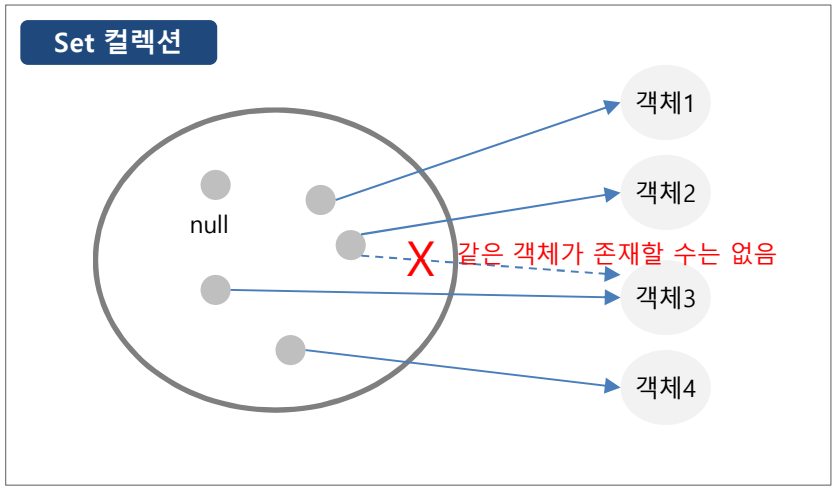
- Set 컬렉션은 수학의 집합과 같은 개념으로 중복이 허용되지 않으며, 순서도 없음
- 구현 클래스 : HashSet, LinkedHashSet, TreeSet

Set 컬렉션 개요

Set 컬렉션 개요

Exam_04

❖ 중복 허용안됨, 저장순서 유지 안됨
❖ null은 1번만 존재함.



Set 메서드 설명

❖ Set 인터페이스는 객체의 추가, 검색, 삭제의 메서드를 포함하고 있음.
❖ List 인터페이스 중 index 관련 메서드가 없음

기능	메서드	설명
추가	Boolean add (E e)	객체 추가
객체 검색	Boolean contains(Object o)	주어진 객체 포함여부
	Boolean isEmpty( )	컬렉션 empty 여부
	Iterator <E> iterator( )	객체 가져오는 반복자 인터페이스
	Int size( )	컬렉션 길이
객체 삭제	Void clear( )	컬렉션 지움
	Boolean remove(Object o)	객체 삭제

Iterator 사례

```
Set <String> set = new HashSet<String> (
Iterator<String> iter = set.iterator( );
while ( iter.hasNext( ) ) {    // set에 남아 있으면 true
  String str = iter.next( );    // set에 있는 수만큼 loop
}
```

❖ **Iterator 인터페이스 메서드**
- boolean hasNext( ) : 객체가 있으면 true, 없으면 false
- E next( ) : 컬렉션 내에서 하나의 객체를 가져옴
- void remove( ) : set 컬렉션에서 객체를 제거함.

11.3.2. Set Collection (HashSet)

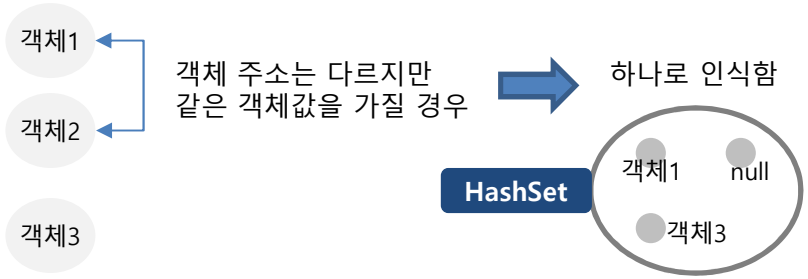
- HashSet은 Set의 구현클래스이며, HashCode() 메서드 활용하여 기존에 있는지를 사전 검사하여 Set에 객체의 추가여부를 판단함.

