컬렉션 프레임워크

컬렉션(Collection) 프레임워크(Framework)의 개념과 구조

컬렉션(Collection) 프레임워크(Framework)의 개념과 구조

☞ 컬렉션 ①

- 동일한 타입을 묶어서 관리하는 자료구조
- 저장 공간의 크기(capacity)를 동적으로 관리

☞ 프레임워크 ②

- 클래스와 인터페이스의 모임 (라이브러리)
- 클래스의 정의에 설계의 원칙 또는 구조가 존재

③ **컬렉션 프레임워크**

- 리스트, 스택, 큐, 트리 등의 **자료 구조**에 정렬, 탐색 등의 **알고리즘을 구조화** 해 놓은 프레임워크

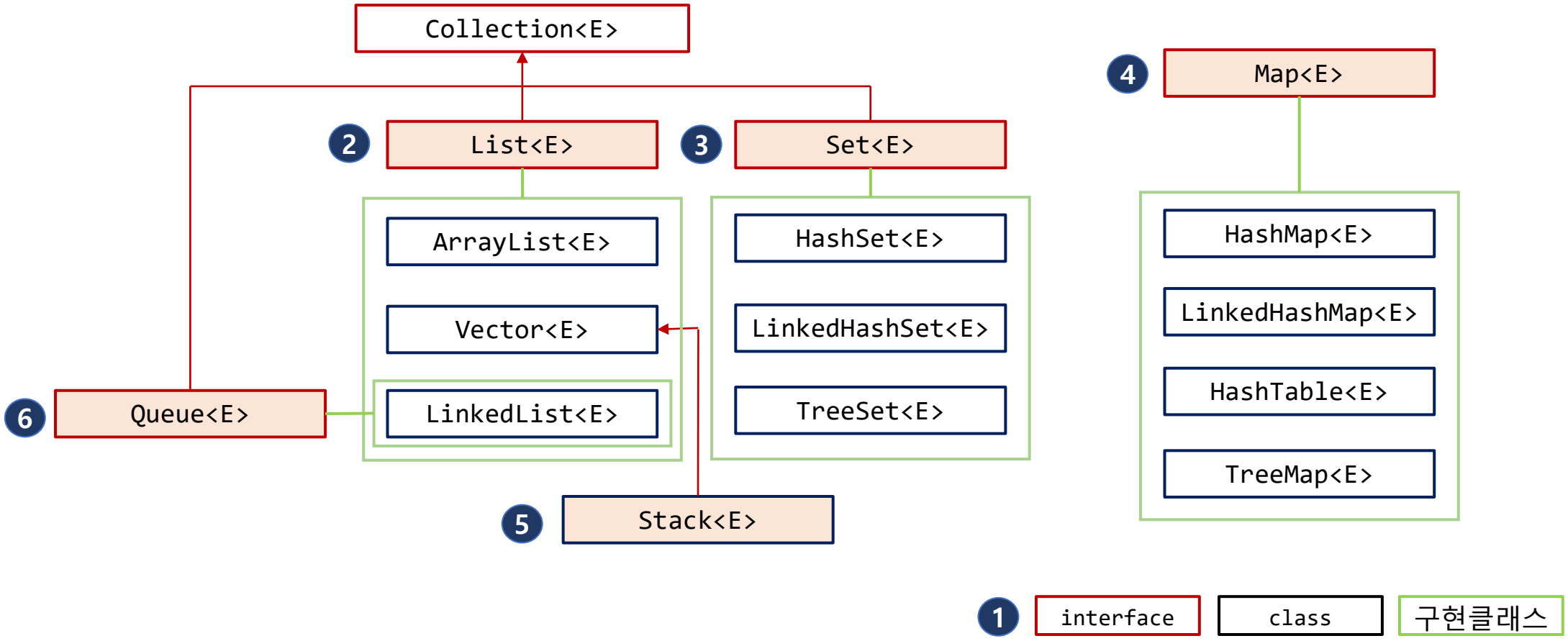
Q. 동일한 타입을 묶어서 관리하는 자료구조라면 배열과의 차이점은??

④ **배열의 2가지 특징**

#1. 동일한 타입만 묶어서 저장 가능
#2. 생성시 크기를 지정하여야 하면 추후 변경 불가 (**컬렉션과의 차이점**)

컬렉션(Collection) 프레임워크(Framework)의 개념과 구조

☞ 컬렉션 프레임워크를 이루는 주요클래스 및 인터페이스와 구현클래스



List<E> 컬렉션의 공통특성

List<E> 컬렉션의 공통특성

1 ☞ 배열 vs. 리스트 ⟶ 저장공간크기 동적변환

↓

저장공간크기 고정

- 배열(Array) 2

저장공간 =7

```
String[] array
    = new String[]{"가", "나", "다", "라", "마", "바", "사"};
```

3

0	1	2	3	4	5	6
가	나	다	라	마	바	사

```
array[2]=null;
array[5]=null;
System.out.println(array.length); //7
```

0	1	2	3	4	5	6
가	나	null	라	마	null	사

- 리스트(List) 4

저장공간 =0

저장공간 =7

```
List<String> aList = new ArrayList<>();
aList.add("가"); aList.add("나"); aList.add("다");
aList.add("라"); aList.add("마"); aList.add("바");
aList.add("사");
```

5

0	1	2	3	4	5	6
가	나	다	라	마	바	사

```
aList.remove("다");
aList.remove("바");
System.out.println(aList.size()); //5
```

저장공간 =5

0	1	2	3	4
가	나	라	마	사

List<E> 컬렉션의 공통특성

```
//#1. 배열
String[] array = new String[]{"가", "나", "다", "라", "마", "바", "사"};

array[2]=null;
array[5]=null;
System.out.println(array.length); //7
System.out.println(Arrays.toString(array)); //[가, 나, null, 라, 마, null, 사]
System.out.println();

//#2. 리스트
List<String> aList = new ArrayList<>();
System.out.println(aList.size()); //0
aList.add("가"); aList.add("나"); aList.add("다"); aList.add("라");
aList.add("마"); aList.add("바"); aList.add("사");

System.out.println(aList.size()); //7
aList.remove("다");
aList.remove("바");
System.out.println(aList.size()); //5
System.out.println(aList); //[가, 나, 라, 마, 사]
```

1

2

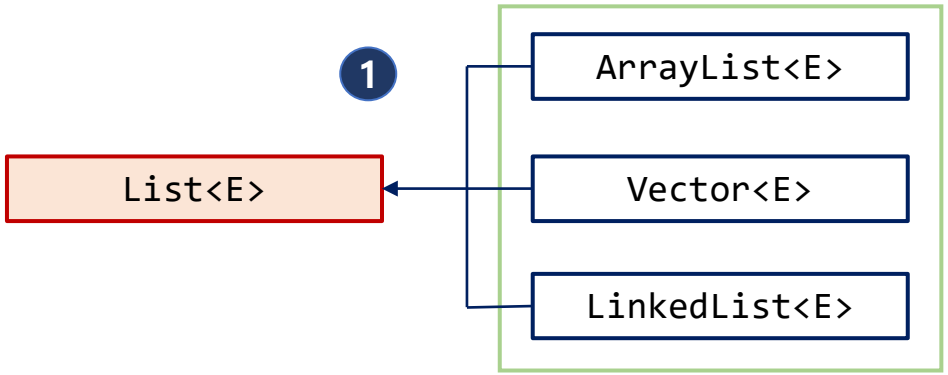
배열객체 한번에 출력하는 메서드

모든 컬렉션 객체(List, Set 등)는 자신의 데이터를 모두 출력하도록 toString() 메서드를 오버라이딩 해놓았음

```
7
[가, 나, null, 라, 마, null, 사]

0
7
5
[가, 나, 라, 마, 사]
```

List<E> 컬렉션의 공통특성

☞ 대표적인 List<E> 인터페이스 구현 클래스



☞ List<E> 컬렉션의 **객체 생성**

2

방법 #1 List<E> interface의 **구현 클래스 생성자**로 **동적**컬렉션 생성 ⟶ **(데이터의 추가 삭제 가능)**

방법 #2 **Arrays.asList(T ...)** 메서드를 이용하여 **정적**컬렉션 생성 ⟶ **(데이터의 추가 삭제 불가능)**

List<E> 컬렉션의 공통특성

☞ List<E> 컬렉션의 객체 생성

TIP (4)

- 기본생성자의 경우 원소 10개를 저장할 수 있는 저장공간(capacity) 확보
- 추후 원소가 많아지면 저장공간을 자동 추가
- 생성자 매개변수로 저장공간의 크기를 직접 넘겨줄 수 있음
  (**단, LinkedList<E>는 제외**)

(6) **주의. Capacity는 메모리 공간만을 의미하며 컬렉션의 크기(size())와는 무관**

(1) - **방법 #1**. List<E> interface의 **구현 클래스 생성자**로 **동적**컬렉션 생성 (데이터의 **추가 삭제 가능**)

(2)
```
List<제네릭타입지정> aList1 = new ArrayList<제네릭타입지정>();
List<제네릭타입지정> aList2 = new Vector<제네릭타입지정>();
List<제네릭타입지정> aList3 = new LinkedList<제네릭타입지정>();
```

(3)
```
ArrayList <제네릭타입지정> aList1 = new ArrayList<제네릭타입지정>();
Vector<제네릭타입지정> aList2 = new Vector<제네릭타입지정>();
LinkedList<제네릭타입지정> aList3 = new LinkedList<제네릭타입지정>();
```

(5) **예제**

```
List<Integer> aList1 = new ArrayList<Integer>(); →capacity=10
List<Integer> aList2 = new ArrayList<Integer>(30);→capacity=30
Vector<String> aList3 = new Vector<String>(); →capacity=10
//List<MyWork> aList4 = new LinkedList<MyWork>(20); →오류
```

List<E> 컬렉션의 공통특성

☞ List<E> 컬렉션의 객체 생성

1 - **방법 #2**. Arrays.asList(**T ...**) 메서드를 이용하여 정적컬렉션 생성
(데이터의 추가(add) 삭제(remove) 불가능, 데이터의 변경(set)은 가능)
(고정된 데이터를 저장하고 활용하고자 할 때 사용)

2
```
List<제네릭타입지정> aList1 = Arrays.asList(제네릭타입데이터들);
```

예제

3
```
List<Integer> aList1 = Arrays.asList(1,2,3,4);
aList1.set(1,7); // [1 7 3 4]
aList1.add(5); // 오류 UnsupportedOperationException
aList1.remove(0); // 오류 UnsupportedOperationException
```

List<E> 컬렉션의 공통특성

☞ List<E> 컬렉션의 객체 생성

1

```
//#방법1. 동적 크기를 가지는 리스트 객체 생성
List<Integer> aList1 = new ArrayList<Integer>(); //capacity(10)
List<Integer> aList2 = new ArrayList<Integer>(30); //capacity(30)

List<String> aList3 = new Vector<String>(); //capacity(10)
List<String> aList4 = new Vector<String>(20); //capacity(20)

List<String> aList5 = new LinkedList<String>(); //capacity(10)
//List<Double> aList6 = new LinkedList<Double>(20); //(불가능) capacity 지정 불가능
```

2

```
//#방법2. 정적 크기를 기지는 리스트 객체 생성
List<Integer> aList7 = Arrays.asList(1,2,3,4);
List<String> aList8 = Arrays.asList("안녕","방가");
aList7.set(1,7); // [1 7 3 4]
aList8.set(0, "감사"); //["감사", "방가"]

//aList7.add(5); // 예외 UnsupportedOperationException
//aList8.remove(0); // 예외 UnsupportedOperationException
```

List<E> 컬렉션의 공통특성

3

List<E>의 구현클래스는
아래의 모든 메서드를 포함

☞ List<E> 컬렉션의 특징

1 - 배열처럼 수집(collect)한 원소(Element)를 인덱스(index)로 관리

- List<E> 인터페이스의 **주요 메서드**

2

구분	리턴타입	메서드이름	기능
데이터 추가	boolean	add(E element)	매개변수로 입력된 원소를 리스트 마지막에 추가
	void	add(int index, E element)	index 위치에 입력된 원소 추가
	boolean	addAll(Collection<? Extends E> c)	매개변수로 입력된 컬렉션 전체를 마지막에 추가
	boolean	addAll(int index, Collection<? Extends E> c)	index 위치에 입력된 컬렉션 전체를 추가
데이터변경	E	set(int index, E element)	index 위치의 원소값을 입력된 원소로 변경
데이터삭제	E	remove(int index)	index 위치의 원소값 삭제
	boolean	remove(Object o)	원소 중 매개변수입력과 동일한 객체 삭제
	void	clear()	전체 원소 삭제

List<E> 컬렉션의 공통특성

☞ List<E> 컬렉션의 특징

- 배열처럼 수집(collect)한 원소(Element)를 인덱스(index)로 관리
- List<E> 인터페이스의 주요 메서드

List<E>의 구현클래스는 아래의 모든 메서드를 포함

구분	리턴타입	메서드이름	기능
리스트 데이터 정보추출	E	get(int index)	Index 위치의 원소값을 꺼내어 리턴
	int	size()	리스트 객체 내에 포함된 원소의 개수
	boolean	isEmpty()	리스트의 원소가 하나도 없는지 여부를 리턴
리스트 배열 변환	Object[]	toArray()	리스트를 Object 배열로 변환
	T[]	toArray(T[] t)	입력매개변수로 전달한 타입의 배열로 변환

1

2

```
list의 size() ≥ 배열의 length ➔ list의 size 크기를 가지는 배열 생성
list의 size() < 배열의 length ➔ 배열 length 크기를 가지는 배열 생성
```

The End

List<E> 컬렉션 – ArrayList<E>

List<E> 컬렉션 – ArrayList<E>

☞ ArrayList<E>

디폴트값은 10이며 원소가 10을 넘는 경우 자동으로 저장공간 확대

1
- List<E> 인터페이스를 구현한 **구현 클래스**
- 배열처럼 수집(collect)한 원소(Element)를 인덱스(index)로 관리하며 저장용량(capacity)을 동적관리

- **데이터 추가**

2
3

```
List<Integer> aList1 = new ArrayList<Integer>();

//#1. add(E element)
aList1.add(3);
aList1.add(4);
aList1.add(5);
System.out.println(aList1.toString()); //[3, 4, 5]

//#2. add(int index, E element)
aList1.add(1, 6);
System.out.println(aList1.toString()); //[3, 6, 4, 5]
```

모든 컬렉션 객체(List, Set 등)는 자신의 데이터를 모두 출력하도록 toString() 메서드를 오버라이딩 해놓았음

```
[3, 4, 5]

[3, 6, 4, 5]
```

List<E> 컬렉션 – ArrayList<E>

☞ ArrayList<E>

- **데이터 추가 (계속)**

1
2

```
//#3. addAll(Collection<? extends E> c)
List<Integer> aList2 = new ArrayList<Integer>();
aList2.add(1);
aList2.add(2);
aList2.addAll(aList1);
System.out.println(aList2.toString()); //[1,2,3,6,4,5]

//#4. addAll(int index, Collection<? extends E> c)
List<Integer> aList3 = new ArrayList<Integer>();
aList3.add(1);
aList3.add(2);
aList3.addAll(1,aList3);
System.out.println(aList3.toString()); //[1,1,2,2]
```

```
[1, 2, 3, 6, 4, 5]

[1, 1, 2, 2]
```