HashSet

HashSet 개요

Exam_04

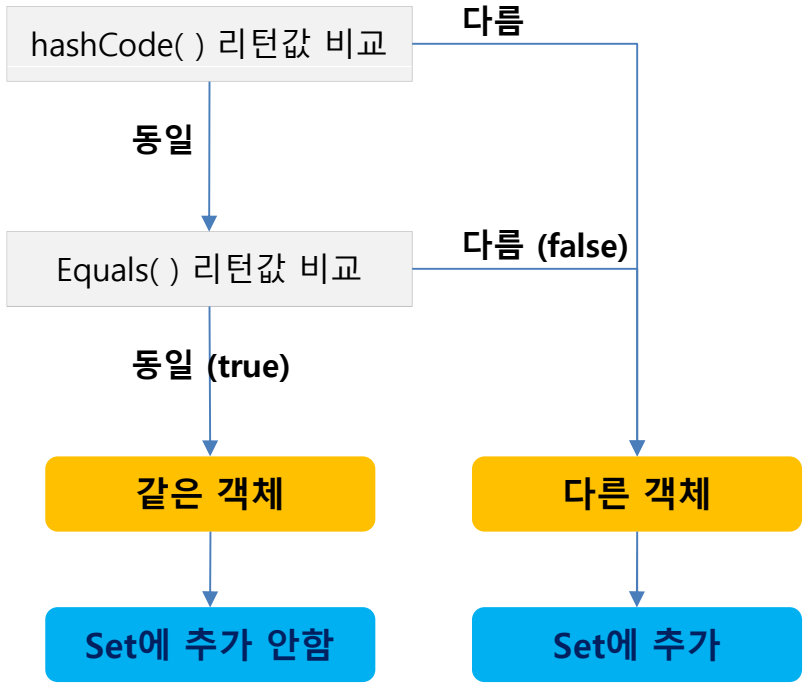
❖ 객체의 값은 같지만 객체의 번지가 다른 경우 활용
❖ HashCode를 활용하여 해시코드 얻음.
그리고, 이미 저장된 해시코드와 비교하여 같은 것이 있으면 동일 객체로 판단하고 저장하지 않음




```
Set <String> set = new HashSet<String> ( );
set.add("휴먼");      // 순서없이 저장됨
set.add("교육");      // 추가적으로 저장됨.
set.add("교육");      // 같은 데이터이므로 Skip 됨
set.remove("교육");      // "교육" 객체 삭제됨.
```

HashSet의 동작원리

❖ 객체 추가(add 메서드 호출) 시점에 hashCode를 활용
❖ 그 후 equals를 통해 내부의 값도 같은지 확인함.
❖ Set에 추가여부는 아래의 로직으로 처리됨



11.3.2. Set Collection (HashSet)

- HashSet의 hashCode / equals의 메서드 재정의를 통해 별도로 생성된 객체의 동일여부를 판단할 수 있으며, 이를 통해 객체가 중복되지 않게 관리할 수 있음

HashSet Overriding 구현 (참조형 변수의 경우)

Member 클래스

```
public String name;
public int age;
public Member (String name, int age) {
  this.name = name;
  this.age = age;
}
public int hashCode( ) {
  return name.hashCode() + age;
}
public boolean equals (Object obj) {
  if (obj instanceof Member ) {
    Member mem = (Member) obj;
    return (mem.name.equals(name))
           && ((mem.age ==age));
  }
  else {
  return false;
  }
}
```

name 의 hash코드 비교
- 같을 경우 equals 메서드 실행
- 다를 경우 set에 추가

입력된 인자가 Member 클래스의 객체일 경우만 점검함.

name의 값과 age의 값이 같을 경우 true를 반환하여 동일함으로 판단하여 Set 내에 신규로 생성하지 않음.

실행 클래스

```
public static void main(String[] agrs) {
    Set<Member> set = new HashSet<Member>();
    Member e1 = new Member ("Human", 30);
    Member e2 = new Member ("Human", 30);

    set.add (e1);
    set.add (e2);
    System.out.println("총 객체수 : " + set.size());
    System.out.println(set);
}
```

실행 결과

```
총 객체수 : 1
[ExamHashSet_01@42d712b]
```

만약 hashCode와 equals를 구현하지 않으면.
개별적으로 등록이 됨.
(왜냐하면 객체 주소는 서로 다르기 때문에)

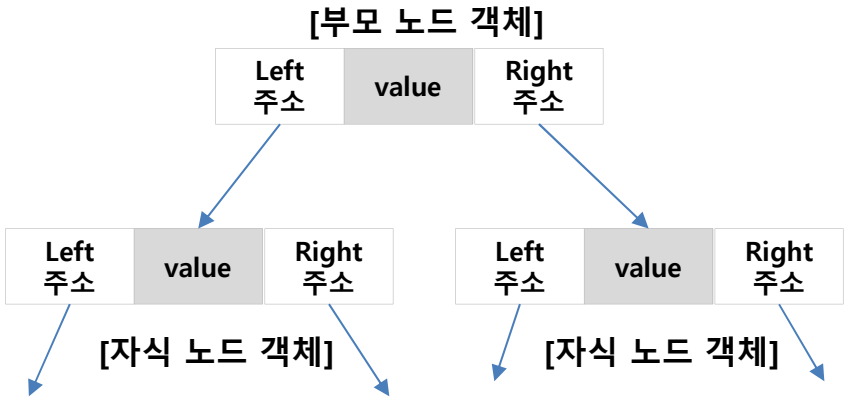
11.3.3. Set Collection (TreeSet)

- TreeSet은 이진트리(Binary tree)를 기반으로 한 Set 컬렉션임.
- TreeSet에 객체를 저장하면 부모값과 비교하여 왼쪽, 오른쪽 노드로 구분하여 저장됨.

TreeSet

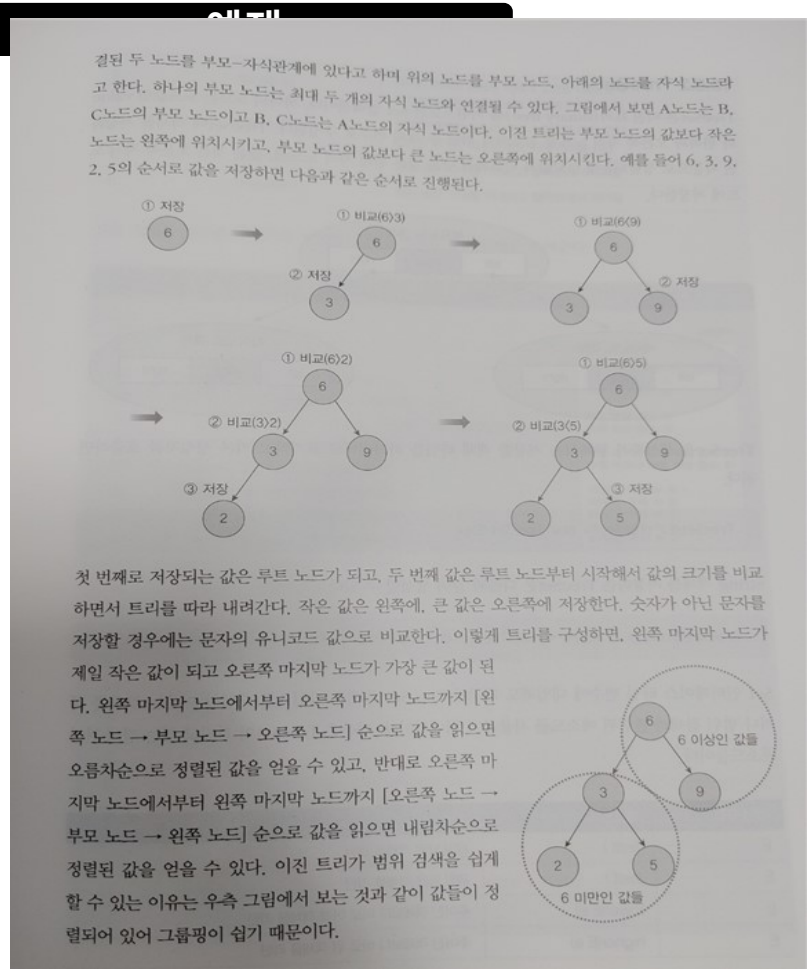
TreeSet 개요

❖ 부모노드 객체보다 작은 것은 왼쪽 자식노드에 저장
❖ 부모노드 객체보다 큰 것은 오른쪽 자식노드에 저장



❖ TreeSet의 선언 방법

```
TreeSet<E> ts = new TreeSet<E>( );
   // E는 객체타입을 의미함.

TreeSet<String> ts = new TreeSet<String>( );
   // 이의 경우 String 객체로 TreeSet이 생성됨.
```



11.3.3. Set Collection (TreeSet)

- TreeSet은 이진트리(Binary tree)를 기반으로 한 Set 컬렉션임.
- TreeSet에 객체를 저장하면 부모값과 비교하여 왼쪽, 오른쪽 노드로 구분하여 저장됨.