List<E> 컬렉션 – ArrayList<E>

☞ ArrayList<E>

- **데이터 변경**

1

```
//#5. set(int index, E element)
aList3.set(1, 5);
aList3.set(3, 6);
//aList3.set(4, 7); //IndexOutOfBoundsException
System.out.println(aList3.toString()); //[1,5,2,6]
```

```
[1, 5, 2, 6]
```

- **데이터 삭제**

2

```
//#6. remove(int index)
aList3.remove(1);
System.out.println(aList3.toString()); //[1,2,6]
```

3

```
//#7. remove(Object o)
aList3.remove(new Integer(2));
System.out.println(aList3.toString()); //[1,6]
```

4

```
//#8. clear()
aList3.clear();
System.out.println(aList3.toString()); //[]
```

```
[1, 2, 6]

[1, 6]

[]
```

List<E> 컬렉션 – ArrayList<E>

☞ ArrayList<E>

- **데이터 정보 추출**

1
```
//#9. isEmpty();
System.out.println(aList3.isEmpty()); //true
```

2
```
//#10. size()
aList3.add(1);
aList3.add(2);
aList3.add(3);
System.out.println(aList3.toString()); //[1,2,3]
System.out.println("size : "+aList3.size()); //size : 3
```

3
```
//#11. get(int index)
System.out.println("0번째 : " + aList3.get(0)); //0번째 : 1
System.out.println("1번째 : " + aList3.get(1)); //1번째 : 2
System.out.println("2번째 : " + aList3.get(2)); //2번째 : 3
for(int i=0; i<aList3.size(); i++) {
  System.out.println(i+"번째 : " + aList3.get(i));
}
```

```
true

[1, 2, 3]
Size : 3

0번째 : 1
1번째 : 2
2번재 : 3

0번째 : 1
1번째 : 2
2번재 : 3
```

List<E> 컬렉션 – ArrayList<E>

☞ ArrayList<E>

- **리스트→ 배열**

```
//#12. toArray()
Object[] object = aList3.toArray();
System.out.println(Arrays.toString(object)); //[1,2,3]

//#13-1. toArray(T[] t)
Integer[] integer1 = aList3.toArray(new Integer[0]);
System.out.println(Arrays.toString(integer1)); //[1,2,3]

//#13-2. toArray(T[] t)
Integer[] integer2 = aList3.toArray(new Integer[5]);
System.out.println(Arrays.toString(integer2)); //[1,2,3,null,null]
```

1 → `[1, 2, 3]`

2 → `[1, 2, 3]`

3 → `[1, 2, 3, null, null]`

4 list의 size() ≥ 배열의 length ➔ list의 size 크기를 가지는 배열 생성
list의 size() < 배열의 length ➔ 배열 length 크기를 가지는 배열 생성

List<E> 컬렉션 – Vector<E>

List<E> 컬렉션 – Vector<E>

☞ Vector<E>

- ArrayList<E>와의 **공통점**:

1
  - 동일한 타입의 객체 수집(collection)
  - 메모리의 동적할당
  - 데이터의 추가, 변경, 삭제 등의 메서드

- ArrayList<E>와의 **다른점**:

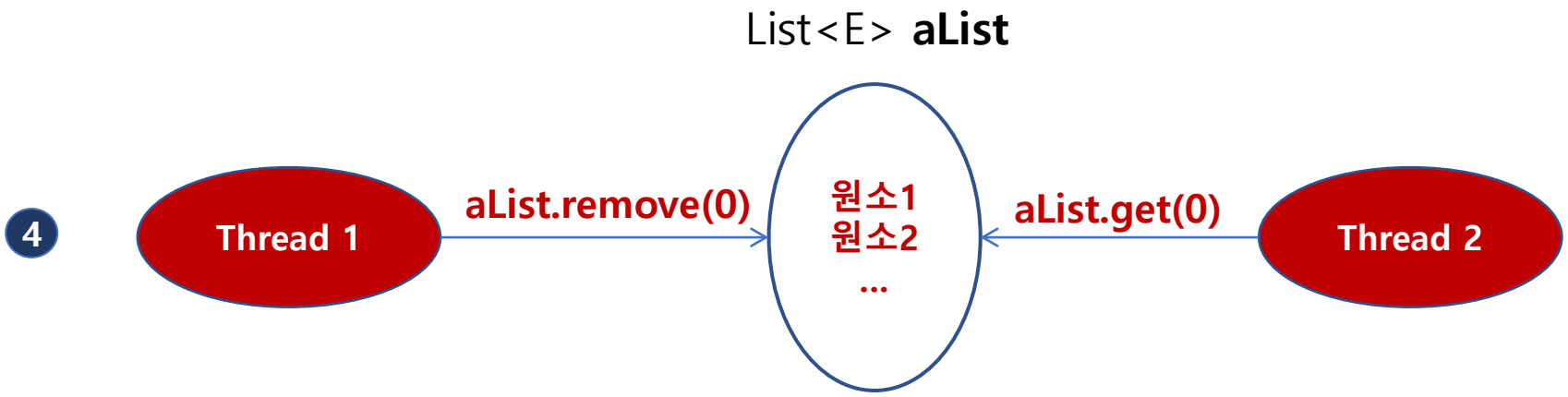
2
  - 모든 메서드가 **동기화(synchronized) 메서드로 구현**되어 **멀티쓰레드에 적합**(Thread Safe)

3

```
public synchronized E remove(int index) {
    modCount++;
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    E oldValue = elementData(index);
```

```
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return elementData(index);
}
```

4



List<E> 컬렉션 – Vector<E>

☞ Vector<E>

- **데이터 추가**

```
List<Integer> vector1 = new Vector<Integer>();

//#1. add(E element)
vector1.add(3);
vector1.add(4);
vector1.add(5);
System.out.println(vector1.toString()); //[3, 4, 5]

//#2. add(int index, E element)
vector1.add(1, 6);
System.out.println(vector1.toString()); //[3, 6, 4, 5]
```

1

2

```
[3, 4, 5]

[3, 6, 4, 5]
```

List<E> 컬렉션 – Vector<E>

☞ Vector<E>

- **데이터 추가 (계속)**

1

```
//#3. addAll(Collection<? extends E> c)
List<Integer> vector2 = new Vector<Integer>();
vector2.add(1);
vector2.add(2);
vector2.addAll(vector1);
System.out.println(vector2.toString()); //[1,2,3,6,4,5]
```

2

```
//#4. addAll(int index, Collection<? extends E> c)
List<Integer> vector3 = new Vector<Integer>();
vector3.add(1);
vector3.add(2);
vector3.addAll(1, vector3);
System.out.println(vector3.toString()); //[1,1,2,2]
```

```
[1, 2, 3, 6, 4, 5]
```

```
[1, 1, 2, 2]
```

List<E> 컬렉션 – Vector<E>

☞ Vector<E>

- **데이터 변경**

1

```
//#5. set(int index, E element)
vector3.set(1, 5);
vector3.set(3, 6);
//vector3.set(4, 7); //IndexOutOfBoundsException
System.out.println(vector3.toString()); //[1,5,2,6]
```

```
[1, 5, 2, 6]
```

- **데이터 삭제**

2

```
//#6. remove(int index)
vector3.remove(1);
System.out.println(vector3.toString()); //[1,2,6]
```

```
[1, 2, 6]
```

3

```
//#7. remove(Object o)
vector3.remove(new Integer(2));
System.out.println(vector3.toString()); //[1,6]
```

```
[1, 6]
```

4

```
//#8. clear()
vector3.clear();
System.out.println(vector3.toString()); //[]
```

```
[]
```

List<E> 컬렉션 – Vector<E>

☞ Vector<E>

- **데이터 정보 추출**

```
//#9. isEmpty();
System.out.println(vector3.isEmpty()); //true

//#10. size()
vector3.add(1);
vector3.add(2);
vector3.add(3);
System.out.println(vector3.toString()); //[1,2,3]
System.out.println("size : "+ vector3.size()); //size : 3

//#11. get(int index)
System.out.println("0번째 : " + vector3.get(0)); //0번째 : 1
System.out.println("1번째 : " + vector3.get(1)); //1번째 : 2
System.out.println("2번째 : " + vector3.get(2)); //2번째 : 3
for(int i=0; i< vector3.size(); i++) {
  System.out.println(i+"번째 : " + vector3.get(i));
}
```

```
true

[1, 2, 3]
Size : 3

0번째 : 1
1번째 : 2
2번재 : 3

0번째 : 1
1번째 : 2
2번재 : 3
```

List<E> 컬렉션 – Vector<E>

☞ Vector<E>

- **리스트→ 배열**

1
```
//#12. toArray()
Object[] object = vector3.toArray();
System.out.println(Arrays.toString(object)); //[1,2,3]
```

2
```
//#13-1. toArray(T[] t)
Integer[] integer = vector3.toArray(new Integer[0]);
System.out.println(Arrays.toString(integer)); //[1,2,3]
```

3
```
//#13-2. toArray(T[] t)
Integer[] integer = vector3.toArray(new Integer[5]);
System.out.println(Arrays.toString(integer)); //[1,2,3,null,null]
```

```
[1, 2, 3]

[1, 2, 3]

[1, 2, 3, null, null]
```

list의 size() ≥ 배열의 length ➔ list의 size 크기를 가지는 배열 생성
list의 size() < 배열의 length ➔ 배열 length 크기를 가지는 배열 생성

The End

List<E> 컬렉션 – LinkedList<E>

List<E> 컬렉션 – LinkedList<E>

☞ LinkedList<E>

- ArrayList<E>와의 **공통점**:
  - 동일한 타입의 객체 수집(collection)
  - 메모리의 동적할당
  - 데이터의 추가, 변경, 삭제 등의 메서드

1

- ArrayList<E>와의 **다른점**:
  - 디폴트 저장공간(10)만 사용하며 생성자로 저장공간의 크기 지정 불가
  - 데이터의 내부 저장방식이 index가 아닌 앞뒤 객체의 **위치** 정보를 저장

2

3

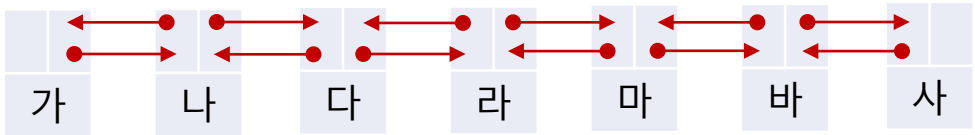
```
List<E> aLinkedList1 = new LinkedList<Integer>();   //O
List<E> aLinkedList1 = new LinkedList<Integer>(20); //X
```

4

ArrayList

0	1	2	3	4	5	6
가	나	다	라	마	바	사

LinkedList



List<E> 컬렉션 – LinkedList<E>

구분	ArrayList<E>	LinkedList<E>
추가,삭제(add, remove)	속도 **늦음**	속도 **빠름**
검색(get)	속도 **빠름**	속도 **늦음**

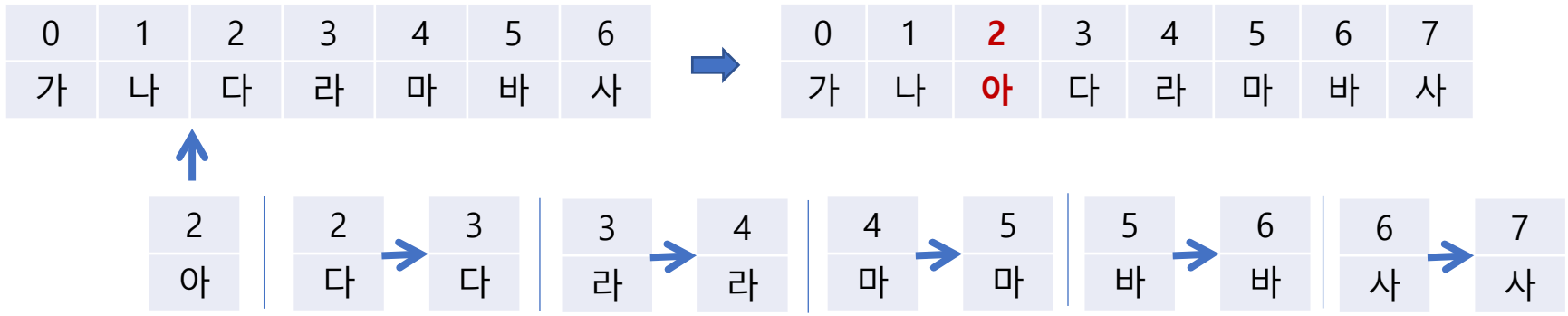
3

☞ LinkedList<E>

- ArrayList<E> vs. LinkedList<E>의 **데이터 추가** 메커니즘

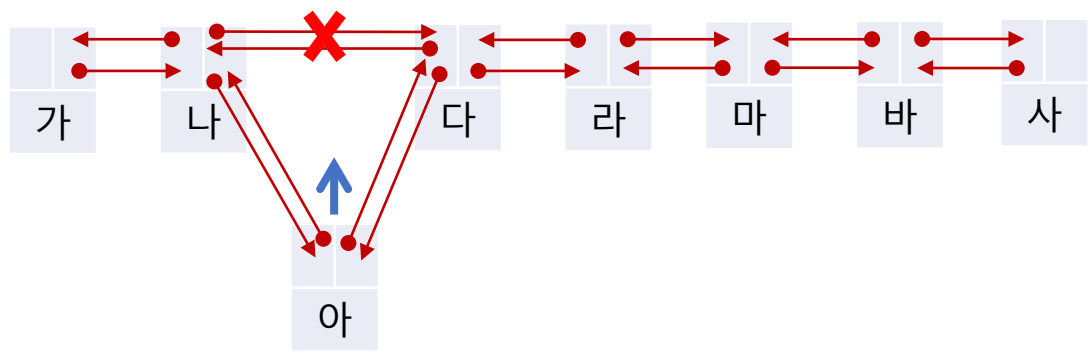
1 **ArrayList**

```
aList.add(2, '아')
```



2 **LinkedList**

```
aLinkedList.add(2, '아')
```



List<E> 컬렉션 – LinkedList<E>

☞ LinkedList<E>

- **데이터 추가**

1
```
List<Integer> linkedList1 = new LinkedList<Integer>();

//#1. add(E element)
linkedList1.add(3);
linkedList1.add(4);
linkedList1.add(5);
System.out.println(linkedList1.toString()); //[3, 4, 5]
```

2
```
//#2. add(int index, E element)
linkedList1.add(1, 6);
System.out.println(linkedList1.toString()); //[3, 6, 4, 5]
```

```
[3, 4, 5]

[3, 6, 4, 5]
```

List<E> 컬렉션 – LinkedList<E>

☞ LinkedList<E>

- **데이터 추가 (계속)**

1

```
//#3. addAll(Collection<? extends E> c)
List<Integer> linkedList2 = new LinkedList<Integer>();
linkedList2.add(1);
linkedList2.add(2);
linkedList2.addAll(linkedList1);
System.out.println(linkedList2.toString()); //[1,2,3,6,4,5]
```

2

```
//#4. addAll(int index, Collection<? extends E> c)
List<Integer> linkedList3 = new LinkedList<Integer>();
linkedList3.add(1);
linkedList3.add(2);
linkedList3.addAll(1, linkedList3);
System.out.println(linkedList3.toString()); //[1,1,2,2]
```

```
[1, 2, 3, 6, 4, 5]
```

```
[1, 1, 2, 2]
```