TreeSet

TreeSet 구현메서드

❖ TreeSet은 Set 인터페이스를 구현한 것으로
검색, 정렬, 범위 등의 메서드가 있음

기능	메서드	설명
객체 검색	E first( )	가장 낮은 객체
	E last( )	가장 높은 객체
	E lower(E e)	주어진 인자의 아래 객체
	E higher(E e)	주어진 인자의 위 객체
	E floor(E e)	주어진 인자의 동일 객체 만약 없으면 아래 객체
	E ceiling(E e)	주어진 인자의 동일 객체 만약 없으면 위 객체
	E pollFirst (E e)	가장 낮은 객체선정 후 제거
	E pollLast (E e)	가장 높은 객체선정 후 제거
정렬	descendingSet( )	내림차순의 정렬
객체 범위	headSet(E e, bool i)	e보다 작은 객체 모두 반환
	tailSet(E e, bool i)	e보다 큰 객체 모두 반환
	subSet(E e1, E e2, bool i)	e1~e2 사이 객체 반환

11.3.3. Set Collection (TreeSet 활용)

- TreeSet은 이진트리(Binary tree)를 기반으로 한 Set 컬렉션임.
- TreeSet에 저장된 값을 검색, 정렬, 범위 지정등에 용이함.

TreeSet 활용

TreeSet 실습

```
TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(new Integer(81)); ts.add(new Integer(82));
ts.add(new Integer(70)); ts.add(new Integer(89));
ts.add(new Integer(85));

System.out.println ("first() : "+ts.first());
System.out.println ("last() : "+ts.last());
System.out.println ("lower(new Integer(71)) : "+ts.lower(new Integer(71)));
System.out.println ("higher(new Integer(71)) : "+ts.higher(new Integer(71)));
System.out.println ("floor(new Integer(71)) : "+ts.floor(new Integer(71)));
System.out.println ("ceiling(new Integer(71)) : "+ts.ceiling(new Integer(71)));
System.out.println ("floor(new Integer(81)) : "+ts.floor(new Integer(81)));
System.out.println ("ceiling(new Integer(82)) : "+ts.ceiling(new Integer(82)));

NavigableSet<Integer> desc = ts.descendingSet();
NavigableSet<Integer> asc = desc.descendingSet();
System.out.println ("desc : "+ desc);
System.out.println ("asc : "+ asc);

NavigableSet<Integer> bet1 = ts.headSet(new Integer(82), true);
NavigableSet<Integer> bet2 = ts.headSet(new Integer(82), false);
NavigableSet<Integer> bet3 = ts.tailSet(new Integer(81), true);
NavigableSet<Integer> bet4 = ts.tailSet(new Integer(81), false);
System.out.println ("headSet(T) : "+ bet1);
System.out.println ("headSet(F) : "+ bet2);
System.out.println ("tailSet(T) : "+ bet3);
System.out.println ("tailSet(F) : "+ bet4);10
```

결과

70, 81, 82, 85, 89

```
first() : 70                        // 가장 작은 수
last() : 89                         // 가장 큰 수
lower(new Integer(71)) : 70         // 71보다 작은 수
higher(new Integer(71)) : 81        // 71보다 큰 수
floor(new Integer(71)) : 70         // 71과 같거나 작은 수
ceiling(new Integer(71)) : 81       // 71과 같거나 큰 수
floor(new Integer(81)) : 81          // 81과 같거나 작은 수
ceiling(new Integer(82)) : 82        // 82과 같거나 큰 수

desc : [89, 85, 82, 81, 70]         // 내림차순 정렬
asc : [70, 81, 82, 85, 89]           // 오름차순 정렬

headSet(T) : [70, 81, 82]           // 인자 포함 아래의 수들
headSet(F) : [70, 81]               // 인자 미포함 아래의 수들
tailSet(T) : [81, 82, 85, 89]       // 인자 포함 위의 수들
tailSet(F) : [82, 85, 89]           // 인자 미포함 위의 수들
```

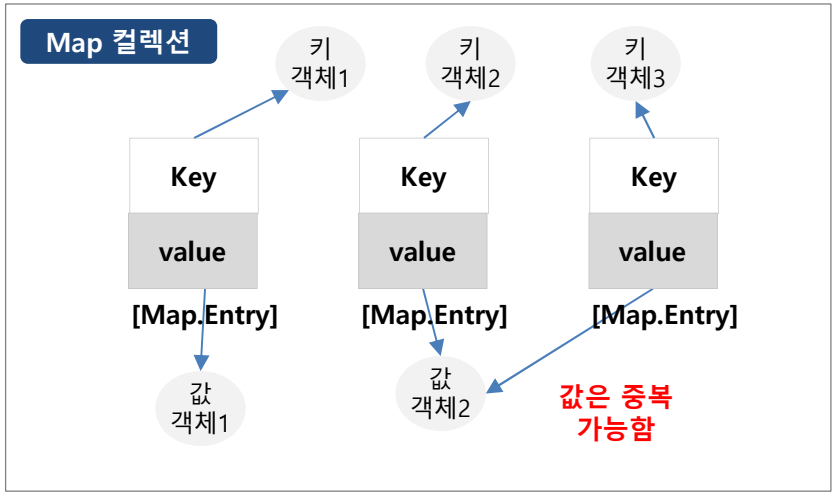
11.4.1. Map Collection

- Map 컬렉션은 [Key : Value]의 쌍으로 구성된 Entry 객체를 저장함. (Key, Value 모두 객체임)
- 구현 클래스 : HashMap, TreeMap, HashTable, Properties이 있음.

Map 컬렉션 개요

Map 컬렉션 개요

❖ Map은 Key, Value의 쌍으로 이루어짐.
❖ Key 중복 안됨, Value는 중복 허용함.
❖ 기존 Key에 Value 대입시 기존 Value는 사라짐.



Map 메서드 설명

❖ Map에서 사용하는 메서드이며, Key 기준으로 객체관리 (key는 중복되지 않기 때문임)

기능	메서드	설명
객체 추가	V put (K key, V val)	주어진 키와 값으로 등록
객체 검색	Boolean contains(Object key)	주어진 키의 포함여부
	Boolean contains(Object val)	주어진 값의 포함여부
	V get(Object key)	키기준으로 리턴
	Boolean isEmpty( )	비어있는지 검사
	Int size( )	저장된 키의 수
객체 삭제	Void clear( )	컬렉션 지움
	V remove(Object key)	동일 키 삭제

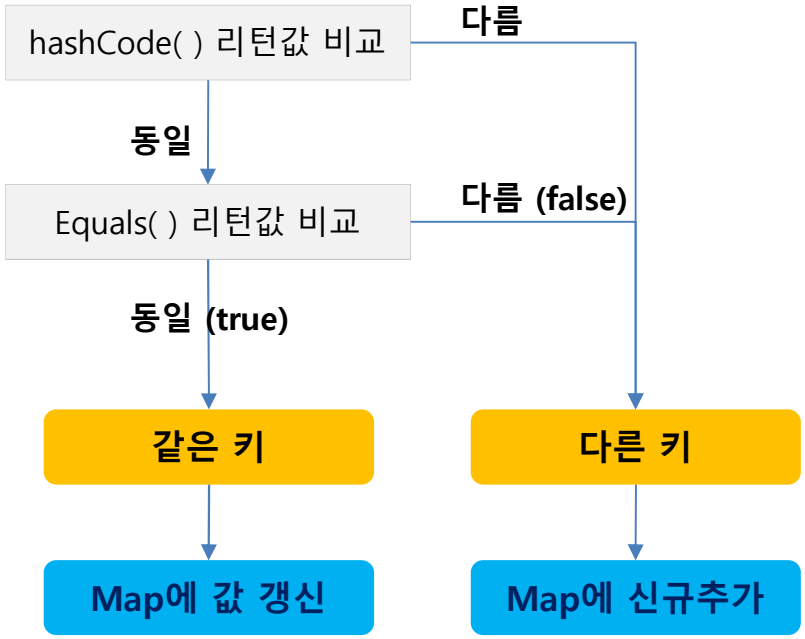
11.4.2. Map Collection (HashMap)

- HashMap은 Map의 구현클래스이며, HashCode() 메서드 활용하여 기존에 있는지를 사전 검사하여 key의 중복을 관리한다.

HashMap 개요 및 예제

HashMap 개요

- ❖ 객체 추가(put 메서드 호출) 시점에 hashCode를 활용
- ❖ 그 후 equals를 통해 내부의 값도 같은지 확인함.
- ❖ Map에 추가 및 갱신 여부는 아래의 로직으로 처리됨
- ❖ 동일 key일 경우는 Value를 갱신함.