List<E> 컬렉션 – LinkedList<E>

☞ LinkedList<E>

- **데이터 변경**

1

```
//#5. set(int index, E element)
linkedList3.set(1, 5);
linkedList3.set(3, 6);
// linkedList3.set(4, 7); //IndexOutOfBoundsException
System.out.println(linkedList3.toString()); //[1,5,2,6]
```

```
[1, 5, 2, 6]
```

- **데이터 삭제**

2

```
//#6. remove(int index)
linkedList3.remove(1);
System.out.println(linkedList3.toString()); //[1,2,6]
```

```
[1, 2, 6]
```

3

```
//#7. remove(Object o)
linkedList3.remove(new Integer(2));
System.out.println(linkedList3.toString()); //[1,6]
```

```
[1, 6]
```

4

```
//#8. clear()
linkedList3.clear();
System.out.println(linkedList3.toString()); //[]
```

```
[]
```

List<E> 컬렉션 – LinkedList<E>

☞ LinkedList<E>

- **데이터 정보 추출**

1
```
//#9. isEmpty();
System.out.println(linkedList3.isEmpty()); //true
```

2
```
//#10. size()
linkedList3.add(1);
linkedList3.add(2);
linkedList3.add(3);
System.out.println(linkedList3.toString()); //[1,2,3]
System.out.println("size : "+ linkedList3.size()); //size : 3
```

3
```
//#11. get(int index)
System.out.println("0번째 : " + linkedList3.get(0)); //0번째 : 1
System.out.println("1번째 : " + linkedList3.get(1)); //1번째 : 2
System.out.println("2번째 : " + linkedList3.get(2)); //2번째 : 3
for(int i=0; i< linkedList3.size(); i++) {
  System.out.println(i+"번째 : " + linkedList3.get(i));
}
```

```
True

[1, 2, 3]
Size : 3

0번째 : 1
1번째 : 2
2번재 : 3

0번째 : 1
1번째 : 2
2번재 : 3
```

List<E> 컬렉션 – LinkedList<E>

☞ LinkedList<E>

- **리스트→ 배열**

```
//#12. toArray()
Object[] object = linkedList3.toArray();
System.out.println(Arrays.toString(object)); //[1,2,3]

//#13-1. toArray(T[] t)
Integer[] integer = linkedList3.toArray(new Integer[0]);
System.out.println(Arrays.toString(integer)); //[1,2,3]

//#13-2. toArray(T[] t)
Integer[] integer = linkedList3.toArray(new Integer[5]);
System.out.println(Arrays.toString(integer)); //[1,2,3]
```

1 → `[1, 2, 3]`

2 → `[1, 2, 3]`

3 → `[1, 2, 3, null, null]`

4 list의 size() ≥ 배열의 length ➔ list의 size 크기를 가지는 배열 생성
list의 size() < 배열의 length ➔ 배열 length 크기를 가지는 배열 생성

List<E> 컬렉션 –
ArrayList<E> vs. LinkedList<E>

List<E> 컬렉션 – ArrayList<E> vs. LinkedList<E> 성능비교

☞ ArrayList<E> vs. LinkedList<E> 성능비교

- **데이터의 추가(add) 시간 비교**

1

```
//#. ArrayList 및 LinkedList 객체 선언
List<Integer> aList = new ArrayList<>();
List<Integer> linkedList = new LinkedList<>();
long startTime=0, endTime=0;
```

3

```
//#1. 데이터 추가 시간 측정비교

//@1-1 ArrayList 데이터 추가시간 측정
startTime = System.nanoTime();
for(int i=0; i<100000; i++) {
  aList.add(0, i);
}
endTime = System.nanoTime();
System.out.println("ArrayList 데이터 추가시간: " + (endTime-startTime) + "ns");
```

aList.add(i)의 경우 차이가 크지 않음
(뒤에 추가되어 데이터의 shift가 일어나지 않음)

4

```
//@1-2 LinkedList 데이터 추가시간 측정
startTime = System.nanoTime();
for(int i=0; i<100000; i++) {
  linkedList.add(0, i);
}
endTime = System.nanoTime();
System.out.println("LinkedList 데이터 추가시간: " + (endTime-startTime) + "ns");
```

5

```
ArrayList 데이터 추가시간: 1009859700ns
LinkedList 데이터 추가시간: 4064800ns
```

약 250배 차이

2

```
System.currentTimeMillis() : 1970.01.01 00:00과 현재시간과의 차이를 ms(밀리초, 1/1000초) 단위로 리턴(long형)
System.nanoTime() : 단순히 시간차이를 구하기 위한 목적으로 사용 (수치의 의미는 없음)
```

List<E> 컬렉션 – ArrayList<E> vs. LinkedList<E> 성능비교

☞ ArrayList<E> vs. LinkedList<E> 성능비교

- **데이터의 검색(get) 시간 비교**

```
//#2. 데이터의 검색(get) 시간 비교

//@2-1. ArrayList 데이터 검색 시간 측정
startTime = System.nanoTime();
for(int i=0; i<aList.size(); i++) {
  aList.get(i);
}
endTime = System.nanoTime();
System.out.println("ArrayList 데이터 검색시간: " + (endTime-startTime) + "ns");

//@2-2. LinkedList 데이터 검색 시간 측정
startTime = System.nanoTime();
for(int i=0; i<aLinkedList.size(); i++) {
  aLinkedList.get(i);
}
endTime = System.nanoTime();
System.out.println("LinkedList 데이터 검색시간: " + (endTime-startTime) + "ns");
```

1

2

3 약 4400배 차이

```
ArrayList 데이터 검색시간: 1207600ns
LinkedList 데이터 검색시간: 5332310400ns
```

List<E> 컬렉션 – ArrayList<E> vs. LinkedList<E> 성능비교

☞ ArrayList<E> vs. LinkedList<E> 성능비교

- **데이터의 제거(remove) 시간 비교**

```
//#3. 데이터의 제거 (remove) 시간 비교

//@3-1. ArrayList 데이터 제거 시간 측정
startTime = System.nanoTime();
for(int i=0; i<aList.size(); i++) {
  aList.remove(0);
}
endTime = System.nanoTime();
System.out.println("ArrayList 데이터 제거시간: " + (endTime-startTime) + "ns");

//@3-2. LinkedList 데이터 제거 시간 측정
startTime = System.nanoTime();
for(int i=0; i<aLinkedList.size(); i++) {
  aLinkedList.remove(0);
}
endTime = System.nanoTime();
System.out.println("LinkedList 데이터 제거시간: " + (endTime-startTime) + "ns");
```

1

2

3 약 500배 차이

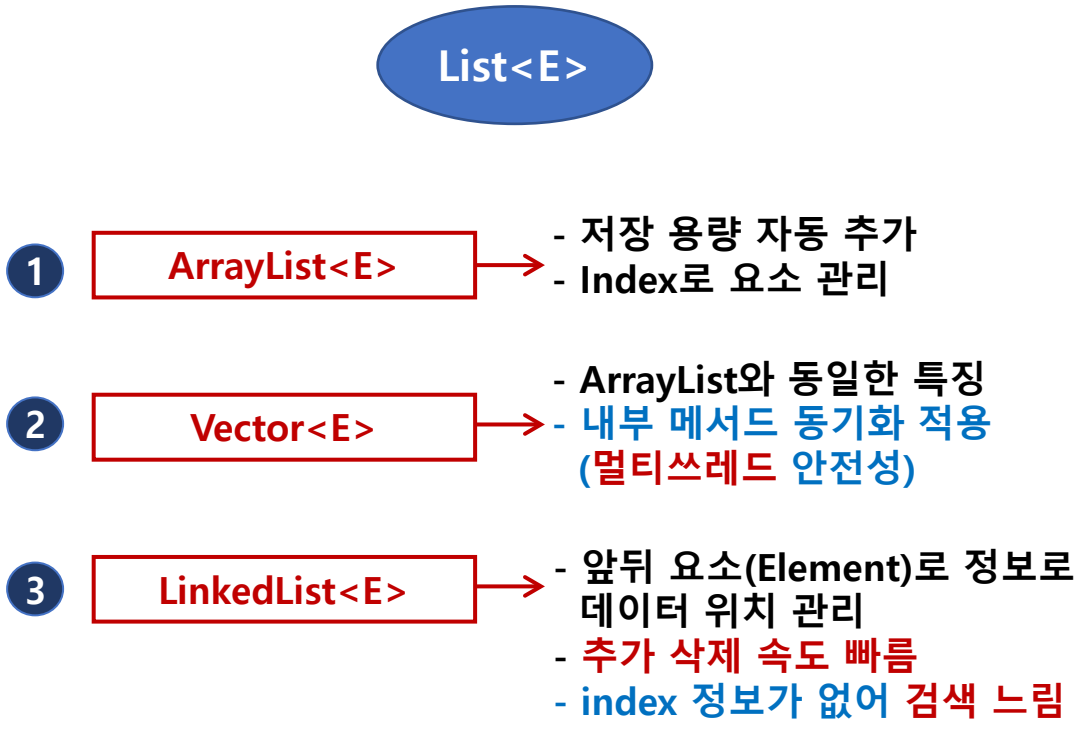
```
ArrayList 데이터 제거시간: 706482900ns
LinkedList 데이터 제거시간: 1409700ns
```

List<E> 컬렉션 – Summary

List<E> 컬렉션

```
[다, 마, 나, 가]
[다, 마, 나, 가]
[다, 마, 나, 가]
```

☞ List<E>의 구현객체 정리




```java
//#1. ArrayList
List<String> arrayList = new ArrayList<String>();
arrayList.add("다");
arrayList.add("마");
arrayList.add("나");
arrayList.add("가");
System.out.println(arrayList.toString()); //[다, 마, 나, 가]

//#2. Vector
List<String> vector = new Vector<String>();
vector.add("다");
vector.add("마");
vector.add("나");
vector.add("가");
System.out.println(vector.toString()); //[다, 마, 나, 가]

//#3. LinkedList
List<String> linkedList = new LinkedList<String>();
linkedList.add("다");
linkedList.add("마");
linkedList.add("나");
linkedList.add("가");
System.out.println(linkedList.toString()); //[다, 마, 나, 가]
```

4

5

6

The End

Set<E> 컬렉션의 공통특성

Set<E> 컬렉션

☞ Set<E> 컬렉션의 특징

1 - 집합의 개념으로 인덱스 정보를 포함하고 있지 않음

2 - **중복저장 불가** → 인덱스 정보가 없기 때문에 중복된 원소 중 특정 위치 값을 꺼낼 방법이 없음
(심지어 null 값도 한 개만 포함가능)

3

*** 중복여부를 확인하기 위해선**
같음을 비교할 수 있어야 함

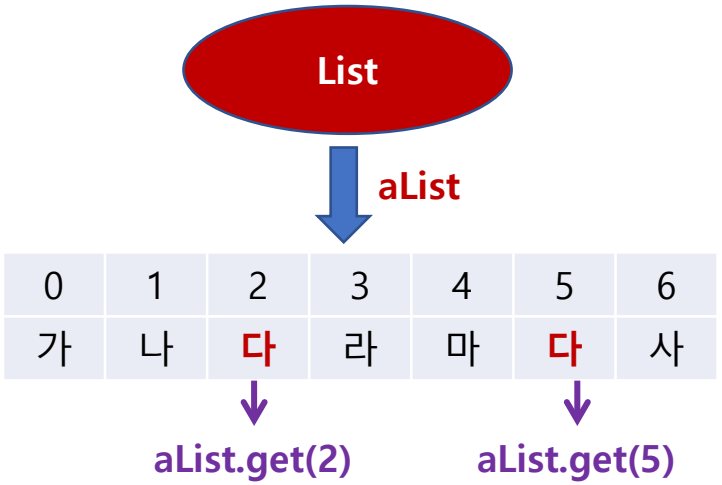
- "안녕" vs. "안녕"
- 123 vs. 123
- A a1 = new A(3) vs. A a2 = new A(3)

Set<E> 컬렉션

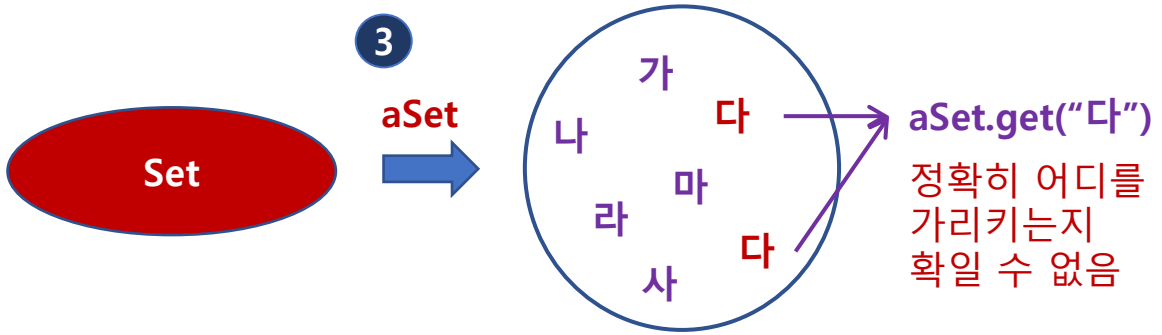
☞ Set<E> 컬렉션의 특징

1. 중복저장 **가능/불가능을 판단**하는 기준 : 중복된 특정 원소 중 하나를 **특정하여 꺼낼 수 있는지 여부**

2.

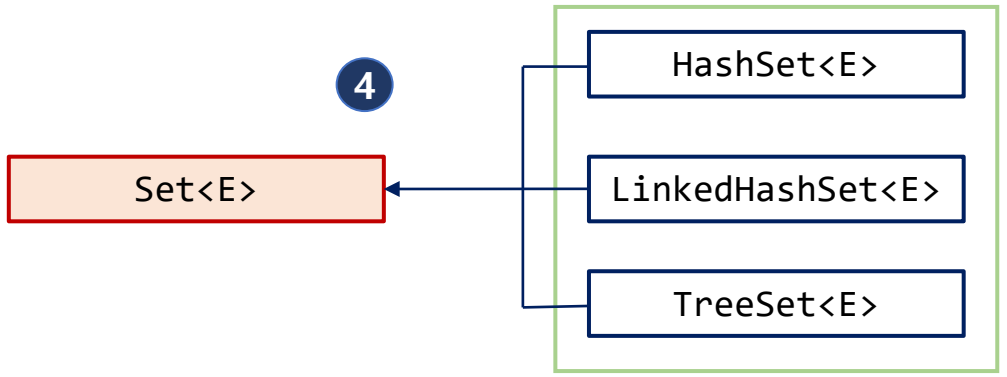


3.



☞ 대표적인 Set<E> 인터페이스 구현 클래스

4.