HashMap의 예제

```
Map <String, Integer> map = new HashMap<String, Integer>();
map.put("human1", 100);  map.put("human2", 90);
map.put("human3", 80);  map.put("human4", 70);
System.out.println("map.size() : " + map.size()); // 4가 출력됨
System.out.println("human2의 값 : " + map.get("human2")); // 90

Set <String> ks = map.keySet();  // set으로 map의 key 얻어옴
Iterator<String> iter = ks.iterator(); // keyset의 반복자
while (iter.hasNext()) {
  String k = iter.next();          // 반복하면서 key을 얻어옴
  Integer v = map.get(k);          // 반복하면서 key의 값을 얻어옴
  System.out.println("KEY, VALUE : " + k + " - " + v );
}
map.remove("human3");              // human3의 key, value 제거
System.out.println("map.size() : " + map.size());  // 3 출력됨

// 아래는 map.Entry를 얻어와 처리하는 방법
Set<Map.Entry<String, Integer>> es = map.entrySet(); //
Iterator<Map.Entry<String, Integer>> esIter= es.iterator();

while (esIter.hasNext()) {
  Map.Entry<String, Integer> nextEntry = esIter.next();
  String k = nextEntry.getKey();
  Integer v = nextEntry.getValue();
  System.out.println("KEY, VALUE : " + k + " - " + v );
}
map.clear();                       // map내의 객체를 모두 제거함.
System.out.println("map.size() : " + map.size());
```

11.4.2. Map Collection (HashMap)

- HashMap은 Map의 구현클래스이며, HashCode() 메서드 활용하여 기존에 있는지를 사전 검사하여 key의 중복을 관리한다.

HashMap 의 hash코드 사용 예제

Student 클래스

```
public class Student {

  public int id;
  public String name;

  public Student(int id, String name) {
    this.id = id;
    this.name = name;
  }
  public int hashCode() {
    return id + name.hashCode();
  }
  public boolean equals (Object o) {
    if (o instanceof Student) {
      Student sd = (Student) o;
      // id는 기본형 변수이므로 바로 비교 가능
      // name은 String으로 참조형 변수이므로
      // equals를 통해서 비교함.
      return (id==sd.id) && (name.equals(sd.name));
    }
    else return false;
  }
}
```

HashMap 실습

```
public static void main (String[] args) {

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

  // Student 객체를 생성하여 map에 등록함.
  map.put(new Student(1, "human1"), 100);
  map.put(new Student(2, "human2"), 90);
  // 중복된 객체이므로 현재의 값으로 갱신됨
  map.put(new Student(2, "human2"), 70);

  System.out.println ("map Size : " + map.size( )); // 2 출력
  System.out.println (map.get(new Student(1,"human1")));
              // 100이 출력됨
  System.out.println (map.get(new Student(2,"human2")));
              // 90이 아닌 70이 출력됨.
}
```

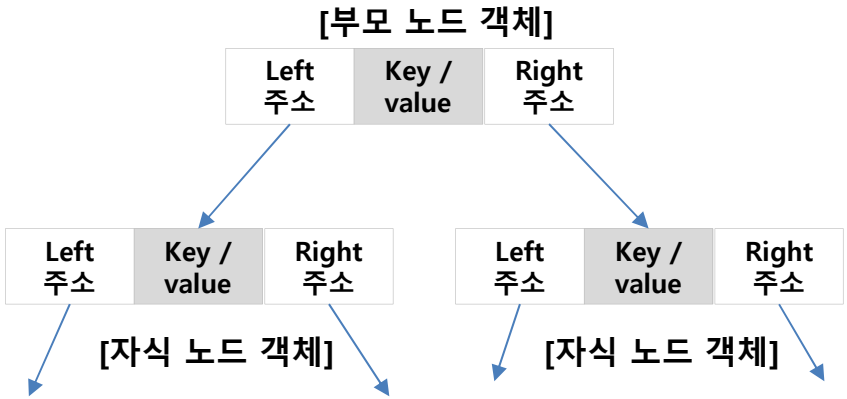
11.4.3. Map Collection (TreeMap)

- TreeMap은 Map의 구현클래스이며, 이진트리를 기반으로 함.
- 부모의 키값과 비교하여 왼쪽 및 오른쪽의 자식노드에 Map.Entry 객체를 생성함.