Set<E> 컬렉션

☞ Set<E> 컬렉션의 주요 메서드

3 **Set<E>의 구현클래스는 아래의 모든 메서드를 포함**

1 **List<E>와 비교하여 index가 포함된 메서드가 없음**

2

구분	리턴타입	메서드이름	기능
데이터 추가	boolean	add(E element)	매개변수로 입력된 원소를 리스트에 추가
	boolean	addAll(Collection<? Extends E> c)	매개변수로 입력된 컬렉션 전체를 추가
데이터 삭제	boolean	remove(Object o)	원소 중 매개변수입력과 동일한 객체 삭제
	void	clear()	전체 원소 삭제
데이터 정보추출	boolean	isEmpty()	Set 객체가 비워져있는지 여부를 리턴
	boolean	contains(Object o)	매개변수로 입력된 원소가 있는지 여부를 리턴
	int	size()	리스트 객체 내에 포함된 원소의 개수
	Iterator<E>	iterrator()	Set 객체내의 데이터를 연속하여 꺼내는 Iterator 객체 리턴
Set객체 배열 변환	Object[]	toArray()	리스트를 Object 배열로 변환
	T[]	toArray(T[] t)	입력매개변수로 전달한 타입의 배열로 변환

Set<E> 컬렉션 – HashSet<E>

Set<E> 컬렉션 – HashSet<E>

☞ HashSet<E>

디폴트값은 16이며 원소가 16을 넘는 경우 자동으로 저장공간 확대

1
- Set<E> 인터페이스를 구현한 구현 클래스
- 수집(collect)한 원소(Element)를 집합의 형태로 관리하며 저장용량 (capacity)을 동적관리
- 입력의 순서와 출력의 순서는 동일하지 않을 수 있음

- **데이터 추가**

2
```
Set<String> hSet1 = new HashSet<String>();
//#1. add(E element)
hSet1.add("가");
hSet1.add("나");
hSet1.add("가");
System.out.println(hSet1.toString()); //[가, 나]

//#2. addAll(Collection<? extends E> c)
Set<String> hSet2 = new HashSet<String>();
hSet2.add("나");
hSet2.add("다");
hSet2.addAll(hSet1);
System.out.println(hSet2.toString()); //[가, 다, 나]
```

3

[가, 나]

입력 순서와 다름

[가, 다, 나]

Set<E> 컬렉션 – HashSet<E>

☞ HashSet<E>

- **데이터 삭제**

```
//#3. remove(Object o)
hSet2.remove("나");
System.out.println(hSet2.toString()); //[가, 다]

//#4. clear()
hSet2.clear();
System.out.println(hSet2.toString()); //[]
```

```
[가, 다]

[]
```

Set<E> 컬렉션 – HashSet<E>

☞ HashSet<E>

- **데이터 정보 추출**

```
//#5. isEmpty()
System.out.println(hSet2.isEmpty()); //true

//#6. contains(Object o)
Set<String> hSet3 = new HashSet<String>();
hSet3.add("가");
hSet3.add("다");
hSet3.add("나");
System.out.println(hSet3.contains("나")); //true
System.out.println(hSet3.contains("라")); //false

//#7. size()
System.out.println(hSet3.size()); //3

//#8. iterator()
Iterator<String> iterator = hSet3.iterator();
while(iterator.hasNext()) {
  System.out.print(iterator.next() + " "); //가 다 나
}
System.out.println();
```

```
//#8. for-each
for(Sting s : hSet3)
  System.out.print(s + " "); //가 다 나
}
System.out.println();
```

```
true

true
false

3

가 다 나
```

Set<E> 컬렉션 – HashSet<E>

☞ HashSet<E>

- **Set → 배열**

```
//#9. toArray()
Object[] objArray = hSet3.toArray();
System.out.println(Arrays.toString(objArray)); //[가 다 나]

//#10-1. toArray(T[] t)
String[] strArray1 = hSet3.toArray(new String[0]);
System.out.println(Arrays.toString(strArray1)); //[가 다 나]

//#10-2. toArray(T[] t)
String[] strArray2 = hSet3.toArray(new String[5]);
System.out.println(Arrays.toString(strArray2)); //[가 다 나 null null]
```

1 `[가 다 나]`

2 `[가 다 나]`

3 `[가 다 나 null null]`

Set<E> 컬렉션 – HashSet<E>

1

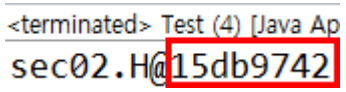
중복확인 메커니즘 이해를 위한 사전 지식

(hashCode()의 개념 + 등가연산(==)과 equals() 메서드의 차이점

hashCode(), equals() → Object 클래스의 메서드 → 모든 클래스 내에 포함

2

```
package sec02;

class H{ }
public class Test {
  public static void main(String[] ar) {
    H h = new H();
    System.out.println(h);
  }
}
```



패키지명.클래스명@해쉬코드

3

```
class A{
  int data;
  public A(int data) {
    this.data = data;
  }
}
```

```
A a1 = new A(3);
A a2 = new A(3);
System.out.println(a1==a2); //false
System.out.println(a1.equals(a2)); //false
```

4

```
Integer a1 = new Integer(3);
Integer a2 = new Integer(3);

System.out.println(a1==a2); //false
System.out.println(a1.equals(a2)); //true
```

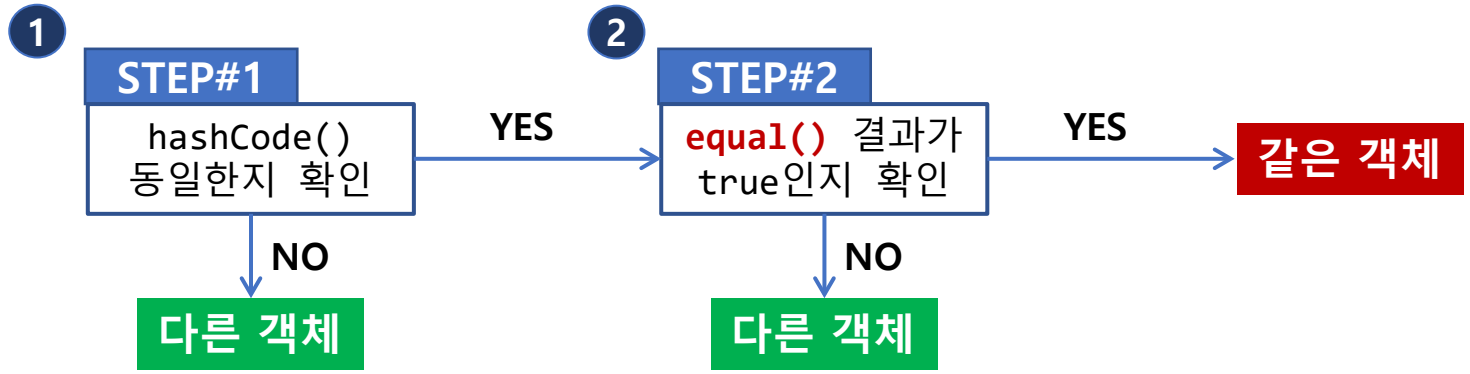
```
String s1 = new String("안녕");
String s2 = new String("안녕");

System.out.println(s1==s2); //false
System.out.println(s1.equals(s2)); //true
```

해쉬코드는 객체를 기반으로 생성된 고유값 (실제 번지와는 다름)

Object의 equal()은 == 와 동일한 연산 (저장 번지 비교)

Set<E> 컬렉션 – HashSet<E>

HashSet<E>에서의 중복확인 메커니즘



3

동일객체의 정의를 위해서는 hashCode()와 equals() 메서드 Overriding 필요

CASE 1

4

두 메서드 모두
Overriding을 하지 않은 경우

5

```
class A{
  int data;
  public A(int data) {
    this.data = data;
  }
}
```

6

```
//#1. CASE1. equals(): 오버라이딩 X + hashcode(): 오버라이딩 X
Set<A> hashSet1 = new HashSet<>();
A a1 = new A(3);
A a2 = new A(3);
System.out.println(a1==a2); //false
System.out.println(a1.equals(a2)); //false
System.out.println(a1.hashCode() + " " + a2.hashCode());

hashSet1.add(a1);
hashSet1.add(a2);
System.out.println(hashSet1.size()); //2 (다른 객체)
```

```
false
false
366712642 1829164700
2
```

Set<E> 컬렉션 – HashSet<E>

HashSet<E>에서의 중복확인 메커니즘

CASE 2

1

equals() 메서드만 Overriding 한 경우

2

```
class B{
  int data;
  public B(int data) {
    this.data = data;
  }
  @Override
  public boolean equals(Object obj) {
    if(obj instanceof B) {
      if(this.data == ((B)obj).data)
        return true;
    }
    return false;
  }
}
```

3

```
//#2. CASE2. equals(): 오버라이딩 O + hashcode(): 오버라이딩 X
Set<B> hashSet2 = new HashSet<>();

B b1 = new B(3);
B b2 = new B(3);
System.out.println(b1==b2); //false
System.out.println(b1.equals(b2)); //true
System.out.println(b1.hashCode() + " " + b2.hashCode());

hashSet2.add(b1);
hashSet2.add(b2);
System.out.println(hashSet2.size()); //2 (다른 객체)
```

```
false
true
2018699554 1311053135
2
```

Set<E> 컬렉션 – HashSet<E>

HashSet<E>에서의 중복확인 메커니즘

CASE 3

```
class C{
  int data;
  public C(int data) {
    this.data = data;
  }
  @Override                    ②
  public boolean equals(Object obj) {
    if(obj instanceof C) {
      if(this.data == ((C)obj).data)
        return true;
    }
    return false;
  }
  @Override                    ③
  public int hashCode() {
    return Objects.hash(data); //data
  }
}
```

①

equals(), hashCode() 메서드 모두 Overriding 한 경우

```
//#3. CASE3. equals(): 오버라이딩 O + hashcode(): 오버라이딩 O
Set<C> hashSet3 = new HashSet<>();                                    ⑤

C c1 = new C(3);
C c2 = new C(3);
System.out.println(c1==c2); //false
System.out.println(c1.equals(c2)); //true
System.out.println(c1.hashCode() + " " + c2.hashCode());

hashSet3.add(c1);
hashSet3.add(c2);
System.out.println(hashSet3.size()); //1 (같은 객체)
```

④

Objects.hash(Object… values)
→ 매개변수 값에 따른 해쉬값 리턴
→ 동일매개변수 + 동일순서 → 동일 해쉬

다른 표현
→ return new Integer(data).hashCode();
→ return data;

```
false
true
34 34
1
```

The End

Set<E> 컬렉션 – LinkedHashSet<E>

Set<E> 컬렉션 – LinkedHashSet<E>

☞ LinkedHashSet<E>

- Set<E> 인터페이스를 구현한 구현 클래스 (HashSet<E>의 자식 클래스, HashSet의 모든 기능 사용가능)
1 - 수집(collect)한 원소(Element)를 집합의 형태로 관리하며 저장공간(capacity)을 동적관리
- **입력 순서와 출력의 순서는 동일** (단, 중복원소의 경우 추가되지 않음)

- **데이터 추가**

2 3

```
Set<String> linkedSet1 = new LinkedHashSet<String>();
//#1. add(E element)
linkedSet1.add("가");
linkedSet1.add("나");
linkedSet1.add("가");
System.out.println(linkedSet1.toString()); //[가, 나]

//#2. addAll(Collection<? extends E> c)
Set<String> linkedSet2 = new LinkedHashSet<String>();
linkedSet2.add("나");
linkedSet2.add("다");
linkedSet2.addAll(linkedSet1);
System.out.println(linkedSet2.toString()); //[나, 다, 가]
```

[가, 나]

입력 순서와 동일

[나, 다, 가]

Set<E> 컬렉션 – LinkedHashSet<E>

☞ LinkedHashSet<E>

- **데이터 삭제**

1
2

```
//#3. remove(Object o)
linkedSet2.remove("나");
System.out.println(linkedSet2.toString()); //[다, 가]

//#4. clear()
linkedSet2.clear();
System.out.println(linkedSet2.toString()); //[]
```

```
[다, 가]

[]
```

Set<E> 컬렉션 – LinkedHashSet<E>

☞ LinkedHashSet<E>

- **데이터 정보 추출**

```
//#5. isEmpty()
System.out.println(linkedSet2.isEmpty()); //true

//#6. contains(Object o)
Set<String> linkedSet3 = new LinkedHashSet<String>();
linkedSet3.add("가");
linkedSet3.add("다");
linkedSet3.add("나");
System.out.println(linkedSet3.contains("나")); //true
System.out.println(linkedSet3.contains("라")); //false

//#7. size()
System.out.println(linkedSet3.size()); //3

//#8. iterator()
Iterator<String> iterator = linkedSet3.iterator();
while(iterator.hasNext()) {
  System.out.print(iterator.next() + " "); //가 다 나
}
System.out.println();
```

```
true

true
false

3

가 다 나
```

Set<E> 컬렉션 – LinkedHashSet<E>

☞ LinkedHashSet<E>

- **Set → 배열**

```
//#9. toArray()
Object[] objArray = linkedSet3.toArray();
System.out.println(Arrays.toString(objArray)); //[가 다 나]

//#10-1. toArray(T[] t)
String[] strArray1 = linkedSet3.toArray(new String[0]);
System.out.println(Arrays.toString(strArray1)); //[가 다 나]

//#10-2. toArray(T[] t)
String[] strArray2 = linkedSet3.toArray(new String[5]);
System.out.println(Arrays.toString(strArray2)); //[가 다 나 null null]
```

1 [가 다 나]

2 [가 다 나]

3 [가 다 나 null null]

Set<E> 컬렉션 – TreeSet<E>

Set<E> 컬렉션 – TreeSet<E>

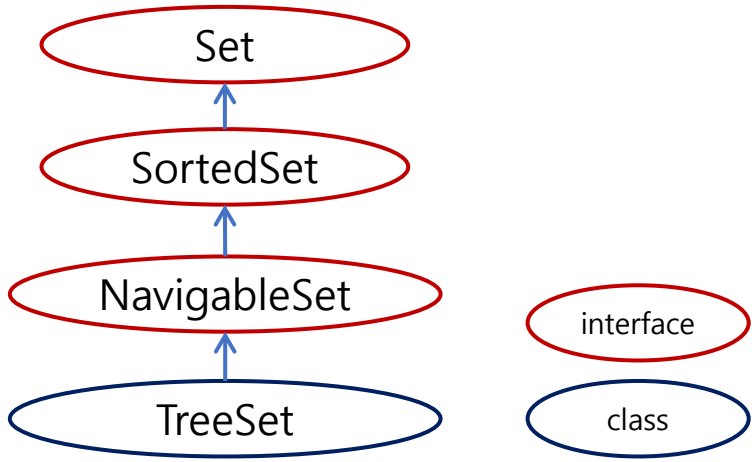
☞ TreeSet<E>

1
- Set<E> 인터페이스를 구현한 구현 클래스
- 수집(collect)한 원소(Element)를 집합의 형태로 관리하며 저장공간(capacity)을 동적관리
- **입력 순서와 관계없이 크기순으로 출력 (저장원소(Element)는 대소비교가 가능해야 함)**

2

TreeSet<E> = **Set<E>의 기본기능** + **정렬/검색 기능추가**

3



- Set<E>으로 객체타입을 선언하는 경우 **추가된 정렬/검색 기능사용 불가**

4
```
Set<String> treeSet = new TreeSet<String>();
treeSet._______ → Set<E> 메서드만 사용가능
```

- 추가된 정렬기능을 사용하기 위해선 **TreeSet<E> 객체 타입 선언**

5
```
TreeSet<String> treeSet = new TreeSet<String>();
treeSet._______ → Set<E> 메서드 + 추가된 정렬/검색 기능 메서드
```

Set<E> 컬렉션 – TreeSet<E>

☞ TreeSet<E>

1 - Set<E> 기본 메서드 이외에 TreeSet<E>에서 **추가로** 사용할 수 있는 **정렬/검색 메서드**

2

구분	리턴타입	메서드이름	기능
데이터 검색	E	first()	Set 원소들 중 가장 작은(lowest) 원소값 리턴
	E	last()	Set 원소들 중 가장 큰(highest) 원소값 리턴
	E	lower(E element)	매개변수로 입력된 원소보다 작은 가장 큰 수
	E	higher(E element)	매개변수로 입력된 원소보다 큰 가장 작은 수
	E	floor(E element)	매개변수로 입력된 원소보다 같거나 작은 가장 큰 수
	E	ceiling(E element)	매개변수로 입력된 원소보다 같거나 큰 가장 작은 수
데이터 꺼내기	E	pollFirst()	Set 원소들 중 가장 작은(lowest) 원소값을 꺼내어 리턴
	E	pollLast()	Set 원소들 중 가장 큰(highest) 원소값을 꺼내어 리턴

Set<E> 컬렉션 – TreeSet<E>

☞ TreeSet<E>

- Set<E> 기본 메서드 이외에 TreeSet<E>에서 추가로 사용할 수 있는 정렬/검색 메서드

구분	리턴타입	메서드이름	기능
1 데이터 부분집합 (SubSet) 생성	SortedSet<E>	headSet(E toElement)	매개변수보다 작은 모든 원소들로 구성된 Set을 리턴 (디폴트: 매개변수값 미포함)
	NavigableSet<E>	headSet(E toElement, boolean inclusive)	첫 번째 매개변수보다 작은 모든 원소들로 구성된 Set을 리턴 두 번째 매개변수 값에 따라 첫 번째 매개변수 포함여부 결정
	SortedSet<E>	tailSet(E fromElement)	매개변수보다 큰 모든 원소들로 구성된 Set을 리턴 (디폴트: 매개변수값 포함)
	NavigableSet<E>	tailSet(E fromElement, boolean inclusive)	첫 번째 매개변수보다 큰 모든 원소들로 구성된 Set을 리턴 두 번째 매개변수 값에 따라 첫 번째 매개변수 포함여부 결정
	SortedSet<E>	subSet(E fromElement, E toElement)	첫 번째 매개변수보다 크고 두 번째 매개변수보다 작은 원소들로 이루어진 Set을 리턴 (디폴트: fromElement 포함, toElement 미포함)
	NavigableSet<E>	subSet(E fromElement, boolean frominclusive, E toElement, boolean toinclusive)	첫 번째 매개변수보다 크고 세 번째 매개변수보다 작은 원소들로 이루어진 Set을 리턴 두 번째와 네 번째 매개변수 값에 따라 첫 번째와 세 번째 원소의 포함여부 결정 (디폴트: fromElement 포함, toElement 미포함)
데이터정렬	NavigableSet<E>	descendingSet()	내림차순의 의미가 아니라 **현재 정렬 기준을 반대로 변환**

Set<E> 컬렉션 – TreeSet<E>

☞ TreeSet<E>

- **데이터의 검색**

1
```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
for(int i=50; i>0; i-=2) { treeSet.add(i); }
System.out.println(treeSet.toString()); //[2,4,6,...,50]
```

2
```
//#1. first()
System.out.println(treeSet.first()); //2
//#2. last()
System.out.println(treeSet.last()); //50
//#3. lower(E element)
System.out.println(treeSet.lower(26)); //24
//#4. higher(E element)
System.out.println(treeSet.higher(26)); //28
//#5. floor(E element)
System.out.println(treeSet.floor(25)); //24
System.out.println(treeSet.floor(26)); //26
//#6. ceiling(E element)
System.out.println(treeSet.ceiling(25)); //26
System.out.println(treeSet.ceiling(26)); //26
```

```
[2,4,6,...,50]

2

50

24

28

24
26

26
26
```

Set<E> 컬렉션 – TreeSet<E>

☞ TreeSet<E>

- **데이터의 꺼내기**

1
```
//#7. pollFirst()
int treeSetSize = treeSet.size();
System.out.println(treeSetSize); //25
for(int i=0; i<treeSetSize; i++) {
  System.out.print(treeSet.pollFirst()+ " "); //2 4 6 ... 50
}
System.out.println();
System.out.println(treeSet.size()); //0
```

2
```
//#8. pollLast()
for(int i=50; i>0; i-=2) { treeSet.add(i); }

treeSetSize = treeSet.size();
System.out.println(treeSetSize); //25
for(int i=0; i<treeSetSize; i++) {
  System.out.print(treeSet.pollLast()+ " "); //50 48 46 ... 2
}
System.out.println();
System.out.println(treeSet.size()); //0
```

```
25
2 4 6 ... 50
0

25
50 48 46 … 2
0
```

Set<E> 컬렉션 – TreeSet<E>

☞ TreeSet<E>

- 데이터 부분집합(SubSet) 생성

```
1  //#9. SortedSet<E> headSet(E toElement)
   for(int i=50; i>0; i-=2) { treeSet.add(i); }
   SortedSet<Integer> sSet = treeSet.headSet(20);
   System.out.println(sSet.toString()); //[2, 4, 6, ..., 18]

2  //#10. NavigableSet<E> headSet(E toElement, boolean inclusive)
   NavigableSet<Integer> nSet = treeSet.headSet(20, false);
   System.out.println(nSet.toString()); //[2, 4, 6, ..., 18]
   nSet = treeSet.headSet(20, true);
   System.out.println(nSet.toString()); //[2, 4, 6, ..., 20]

3  //#11. SortedSet<E> tailSet(E toElement)
   sSet = treeSet.tailSet(20);
   System.out.println(sSet.toString()); //[20, 22, 24, ..., 50]

4  //#12. NavigableSet<E> tailSet(E toElement, boolean inclusive)
   nSet = treeSet.tailSet(20, false);
   System.out.println(nSet.toString()); //[22, 24, 26 ..., 50]
   nSet = treeSet.tailSet(20, true);
   System.out.println(nSet.toString()); //[20, 22, 24, ..., 50]
```

```
[2, 4, 6, ..., 18]

[2, 4, 6, ..., 18]
[2, 4, 6, ..., 20]

[20, 22, 24, ..., 50]

[22, 24, 26 ..., 50]
[20, 22, 24, ..., 50]
```

Set<E> 컬렉션 – TreeSet<E>

☞ TreeSet<E>

- 데이터 부분집합(SubSet) 생성

```
//#13. SortedSet<E> subSet(E fromElement, E toElement)
sSet = treeSet.subSet(10, 20);
System.out.println(sSet.toString()); //[10, 12, 14, 16, 18]

//#14. NavigableSet<E> subSet(E fromElement, boolean
                    frominclusive,E toElement, boolean toinclusive)
nSet = treeSet.subSet(10, true, 20, false);
System.out.println(nSet.toString()); //[10, 12, 14, 16, 18]
nSet = treeSet.subSet(10, false, 20, true);
System.out.println(nSet.toString()); //[12, 14, 16, 18, 20]
```

```
[10, 12, 14, 16, 18]

[10, 12, 14, 16, 18]

[12, 14, 16, 18, 20]
```

- 데이터 정렬

```
//#15. NavigableSet<E> descendingSet()
System.out.println(treeSet); //[2, 4, 6, ..., 50]
NavigableSet<Integer> descendingSet = treeSet.descendingSet();
System.out.println(descendingSet); //[50, 48, 46, ..., 2]
descendingSet = descendingSet.descendingSet();
System.out.println(descendingSet); //[2, 4, 6, ..., 50]
```

```
[2, 4, 6, ..., 50]

[50, 48, 46, ..., 2]

[2, 4, 6, ..., 50]
```

Set<E> 컬렉션 – TreeSet<E>

1 **TreeSet<E>에서 크기비교**

Integer

2

```
//#1. Integer 크기 비교
TreeSet<Integer> treeSet1 = new TreeSet<Integer>();
Integer intValue1 = new Integer(20);
Integer intValue2 = new Integer(10);
treeSet1.add(intValue1);
treeSet1.add(intValue2);
System.out.println(treeSet1.toString()); //[10, 20]
```

→ intValue1 > intValue2

String

3

```
//#2. String 크기 비교
TreeSet<String> treeSet2 = new TreeSet<String>();
String str1 = "가나";
String str2 = "다라";
treeSet2.add(str1);
treeSet2.add(str2);
System.out.println(treeSet2.toString()); //[가나, 다라]
```

→ str1 < str2

Set<E> 컬렉션 – TreeSet<E>

TreeSet<E>에서 크기비교

1

```
class MyClass{
  int data1;
  int data2;
  public MyClass(int data1, int data2) {
    this.data1=data1;
    this.data2=data2;
  }
}
```

5

Class Integer

java.lang.Object
  java.lang.Number
    java.lang.Integer

All Implemented Interfaces:

Serializable, Comparable<Integer>

Class String

java.lang.Object
  java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

MyClass

2

```
//#3. MyClass 크기 비교
TreeSet<MyClass> treeSet3 = new TreeSet<MyClass>();
MyClass myClass1 = new MyClass(2,5);
MyClass myClass2 = new MyClass(3,3);
treeSet3.add(myClass1);                    //예외발생
treeSet3.add(myClass2);                    //예외발생
System.out.println(treeSet3.toString()); //예외발생
```

myClass1 ??? myClass2

4

방법#1.
Comparable<T> interface 구현
방법#2.
TreeSet 생성자 매개변수로
Comparator<T> 객체 제공

3

크다/작다의 기준을 제공해주어야 함

Set<E> 컬렉션 – TreeSet<E>

MyClass 객체에 크기비교기능 부여

1

방법#1 java.lang.Comparable<T> interface 구현

→ int compareTo(T t) 추상메서드 포함 (함수적 인터페이스)

→ 매개변수 t 보다 작은 경우: -1
매개변수 t와 같은 경우 : 0
매개변수 t 보다 큰 경우 : 1

2

```
class MyClass{
  int data1;
  int data2;
  public MyClass(int data1, int data2) {
    this.data1=data1;
    this.data2=data2;
  }
}
```

3

```
class MyComparableClass
                    implements Comparable<MyComparableClass>{
  int data1;
  int data2;
  public MyComparableClass(int data1, int data2) {
    this.data1=data1;
    this.data2=data2;
  }
  @Override
  public int compareTo(MyComparableClass m) {
    if(data1<m.data1) { return -1; }
    else if (data1==m.data1) {return 0;}
    else return 1;
  }
}
```

→ data2는 상관없이 data1만으로 크기를 결정하는 예

Set<E> 컬렉션 – TreeSet<E>

MyClass 객체에 크기비교기능 부여

방법#1 java.lang.Comparable<T> interface 구현

→ int **compareTo(T t)** 추상메서드 포함 (함수적 인터페이스)

→ 매개변수 t 보다 **작은** 경우: **-1**
매개변수 t와 **같은** 경우 : **0**
매개변수 t 보다 **큰** 경우 : **1**

1

```
//#4. MyComparableClass 크기 비교
TreeSet<MyComparableClass> treeSet4 = new TreeSet<MyComparableClass>();
MyComparableClass myComparableClass1 = new MyComparableClass(2,5);
MyComparableClass myComparableClass2 = new MyComparableClass(3,3);
treeSet4.add(myComparableClass1);
treeSet4.add(myComparableClass2);
for(MyComparableClass mcc : treeSet4) {
  System.out.println(mcc.data1);
}
```

2

```
myComparableClass1 < myComparableClass2
```

```
2
3
```

Set<E> 컬렉션 – TreeSet<E>

MyClass 객체에 크기비교기능 부여

방법#2 TreeSet 생성자 매개변수로 java.util.**Comparator<T>** interface 객체 제공

→ int compare (T t1, T t2) 추상메서드 포함

t1<t2 인 경우 : **-1**
t1=t2 인 경우 : **0**
t1>t2 인 경우 : **1**

1

```
//#5. MyClass 크기 비교 (Comparator<T> 객체 생성자 전달)
TreeSet<MyClass> treeSet5 = new TreeSet<MyClass>(new Comparator<MyClass>() {
  @Override
  public int compare(MyClass o1, MyClass o2) {
    if(o1.data1 < o2.data1) return -1;
    else if(o1.data1 == o2.data1) return 0;
    else return 1;
  }
});

MyClass myClass1 = new MyClass(2,5);
MyClass myClass2 = new MyClass(3,3);
treeSet5.add(myClass1);
treeSet5.add(myClass2);
for(MyClass mc : treeSet5) {
  System.out.println(mc.data1);
}
```

2

3

→ **myClass1 < myClass2**

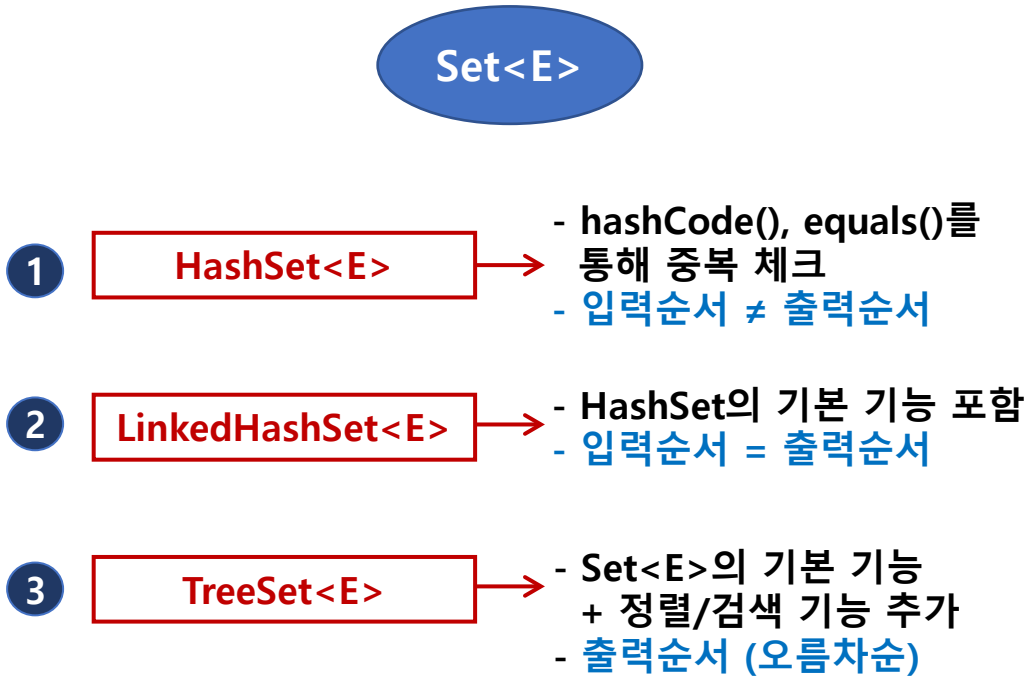
```
2
3
```

Set<E> 컬렉션 – Summary

Set<E> 컬렉션 – TreeSet<E>

```
[가, 다, 마, 나]
[다, 마, 나, 가]
[가, 나, 다, 마]
```

☞ Set<E>의 구현객체 정리




```java
//#1. HashSet
Set<String> hashSet = new HashSet<String>();
hashSet.add("다");
hashSet.add("마");
hashSet.add("나");
hashSet.add("가");
System.out.println(hashSet.toString()); //[가, 다, 마, 나]

//#2. LinkedHashSet
Set<String> linkedHashSet = new LinkedHashSet<String>();
linkedHashSet.add("다");
linkedHashSet.add("마");
linkedHashSet.add("나");
linkedHashSet.add("가");
System.out.println(linkedHashSet.toString()); //[다, 마, 나, 가]

//#3. TreeSet
Set<String> treeSet = new TreeSet<String>();
treeSet.add("다");
treeSet.add("마");
treeSet.add("나");
treeSet.add("가");
System.out.println(treeSet.toString()); //[가, 나, 다, 마]
```