TreeMap 개요

TreeMap 개요

❖ 부모노드 키 값보다 작은 것은 왼쪽 자식노드에 저장
❖ 부모노드 키 값보다 큰 것은 오른쪽 자식노드에 저장



❖ TreeMap의 선언 방법

```
TreeMap<K, V> tm = new TreeMap<K, V>( );
   // K는 key 객체, V는 value 객체를 의미함.

TreeMap<String, Integer> tm
          = new TreeMap<String, Integer>( );
   // String 객체로 Key, Integer 객체로 Value 생성됨.
```

TreeMap 구현 메서드

❖ TreeMap은 Map 인터페이스를 구현한 것으로 검색, 정렬, 범위 등의 메서드가 있음

기능	메서드	설명
객체 검색	firstEntry( )	가장 낮은 Map.Entry 리턴
	LastEntry( )	가장 높은 Map.Entry 리턴
	lowerEntry(K k	주어진 key 바로 아래 map.entry
	higherEntry(K k)	주어진 key 바로 위 map.entry
	floor(K k)	주어진 Ket와 동일 map.Entry 만약 없으면 아래 리턴
	ceiling(K k)	주어진 Ket와 동일 map.Entry 만약 없으면 위 리턴
	pollFirst (K k)	가장 낮은 map.entry 선정 후 제거
	pollLast (K k)	가장 높은 map.entry 선정 후 제거
정렬	descendingKeySet( ) : Key 기준 내림차순	
객체 범위	HeadKeySet(K k, bool I) : k보다 작은 map.entry 반환	
	tailKeySet(K k, bool i)	
	subKeySet(K k1, K k2, bool i)	

11.4.3. Map Collection (TreeMap)

- 점수(Key), 이름(Value)의 기준으로 TreeMap을 확인하는 실습

TreeMap 실습

TreeMap 개요

```
TreeMap<Integer, String> tm = new TreeMap<Integer, String>();
tm.put(new Integer(80), "human1"); tm.put(new Integer(70), "human2");
tm.put(new Integer(99), "human3"); tm.put(new Integer(90), "human4");

Map.Entry<Integer, String> me = null;
System.out.println("tm.firstEntry().getKey() + " - " + tm.firstEntry().getValue());          // 70 - human2 출력
System.out.println("tm.lastEntry().getKey() + " - " + tm.lastEntry().getValue());            // 99 - human3 출력
System.out.println("tm.lowerEntry(82).getKey() + " - " + tm.lowerEntry(82).getValue());      // 80 - human1 출력
System.out.println("tm.higherEntry(82).getKey() + " - " + tm.higherEntry(82).getValue());    // 90 - human4 출력
System.out.println("tm.floorEntry(80).getKey() + " - " + tm.floorEntry(82).getValue());      // 80 - human1 출력
System.out.println("tm.ceilingEntry(80).getKey() + " - " + tm.ceilingEntry(80).getValue());  // 80 - human1 출력

NavigableMap<Integer, String> descMap = tm.descendingMap();
Set<Map.Entry<Integer, String>> descEntrySet = descMap.entrySet();
System.out.println(descEntrySet);                    // [99=human3, 90=human4, 80=human1, 70=human2] 출력
NavigableMap<Integer, String> ascMap = descMap.descendingMap();
Set<Map.Entry<Integer, String>> ascEntrySet = ascMap.entrySet();
System.out.println("ascEntrySet : " + ascEntrySet);  // [70=human2, 80=human1, 90=human4, 99=human3] 출력

NavigableMap<Integer, String> rangeMap = tm.subMap(80, true, 99, false);
System.out.println("rangeMap : " + rangeMap);        // {80=human1, 90=human4 출력
```

70 (human2),
80 (human1),
90 (human4),
99 (human3)

11.5. 비교연산자 (CompareTo 메서드)

- TreeSet 객체 및 TreeMap Key는 저장과 동시에 오름차순(문자의 경우 unicode순)으로 정렬됨.
- 사용자 정의로 정렬하려면 Comparable 인터페이스의 compareTo() 메서드 재정의 필요함.

TreeSet 활용

Person 클래스

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

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

  public int compareTo(Person p) {
    if (age<p.age) return -1;
    else if (age == p.age) return 0;
    else return 1;
  }
}
```

compareTo 메서드

compareTo 메서드는
1. 주어진 객체와 같으면, 0 리턴
2. 주어진 객체보다 작으면 음수 리턴
3. 주어진 객체보다 크면 양수 리턴.

위의 예에서는
-1, 0, 1을
리턴하게 함

결과

```
public static void main(String[] args) {
  TreeSet<Person> ts = new TreeSet<Person>();
  // 아래 내용 입력시 오름차순으로 입력됨.
  // 만약 compareTo의 return을 바꾸면 내림차순이 됨.
  ts.add(new Person("human1", 24));
  ts.add(new Person("human2", 25));
  ts.add(new Person("human3", 27));
  ts.add(new Person("human4", 21));