4

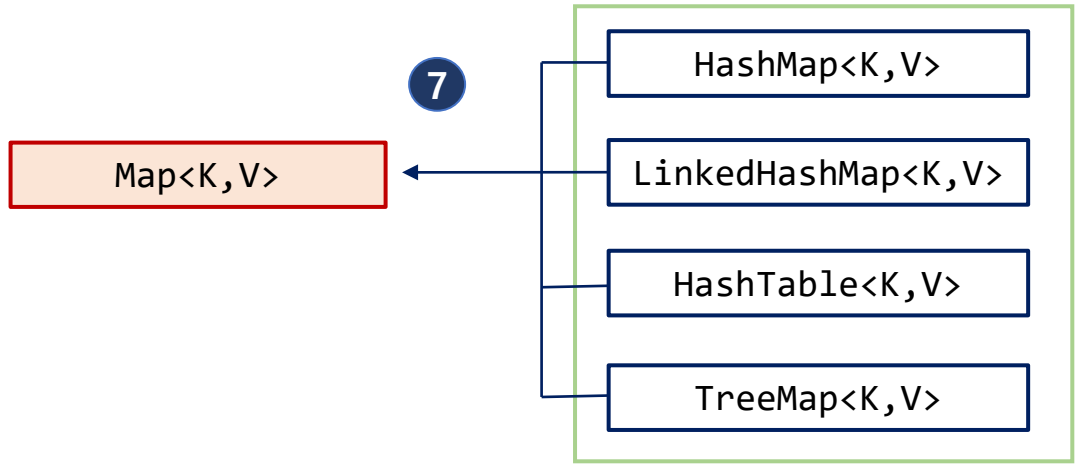
5

6

The End

Map<K, V> 컬렉션의 공통특성

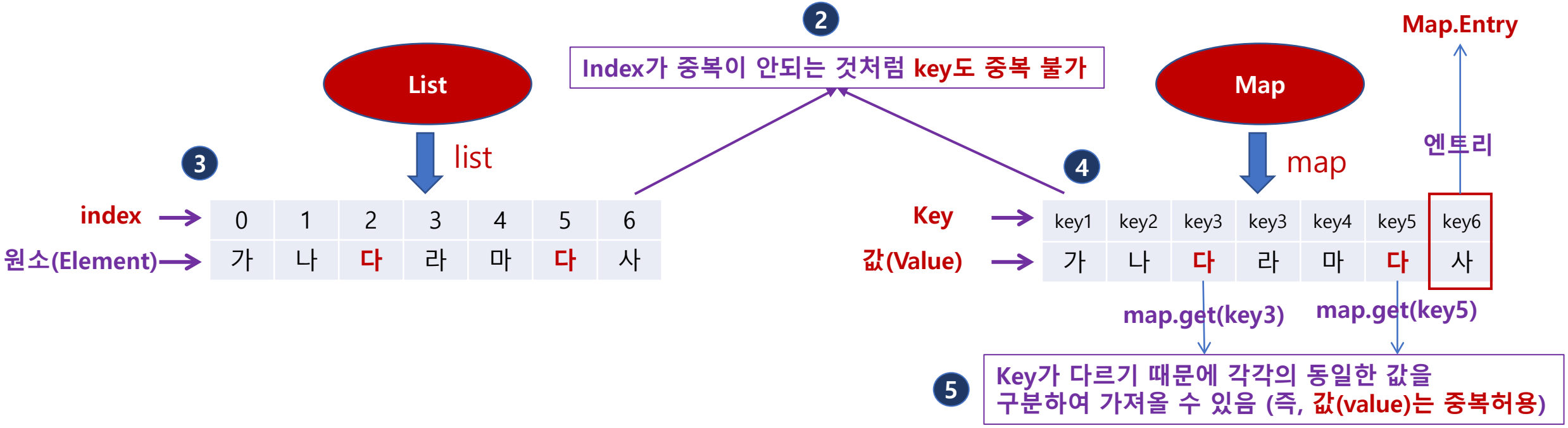
Map<K, V> 컬렉션



☞ Map<K, V> 컬렉션의 특징

1
- Key와 Value 한 쌍(Entry)으로 데이터를 저장
- Key는 중복저장 불가, Value는 중복 가능
- Collection과는 별개의 interface임 (List, Set과 기본 메서드 다름)

6
아래의 Map 객체는 6개의 Map.Entry 객체를 포함



Map<K, V> 컬렉션

☞ Map<K, V> 컬렉션의 주요 메서드

1 **Map<K, V>의 구현클래스는 아래의 모든 메서드를 포함**

구분	리턴타입	메서드이름	기능
데이터 추가	V	put(K key, V value)	입력 매개변수 (키, 값)을 Map 객체에 추가
	void	putAll(Map<? extends K, ? extends V> m)	입력 매개변수의 Map 객체를 통째로 추가
데이터 변경	V	replace(K key, V value)	Key 값에 해당하는 값을 value로 변경 (old값 리턴) (단, 해당 key가 없으면 null 리턴)
	boolean	replace(K key, V oldValue, V newValue)	(key, oldValue)의 쌍을 가지는 엔트리에서 값을 newValue로 변경 (단, 해당 엔트리가 없으면 false를 리턴)
데이터 정보추출	V	get(Object key)	매개변수의 key값에 해당하는 value 값을 리턴
	boolean	containsKey(Object key)	매개변수의 key값이 포함되어 있는지 여부
	boolean	containsValue(Object value)	매개변수의 value값이 포함되어 있는지 여부
	Set<K>	keySet()	Map 데이터들 중 key들만 뽑아서 Set 객체로 리턴
	Set<Entry<K,V>>	entrySet()	Map의 각 Entry들을 Set 객체로 담아 리턴
	int	size()	Map에 포함된 데이터(Entry)의 개수

2

Map<K, V> 컬렉션

☞ Map<K, V> 컬렉션의 주요 메서드

Map<K, V>의 구현클래스는 아래의 모든 메서드를 포함

구분	리턴타입	메서드이름	기능
1 데이터 삭제	V	remove(Object key)	입력 매개변수의 key를 갖는 엔트리 삭제 (단, 해당 key가 없으면 아무런 동작 안함)
	boolean	remove(Object key, Object value)	입력 매개변수의 (key, value)를 갖는 엔트리 삭제 (단, 해당 엔트리가 없으면 아무런 동작 안함)
	void	clear()	Map 객체내의 모든 데이터 삭제

Map<K, V> 컬렉션 – HashMap<K,V>

Map<K, V> 컬렉션 – HashMap<K, V>

☞ HashMap<K, V>

디폴트값은 16이며 원소가 16을 넘는 경우 자동으로 저장공간 확대

1
- Map<K, V> 인터페이스를 구현한 대표적인 구현 클래스
- Key, Value의 쌍으로 데이터를 관리하며 저장공간(capacity)을 동적관리
- 입력의 순서와 출력의 순서는 동일하지 않을 수 있음 **(Key 값이 Set으로 관리)**

- **데이터 추가**

2
3

```
Map<Integer, String> hMap1 = new HashMap<Integer, String>();

//#1. put(K key, V value)
hMap1.put(2, "나다라");
hMap1.put(1, "가나다");
hMap1.put(3, "다라마");
System.out.println(hMap1.toString()); //{1=가나다, 2=나다라, 3=다라마}

//#2. putAll(<Map<? extends K,? extends V> m)
Map<Integer, String> hMap2 = new HashMap<Integer, String>();
hMap2.putAll(hMap1);
System.out.println(hMap2.toString()); //{1=가나다, 2=나다라, 3=다라마}
```

입력순서와 불일치

{1=가나다, 2=나다라, 3=다라마}

{1=가나다, 2=나다라, 3=다라마}

Map<K, V> 컬렉션 – HashMap<K, V>

☞ HashMap<K, V>

- **데이터 변경**

1
2

```
//#3.replace(K key, V value)
hMap2.replace(1, "가가가");
hMap2.replace(4, "라라라"); //동작안함
System.out.println(hMap2.toString()); //{1=가가가, 2=나다라, 3=다라마}

//#4.replace(K key, V oldValue, V newValue)
hMap2.replace(1, "가가가", "나나나");
hMap2.replace(2, "다다다", "라라라"); //동작안함
System.out.println(hMap2.toString()); //{1=나나나, 2=나다라, 3=다라마}
```

```
{1=가가가, 2=나다라, 3=다라마}

{1=나나나, 2=나다라, 3=다라마}
```

Map<K, V> 컬렉션 – HashMap<K, V>

☞ HashMap<K, V>

- **데이터 정보추출**

```
//#5. V get(Object key)
System.out.println(hMap2.get(1)); //나나나
System.out.println(hMap2.get(2)); //나다라
System.out.println(hMap2.get(3)); //다라마

//#6. containsKey(Object key)
System.out.println(hMap2.containsKey(1)); //true
System.out.println(hMap2.containsKey(5)); //false

//#7. containsValue(Object value)
System.out.println(hMap2.containsValue("나나나")); //true
System.out.println(hMap2.containsValue("다다다")); //false
```

```
나나나
나다라
다라마

true
false

true
false
```

Map<K, V> 컬렉션 – HashMap<K, V>

☞ HashMap<K, V>

- **데이터 정보추출**

1
2
3

```
//#8. Set<K> keySet()
Set<Integer> keySet = hMap2.keySet();
System.out.println(keySet.toString()); //[1, 2, 3]

//#9. Set<Map.Entry<K,V>> entrySet()
Set<Map.Entry<Integer, String>> entrySet = hMap2.entrySet();
System.out.println(entrySet); //[1=나나나, 2=나다라, 3=다라마]

//#10. size()
System.out.println(hMap2.size()); //3
```

```
[1, 2, 3]

[1=나나나, 2=나다라, 3=다라마]

3
```

Map<K, V> 컬렉션 – HashMap<K, V>

☞ HashMap<K, V>

- **데이터 삭제**

```
//#11. remove(Object key)
hMap2.remove(1);
hMap2.remove(4); //동작안함
System.out.println(hMap2.toString()); //{2=나다라, 3=다라마}

//#12. remove(Object key, Object value)
hMap2.remove(2, "나다라");
hMap2.remove(3, "다다다"); //동작안함
System.out.println(hMap2.toString()); //{3=다라마}

//#13. clear()
hMap2.clear();
System.out.println(hMap2.toString()); //{}
```

1 `{2=나다라, 3=다라마}`

2 `{3=다라마}`

3 `{}`

Map<K, V> 컬렉션 – HashMap<K, V>

중복확인 메커니즘 이해를 위한 사전 지식

1

(hashCode()의 개념 + 등가연산(==)과 equals() 메서드의 차이점

hashCode(), equals() → Object 클래스의 메서드 → 모든 클래스 내에 포함

2

```
package sec02;

class H{ }
public class Test {
  public static void main(String[] ar) {
    H h = new H();
    System.out.println(h);
  }
}
```

<terminated> Test (4) [Java Ap
sec02.H@15db9742

패키지명.클래스명@해쉬코드

3

```
class A{
  int data;
  public A(int data) {
    this.data = data;
  }
}
```

```
A a1 = new A(3);
A a2 = new A(3);
System.out.println(a1==a2); //false
System.out.println(a1.equals(a2)); //false
```

4

```
Integer a1 = new Integer(3);
Integer a2 = new Integer(3);

System.out.println(a1==a2); //false
System.out.println(a1.equals(a2)); //true
```

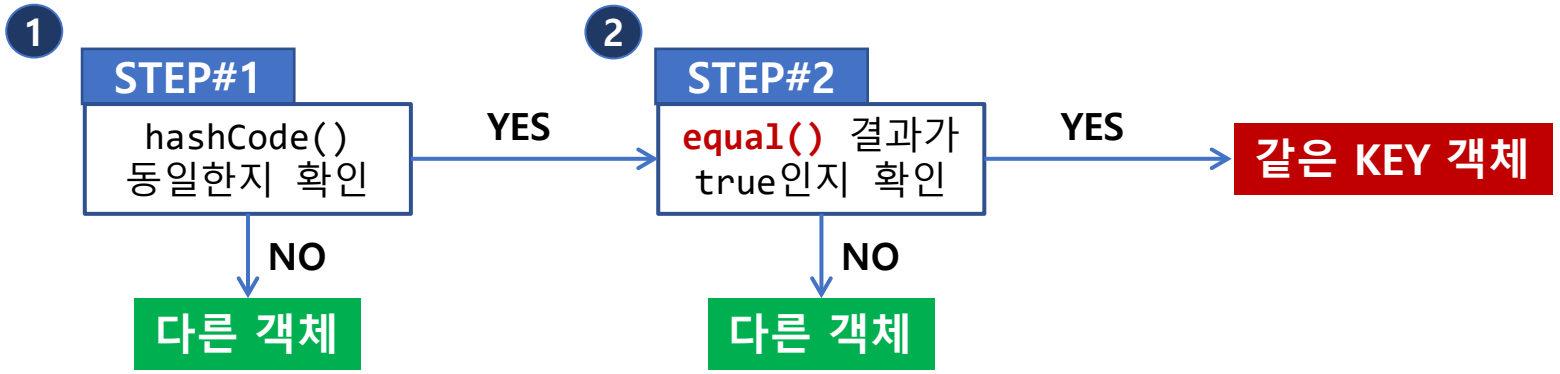
```
String s1 = new String("안녕");
String s2 = new String("안녕");

System.out.println(s1==s2); //false
System.out.println(s1.equals(s2)); //true
```

해쉬코드는 객체가 저장된 번지와 연관된 값 (실제 번지와는 다름)

Object의 equal()은 == 와 동일한 연산 (저장 번지 비교)

Map<K, V> 컬렉션 – HashMap<K, V>

HashMap<K, V>에서의 중복확인 메커니즘



3

동일객체의 정의를 위해서는 hashCode와 equals() 메서드 Overriding 필요

CASE 1

4

두 메서드 모두
Overriding을 하지 않은 경우

5

```
class A{
  int data;
  public A(int data) {
    this.data = data;
  }
}
```

6

```
//#1. CASE1. equals(): 오버라이딩 X + hashcode(): 오버라이딩 X
Map<A, String> hashMap1 = new HashMap<A, String>();
A a1 = new A(3);
A a2 = new A(3);
System.out.println(a1==a2); //false
System.out.println(a1.equals(a2)); //false
System.out.println(a1.hashCode() + " " + a2.hashCode());
hashMap1.put(a1, "첫번째");
hashMap1.put(a2, "두번째");
System.out.println(hashMap1.size()); //2 (다른 객체)
```

```
false
false
366712642 1829164700
2
```

Map<K, V> 컬렉션 – HashMap<K, V>

HashMap<K, V>에서의 중복확인 메커니즘

CASE 2

1

equals() 메서드만 Overriding 한 경우

2

```
class B{
  int data;
  public B(int data) {
    this.data = data;
  }
  @Override
  public boolean equals(Object obj) {
    if(obj instanceof B) {
      if(((B)obj).data == data)
        return true;
    }
    return false;
  }
}
```

3

```
//#2. CASE2. equals(): 오버라이딩 O + hashcode(): 오버라이딩 X
Map<B, String> hashMap2 = new HashMap<>();

B b1 = new B(3);
B b2 = new B(3);
System.out.println(b1==b2); //false
System.out.println(b1.equals(b2)); //true
System.out.println(b1.hashCode() + " " + b2.hashCode());

hashMap2.put(b1, "첫번째");
hashMap2.put(b2, "두번째");
System.out.println(hashMap2.size()); //2 (다른 객체)
```

```
false
true
2018699554 1311053135
2
```

Map<K, V> 컬렉션 – HashMap<K, V>

HashMap<K, V>에서의 중복확인 메커니즘

CASE 3

1

```
class C{
  int data;
  public C(int data) {
    this.data = data;
  }
  @Override                                    2
  public boolean equals(Object obj) {
    if(obj instanceof C) {
      if(((C)obj).data == data)
        return true;
    }
    return false;
  }
  @Override                                    3
  public int hashCode() {
    return Objects.hash(data); //data
  }
}
```

```
//#3. CASE3. equals(): 오버라이딩 O + hashcode(): 오버라이딩 O

Map<C, String> hashMap3 = new HashMap<>();                    4

C c1 = new C(3);
C c2 = new C(3);
System.out.println(c1==c2); //false
System.out.println(c1.equals(c2)); //true
System.out.println(c1.hashCode() + " " + c2.hashCode());

hashMap3.put(c1, "첫번째");
hashMap3.put(c2, "두번째");
System.out.println(hashMap3.size()); //1 (같은 객체)
```

```
false
true
34 34
1
```

equals(), hashCode() 메서드 모두 Overriding 한 경우

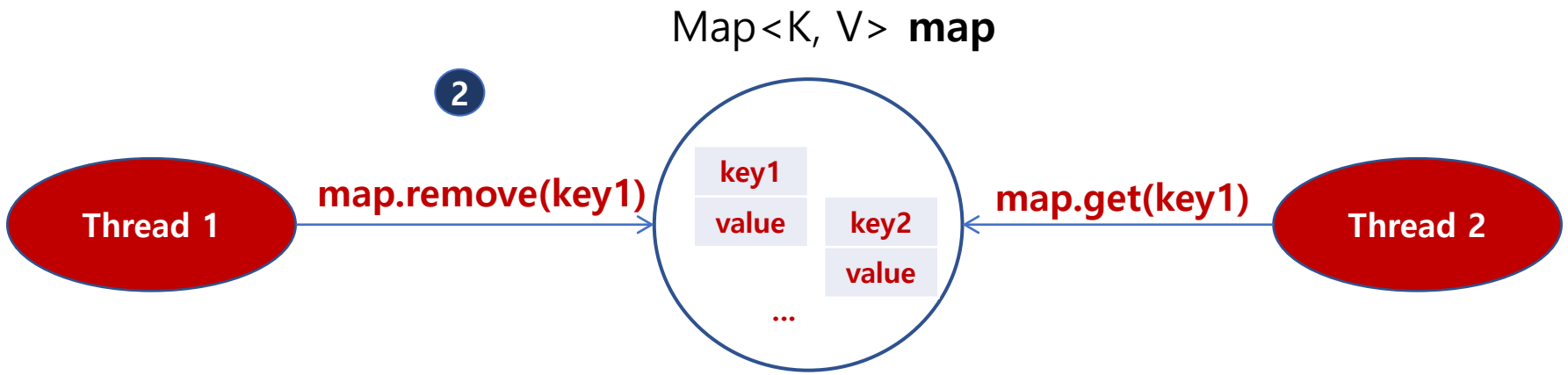
The End

Map<K, V> 컬렉션 – HashTable<K,V>

Map<K, V> 컬렉션 – HashTable<K, V>

☞ Hashtable<K, V>

1
- HashMap<K, V>이 단일 쓰레드에 적합한 반면 Hashtable은 쓰레드 안정성을 가짐
- 즉, 모든 메서드가 **동기화(synchronized) 메서드로 구현**되어 **멀티쓰레드에 적합**(Thread Safe)
- 입력의 순서와 출력의 순서는 동일하지 않을 수 있음



3
동기화 되지 않는 HashMap<K, V>/HashSet<E>도 다음과 같이 **동기화 Wrapping**을 통해 멀티쓰레드에서 사용가능

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<K, V>());
Set<E> m = Collections.synchronizedSet(new HashSet<E>());
```

Map<K, V> 컬렉션 – HashTable<K, V>

☞ Hashtable<K, V>

- **데이터 추가**

1
```
Map<Integer, String> hTable1 = new Hashtable<Integer, String>();
//#1. put(K key, V value)
hTable1.put(2, "나다라");
hTable1.put(1, "가나다");
hTable1.put(3, "다라마");
System.out.println(hTable1.toString()); //{3=다라마, 2=나다라 ,1=가나다}
```

2
```
//#2. putAll(<Map<? extends K,? extends V> m)
Map<Integer, String> hTable2 = new Hashtable<Integer, String>();
hTable2.putAll(hTable1);
System.out.println(hTable2.toString()); //{3=다라마, 2=나다라 ,1=가나다}
```

입력순서와
불일치

{3=다라마, 2=나다라, 1=가나다}

{3=다라마, 2=나다라, 1=가나다}

Map<K, V> 컬렉션 – HashTable<K, V>

☞ HashTable<K, V>

- **데이터 변경**

1

```
//#3.replace(K key, V value)
hTable2.replace(1, "가가가");
hTable2.replace(4, "라라라"); //동작안함
System.out.println(hTable2.toString()); //{3=다라마, 2=나다라, 1=가가가}
```

2

```
//#4.replace(K key, V oldValue, V newValue)
hTable2.replace(1, "가가가", "나나나");
hTable2.replace(2, "다다다", "라라라"); //동작안함
System.out.println(hTable2.toString()); //{3=다라마, 2=나다라, 1=나나나}
```

{3=다라마, 2=나다라, 1=가가가}

{3=다라마, 2=나다라, 1=나나나}

Map<K, V> 컬렉션 – HashTable<K, V>

☞ HashTable<K, V>

- **데이터 정보추출**

```
1  //#5. V get(Object key)
   System.out.println(hTable2.get(1)); //나나나
   System.out.println(hTable2.get(2)); //나다라
   System.out.println(hTable2.get(3)); //다라마

2  //#6. containsKey(Object key)
   System.out.println(hTable2.containsKey(1)); //true
   System.out.println(hTable2.containsKey(5)); //false

3  //#7. containsValue(Object value)
   System.out.println(hTable2.containsValue("나나나")); //true
   System.out.println(hTable2.containsValue("다다다")); //false
```

```
나나나
나다라
다라마

true
false

true
false
```

Map<K, V> 컬렉션 – HashTable<K, V>

☞ HashTable<K, V>

- **데이터 정보추출**

```
1  //#8. Set<K> keySet()
   Set<Integer> keySet = hTable2.keySet();
   System.out.println(keySet.toString()); //[3, 2, 1]

2  //#9. Set<Map.Entry<K,V>> entrySet()
   Set<Map.Entry<Integer, String>> entrySet = hTable2.entrySet();
   System.out.println(entrySet); //[1=나나나, 2=나다라, 3=다라마]

3  //#10. size()
   System.out.println(hTable2.size()); //3
```

```
[3, 2, 1]

[1=나나나, 2=나다라, 3=다라마]

3
```

Map<K, V> 컬렉션 – HashTable<K, V>

☞ HashTable<K, V>

- **데이터 삭제**

```
//#11. remove(Object key)
hTable2.remove(1);
hTable2.remove(4); //동작안함
System.out.println(hTable2.toString()); //{2=나다라, 3=다라마}

//#12. remove(Object key, Object value)
hTable2.remove(2, "나다라");
hTable2.remove(3, "다다다"); //동작안함
System.out.println(hTable2.toString()); //{3=다라마}

//#13. clear()
hTable2.clear();
System.out.println(hTable2.toString()); //{}
```

1 `{2=나다라, 3=다라마}`

2 `{3=다라마}`

3 `{}`

Map<K, V> 컬렉션 – LinkedHashMap<K,V>

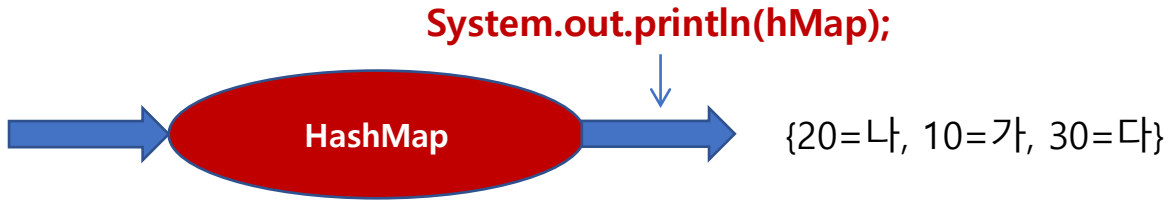
Map<K, V> 컬렉션 – LinkedHashMap<K, V>

☞ LinkedHashMap<K, V>

1 - HashMap<K, V>과 동일한 기능을 수행 (데이터의 추가, 변경, 삭제 등 메서드)
- 입력 순서 = 출력 순서

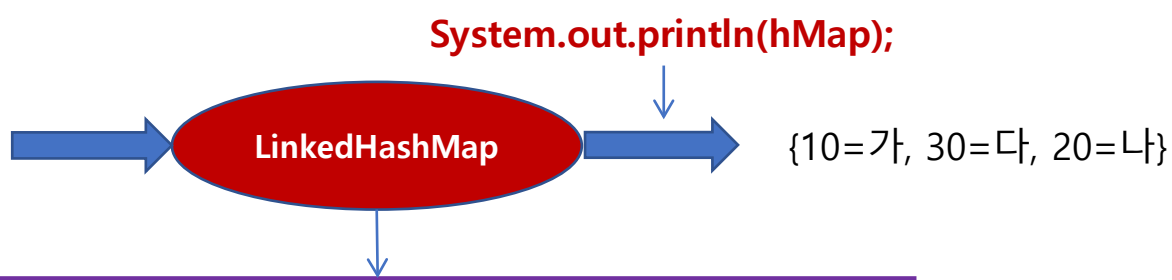
2

```
hMap.put(10,"가");
hMap.put(30,"다");
hMap.put(20,"나");
```

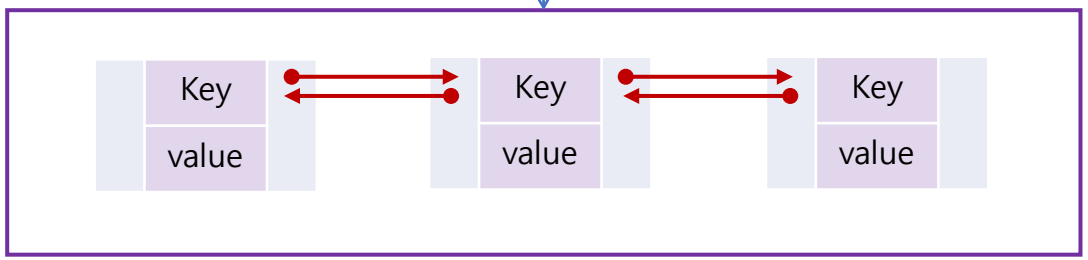


3

```
hMap.put(10,"가");
hMap.put(30,"다");
hMap.put(20,"나");
```



4



Map<K, V> 컬렉션 – LinkedHashMap<K, V>

☞ LinkedHashMap<K, V>

- **데이터 추가**

```
Map<Integer, String> lhMap1 = new LinkedHashMap<Integer, String>();

//#1. put(K key, V value)
lhMap1.put(2, "나다라");
lhMap1.put(1, "가나다");
lhMap1.put(3, "다라마");
System.out.println(lhMap1.toString()); //{2=나다라, 1=가나다, 3=다라마}

//#2. putAll(<Map<? extends K,? extends V> m)
Map<Integer, String> lhMap2 = new LinkedHashMap<Integer, String>();
lhMap2.putAll(lhMap1);
System.out.println(lhMap2.toString()); //{2=나다라, 1=가나다, 3=다라마}
```

1

2

입력순서와 일치

{2=나다라, 1=가나다, 3=다라마}

{2=나다라, 1=가나다, 3=다라마}

Map<K, V> 컬렉션 – LinkedHashMap<K, V>

☞ LinkedHashMap<K, V>

- **데이터 변경**

1

```
//#3.replace(K key, V value)
lhMap2.replace(1, "가가가");
lhMap2.replace(4, "라라라"); //동작안함
System.out.println(lhMap2.toString()); //{2=나다라, 1=가가가, 3=다라마}
```

2

```
//#4.replace(K key, V oldValue, V newValue)
lhMap2.replace(1, "가가가", "나나나");
lhMap2.replace(2, "다다다", "라라라"); //동작안함
System.out.println(lhMap2.toString()); //{2=나다라, 1=나나나, 3=다라마}
```

{2=나다라, 1=가가가, 3=다라마}

{2=나다라, 1=나나나, 3=다라마}

Map<K, V> 컬렉션 – LinkedHashMap<K, V>

☞ LinkedHashMap<K, V>

- **데이터 정보추출**

1
```
//#5. V get(Object key)
System.out.println(lhMap2.get(1)); //나나나
System.out.println(lhMap2.get(2)); //나다라
System.out.println(lhMap2.get(3)); //다라마
```

2
```
//#6. containsKey(Object key)
System.out.println(lhMap2.containsKey(1)); //true
System.out.println(lhMap2.containsKey(5)); //false
```

3
```
//#7. containsValue(Object value)
System.out.println(lhMap2.containsValue("나나나")); //true
System.out.println(lhMap2.containsValue("다다다")); //false
```

```
나나나
나다라
다라마

true
false

true
false
```

Map<K, V> 컬렉션 – LinkedHashMap<K, V>

☞ LinkedHashMap<K, V>

- **데이터 정보추출**

```
1  //#8. Set<K> keySet()
   Set<Integer> keySet = lhMap2.keySet();
   System.out.println(keySet.toString()); //[2, 1, 3]

2  //#9. Set<Map.Entry<K,V>> entrySet()
   Set<Map.Entry<Integer, String>> entrySet = lhMap2.entrySet();
   System.out.println(entrySet); //[2=나다라, 1=나나나, 3=다라마]

3  //#10. size()
   System.out.println(lhMap2.size()); //3
```

```
[2, 1, 3]

[2=나다라, 1=나나나, 3=다라마]

3
```

Map<K, V> 컬렉션 – LinkedHashMap<K, V>

☞ LinkedHashMap<K, V>

- **데이터 삭제**

```
//#11. remove(Object key)
1 lhMap2.remove(1);
lhMap2.remove(4); //동작안함
System.out.println(lhMap2.toString()); //{2=나다라, 3=다라마}

//#12. remove(Object key, Object value)
2 lhMap2.remove(2, "나다라");
lhMap2.remove(3, "다다다"); //동작안함
System.out.println(lhMap2.toString()); //{3=다라마}