  Iterator<Person> iter = ts.iterator();

  while (iter.hasNext()) {
    Person p = iter.next();
    System.out.println ("Person : " + p.age + " - " + p.name);
  }

}
```

실행결과

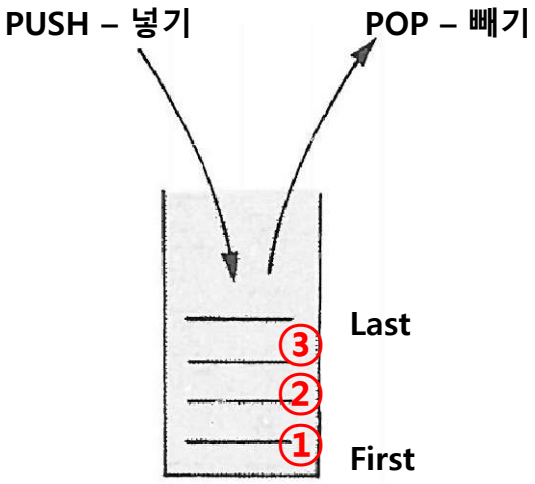
```
Person : 21 - human4
Person : 24 - human1
Person : 25 - human2
Person : 27 - human3.
```

11.6. LIFO 컬렉션

- LIFO(Last Input First Out – 후입선출) 은 나중에 넣은 것이 먼저 빠져나가는 구조임.
- LIFO는 Stack 클래스가 대표적임

LIFO 컬렉션 개요

LIFO 개요



메서드	설명
push (E item)	객체 추가
peek ( )	맨 위의 객체 가져옴. 삭제 안함
Pop ( )	맨 위의 객체 가져오면서 삭제함

LIFO(Stack) 실습 예제

Coin 클래스

```
public class Coin {
  private int value;
  public Coin (int value) {
    this.value = value;
  }
  public int getValue () {
    return value;
  }
}
```

실행 파일

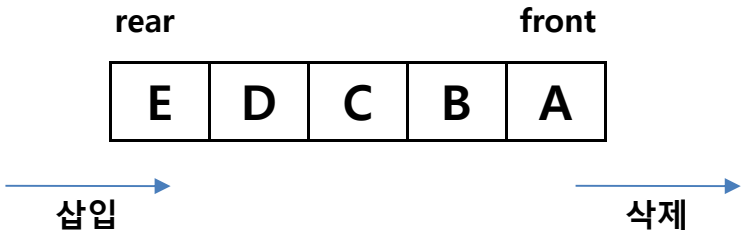
```
public static void main(String[] args) {
  Stack<Coin> coin = new Stack<Coin>();

  // Stack에 추가 (add도 가능함)
  coin.push(new Coin(100));  coin.push(new Coin(70));
  coin.push(new Coin(80));  coin.push(new Coin(90));
  coin.add(new Coin(10));

  while (!coin.isEmpty()) {
    Coin rmCoin = coin.pop();   // 맨 위의 객체 가져오면서 삭제.
    // 10, 90, 80, 70, 100 순으로 보임.
    System.out.println(rmCoin.getValue());
  }
}
```

11.7. FIFO 컬렉션

- FIFO(First Input First Out – 선입선출) 은 먼저 넣은 것이 먼저 빠져나가는 구조임.
- FIFO는 Queue 클래스가 대표적임

FIFO 컬렉션 개요

FIFO 개요



메서드	설명
offer (E item)	객체 추가
peek ( )	객체 가져옴. 삭제 안함
poll ( )	객체 가져오면서 삭제함

FIFO(Queue) 실습 예제

Coin 클래스

```
public class Coin {
  private int value;
  public Coin (int value) {
    this.value = value;
  }
  public int getValue () {
    return value;
  }
}
```

실행 파일

```
public static void main(String[] args) {
  Queue<Coin> coin = new LinkedList<Coin>( );

  // Queue에 추가 (add도 가능함)
  coin.offer(new Coin(100));  coin.offer(new Coin(70));
  coin.offer(new Coin(80));  coin.offer(new Coin(90));
  coin.add(new Coin(10));

  while (!coin.isEmpty()) {
    Coin rmCoin = coin.poll();    // 객체 가져오면서 삭제.
    // 100, 70, 80, 90, 10순으로 보임.
    System.out.println(rmCoin.getValue());
  }
}
```