//#13. clear()
3 lhMap2.clear();
System.out.println(lhMap2.toString()); //{}
```

```
{2=나다라, 3=다라마}

{3=다라마}

{}
```

The End

Map<K, V> 컬렉션 – TreeMap<K,V>

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

1
- Map<K, V> 인터페이스를 구현한 구현 클래스
- 엔트리(Key, Value) 집합의 형태로 관리하며 저장공간(capacity)을 동적관리
- **입력 순서와 관계없이 key 값의 크기순으로 출력 (key 값은 대소비교가 가능해야 함)**

2 **TreeMap<K, V>** = **Map<K, V>의 기본기능** + **정렬/검색 기능추가**

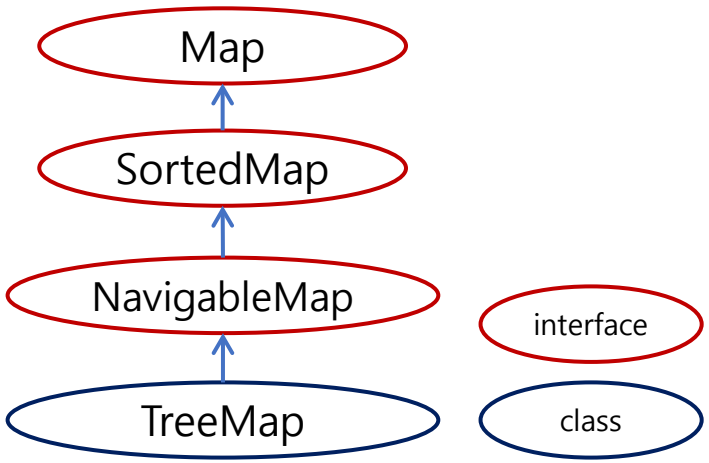
- Map<K, V>으로 객체타입을 선언하는 경우 **추가된 정렬/검색 기능사용 불가**

4
```
Map<Integer, String> treeMap = new TreeMap<Integer, String>();
treeMap._______ → Map<K, V> 메서드만 사용가능
```

- 추가된 정렬기능을 사용하기 위해선 **TreeMap<K, V> 객체 타입 선언**

5
```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
treeMap._______ → Map<K, V> 메서드 + 추가된 정렬/검색 기능 메서드
```

3



Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

1 - Map<K, V> 기본 메서드 이외에 TreeMap<K, V>에서 추가로 사용할 수 있는 정렬/검색 메서드

2

구분	리턴타입	메서드이름	기능
데이터 검색	K	firstKey()	Map 원소들 중 가장 작은(lowest) key 값 리턴
	Map.Entry<K,V>	firstEntry()	Map 원소들 중 가장 작은(lowest) key 값을 가진 Entry 리턴
	K	lastKey()	Map 원소들 중 가장 큰(highest) key 값 리턴
	Map.Entry<K,V>	lastEntry()	Map 원소들 중 가장 큰(highest) key 값을 가진 Entry 리턴
	K	lowerKey(K key)	매개변수로 입력된 key값보다 작은 key 들 중 가장 큰 key 값
	Map.Entry<K,V>	lowerEntry(K key)	매개변수로 입력된 key값보다 작은 key 들 중 가장 큰 key 값을 가지는 Entry 리턴
	K	higherKey(K key)	매개변수로 입력된 key값보다 큰 key 들 중 가장 작은 key 값
	Map.Entry<K,V>	higherEntry(K key)	매개변수로 입력된 key값보다 큰 key 들 중 가장 작은 key 값을 가지는 Entry 리턴
데이터 꺼내기	Map.Entry<K,V>	pollFirstEntry()	Map 원소들 중 가장 작은(lowest) key 값을 가지는 Entry를 꺼내어 리턴
	Map.Entry<K,V>	pollLastEntry()	Map 원소들 중 가장 큰(highest) key 값을 가지는 Entry를 꺼내어 리턴

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- Map<K, V> 기본 메서드 이외에 TreeMap<K, V>에서 추가로 사용할 수 있는 정렬/검색 메서드

구분	리턴타입	메서드이름	기능
① 데이터 부분집합 (SubSet) 생성	SortedMap<K, V>	headMap(K toKey)	toKey보다 작은 key 값을 가지는 모든 Entry를 포함한 Map 리턴 (디폴트: 매개변수값 미포함)
	NavigableMap<K, V>	headMap(K toKey, boolean inclusive)	toKey 작은 key 값을 가지는 모든 Entry를 포함한 Map 리턴 Inclusive 값에 따라 toKey값의 포함여부 결정
	SortedMap<K, V>	tailMap(K fromKey)	toKey 큰 key 값을 가지는 모든 Entry를 포함한 Map 리턴 (디폴트: 매개변수값 포함)
	NavigableMap<K, V>	tailMap(K fromKey, boolean inclusive)	fromKey보다 큰 key 값을 가지는 모든 Entry를 포함한 Map 리턴 inclusive 값에 따라 fromKey값의 포함여부 결정
	SortedMap<K,V>	subSet(K fromKey, K toKey)	fromKey 매개변수보다 같거나 크고 toKey보다 작은 key값을 가지는 모든 Entry를 포함한 Map 리턴 (디폴트: fromElement 포함, toElement 미포함)
	NavigableMap<K,V>	subSet(K fromKey, boolean frominclusive, K toKey, boolean toinclusive)	fromKey 매개변수보다 같거나 크고 toKey보다 작은 key값을 가지는 모든 Entry를 포함한 Map 리턴. frominclusive, toinclusive 값에 따라 fromKey, toKey의 포함여부 결정 (디폴트: fromElement 포함, toElement 미포함)

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- Map<K, V> 기본 메서드 이외에 TreeMap<K, V>에서 추가로 사용할 수 있는 정렬/검색 메서드

	구분	리턴타입	메서드이름	기능
1	데이터 정렬	NavigableSet<K>	descendingKeySet()	Map에 포함된 모든 Key값의 정렬을 반대로 변환한 Set 객체 리턴
		NavigableMap<K, V>	descendingMap()	Map에 포함된 모든 Key값의 정렬을 반대로 변환한 Map 객체 리턴

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- **데이터의 검색**

1
```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
for(int i=20; i>0; i-=2) { treeMap.put(i, i+"번째 데이터"); }
System.out.println(treeMap.toString());
                  //{2=2번째 데이터, 4=4번째 데이터,..20=20번째 데이터}
```

2
```
//#1. firstKey()
System.out.println(treeMap.firstKey()); //2
//#2. firstEntry()
System.out.println(treeMap.firstEntry()); //2=2번째 데이터
//#3. lastKey()
System.out.println(treeMap.lastKey()); //20
//#4. lasttEntry()
System.out.println(treeMap.lastEntry()); //20=20번째 데이터
//#5. lowerKey(K key)
System.out.println(treeMap.lowerKey(11)); //10
System.out.println(treeMap.lowerKey(10)); //8
//#6. higherKey(K key)
System.out.println(treeMap.higherKey(11)); //12
System.out.println(treeMap.higherKey(10)); //12
```

```
{2=2번째 데이터, 4=4번째 데이터,… ,20=20번째 데이터}

2

2=2번째 데이터

20

20=20번째 데이터

10
8

12
12
```

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- **데이터의 꺼내기**

1
```
//#7. pollFirstEntry()
System.out.println(treeMap.pollFirstEntry()); //2=2번째 데이터
System.out.println(treeMap.toString());
            //{4=4번째 데이터, 6=6번째 데이터, ...20=20번째 데이터}
```

2
```
//#8. pollLastEntry()
System.out.println(treeMap.pollLastEntry()); //20=20번째 데이터
System.out.println(treeMap.toString());
            //{4=4번째 데이터, 6=6번째 데이터, ...18=18번째 데이터}
```

```
2=2번째 데이터

{4=4번째 데이터, 6=6번째 데이터, ...20=20번째 데이터}

20=20번째 데이터

{4=4번째 데이터, 6=6번째 데이터, ...18=18번째 데이터}
```

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- 데이터 부분집합(SubMap) 생성 (1/2)

```
1  //#9. SortedMap<K,V> headMap(K toKey)
   SortedMap<Integer, String> sortedMap = treeMap.headMap(8);
   System.out.println(sortedMap); //{4=4번째 데이터, 6=6번째 데이터}

2  //#10. NavigableMap<K,V> headMap(K toKey, boolean inclusive)
   NavigableMap<Integer, String> navigableMap = treeMap.headMap(8, true);
   System.out.println(navigableMap);
                          //{4=4번째 데이터, 6=6번째 데이터, 8=8번째 데이터}

3  //#11. SortedMap<K,V> tailMap(K toKey)
   sortedMap = treeMap.tailMap(14);
   System.out.println(sortedMap);
                       //{14=14번째 데이터, 16=16번째 데이터, 18=18번째 데이터}

4  //#12. NavigableMap<K,V> tailMap(K toKey, boolean inclusive)
   navigableMap = treeMap.tailMap(14, false);
   System.out.println(navigableMap); //{16=16번째 데이터, 18=18번째 데이터}
```

{4=4번째 데이터, 6=6번째 데이터}

{4=4번째 데이터, 6=6번째 데이터, 8=8번째 데이터}

{14=14번째 데이터, 16=16번째 데이터, 18=18번째 데이터}

{16=16번째 데이터, 18=18번째 데이터}

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- 데이터 부분집합(SubMap) 생성 (2/2)

1
```
//#13. SortedMap<K,V> subMap(K fromKey, K toKey)
sortedMap = treeMap.subMap(6,10);
System.out.println(sortedMap); //{6=6번째 데이터, 8=8번째 데이터}
```

2
```
//#14. NavigableMap<K,V> subMap(K fromKey, boolean frominclusive,
                                K toKey, boolean toinclusive)
navigableMap = treeMap.subMap(6, false, 10, true);
System.out.println(navigableMap); //{8=8번째 데이터, 10=10번째 데이터}
```

```
{6=6번째 데이터, 8=8번째 데이터}

{8=8번째 데이터, 10=10번째 데이터}
```

Map<K, V> 컬렉션 – TreeMap<K, V>

☞ TreeMap<K, V>

- 데이터 정렬

1
```
//#15. NavigableSet<K> descendingKeySet()
NavigableSet<Integer> navigableSet = treeMap.descendingKeySet();
System.out.println(navigableSet); //[18, 16, 14, …, 4]
System.out.println(navigableSet.descendingSet()); //[4, 6, 8, …, 18]
```

2
```
//#16. NavigableMap<K,V> descendingMap()
navigableMap = treeMap.descendingMap();
System.out.println(navigableMap); //{18=18번째 데이터, 16=16번째 데이터...,4=4번째 데이터}
System.out.println(navigableMap.descendingMap()); // {4=4번째 데이터, 6=6번째 데이터,..., 18=18번째 데이터}
```

```
[18, 16, 14, ..., 4]
[4, 6, 8, ..., 18]
```

```
{18=18번째 데이터, 16=16번째 데이터...,4=4번째 데이터}
{4=4번째 데이터, 6=6번째 데이터,..., 18=18번째 데이터}
```

Map<K, V> 컬렉션 – TreeMap<K, V>

1 **TreeMap<K, V>에서 크기비교**

Integer

2

```
//#1. Integer 크기 비교
TreeMap<Integer, String> treeMap1 = new TreeMap<Integer, String>();
Integer intValue1 = new Integer(20);
Integer intValue2 = new Integer(10);
// intValue1 > intValue2
treeMap1.put(intValue1, "가나다");
treeMap1.put(intValue2, "나다라");
System.out.println(treeMap1.toString()); //{10=나다라, 20=가나다}
```

→ **intValue1 > intValue2**

String

3

```
//#2. String 크기 비교
TreeMap<String, Integer> treeMap2 = new TreeMap<String, Integer>();
String str1 = "가나";
String str2 = "다라";
// str1 < str2
treeMap2.put(str1, 10);
treeMap2.put(str2, 20);
System.out.println(treeMap2.toString()); //{가나=20, 다라=10}
```

→ **str1 < str2**

Map<K, V> 컬렉션 – TreeMap<K, V>

TreeMap<K, V>에서 크기비교

1

```
class MyClass{
  int data1;
  int data2;
  public MyClass(int data1, int data2) {
    this.data1=data1;
    this.data2=data2;
  }
}
```

5

Class Integer

java.lang.Object
  java.lang.Number
    java.lang.Integer

All Implemented Interfaces:

Serializable, Comparable<Integer>

Class String

java.lang.Object
  java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

MyClass

2

```
//#3. MyClass 객체를 Key로 사용
TreeMap<MyClass, String> treeMap3 = new TreeMap<>();
MyClass myClass1 = new MyClass(2,5);
MyClass myClass2 = new MyClass(3,3);
treeMap3.put(myClass1, "가나다");          //예외발생
treeMap3.put(myClass2, "가나다");          //예외발생
System.out.println(treeMap3.toString()); //예외발생
```

myClass1 ??? myClass2

4

방법#1.
Comparable<T> interface 구현
방법#2.
TreeMap 생성자 매개변수로
Comparator<T> 객체 제공

3

크다/작다의 기준을 제공해주어야 함

Map<K, V> 컬렉션 – TreeMap<K, V>

MyClass 객체에 크기비교기능 부여

1

방법#1 java.lang.Comparable<T> interface 구현

→ int **compareTo(T t)** 추상메서드 포함 (함수적 인터페이스)

→ 매개변수 t 보다 **작은** 경우: **-1**
매개변수 t와 **같은** 경우 : **0**
매개변수 t 보다 **큰** 경우 : **1**

2

```
class MyClass{
  int data1;
  int data2;
  public MyClass(int data1, int data2) {
    this.data1=data1;
    this.data2=data2;
  }
}
```

3

```
class MyComparableClass
                    implements Comparable<MyComparableClass>{
  int data1;
  int data2;
  public MyComparableClass(int data1, int data2) {
    this.data1=data1;
    this.data2=data2;
  }
  @Override
  public int compareTo(MyComparableClass m) {
    if(data1<m.data1) { return -1; }
    else if (data1==m.data1) {return 0;}
    else return 1;
  }
}
```

→ data2는 상관없이 data1만으로 크기를 결정하는 예

Map<K, V> 컬렉션 – TreeMap<K, V>

MyClass 객체에 크기비교기능 부여

방법#1 java.lang.Comparable<T> interface 구현

→ int **compareTo(T t)** 추상메서드 포함 (함수적 인터페이스)

→ 매개변수 t 보다 **작은** 경우: **-1**
매개변수 t와 **같은** 경우 : **0**
매개변수 t 보다 **큰** 경우 : **1**

1

```
//#4. MyComparableClass 크기 비교
TreeMap<MyComparableClass, String> treeMap4 = new TreeMap<>();
MyComparableClass myComparableClass1 = new MyComparableClass(2,5);
MyComparableClass myComparableClass2 = new MyComparableClass(3,3);
treeMap4.put(myComparableClass1, "가나다");
treeMap4.put(myComparableClass2, "나다라");
for(MyComparableClass mycomp : treeMap4.keySet()) {
  System.out.println(mycomp.data1); //2, 3
}
```

2

```
myComparableClass1 < myComparableClass2
```

```
2
3
```

Map<K, V> 컬렉션 – TreeMap<K, V>

MyClass 객체에 크기비교기능 부여

방법#2 TreeMap 생성자 매개변수로 java.util.**Comparator<T>** interface 객체 제공

1 → int **compare (T t1, T t2)** 추상메서드 포함

- t1<t2 인 경우 : -1
- t1=t2 인 경우 : 0
- t1>t2 인 경우 : 1

```
//#5. MyClass 크기 비교 (Comparator<T> 객체 생성자 전달)
TreeMap<MyClass, String> treeMap5 = new TreeMap<>(new Comparator<MyClass>() {
  @Override
  public int compare(MyClass o1, MyClass o2) {
    if(o1.data1 < o2.data1) return -1;
    else if(o1.data1 == o2.data1) return 0;
    else return 1;
  }});

MyClass myClass1 = new MyClass(2,5);
MyClass myClass2 = new MyClass(3,3);
treeMap5.put(myClass1, "가나다");
treeMap5.put(myClass2, "나다라");
for(MyClass mc : treeMap5.keySet()) {
  System.out.println(mc.data1); //2, 3
}
```

2

3

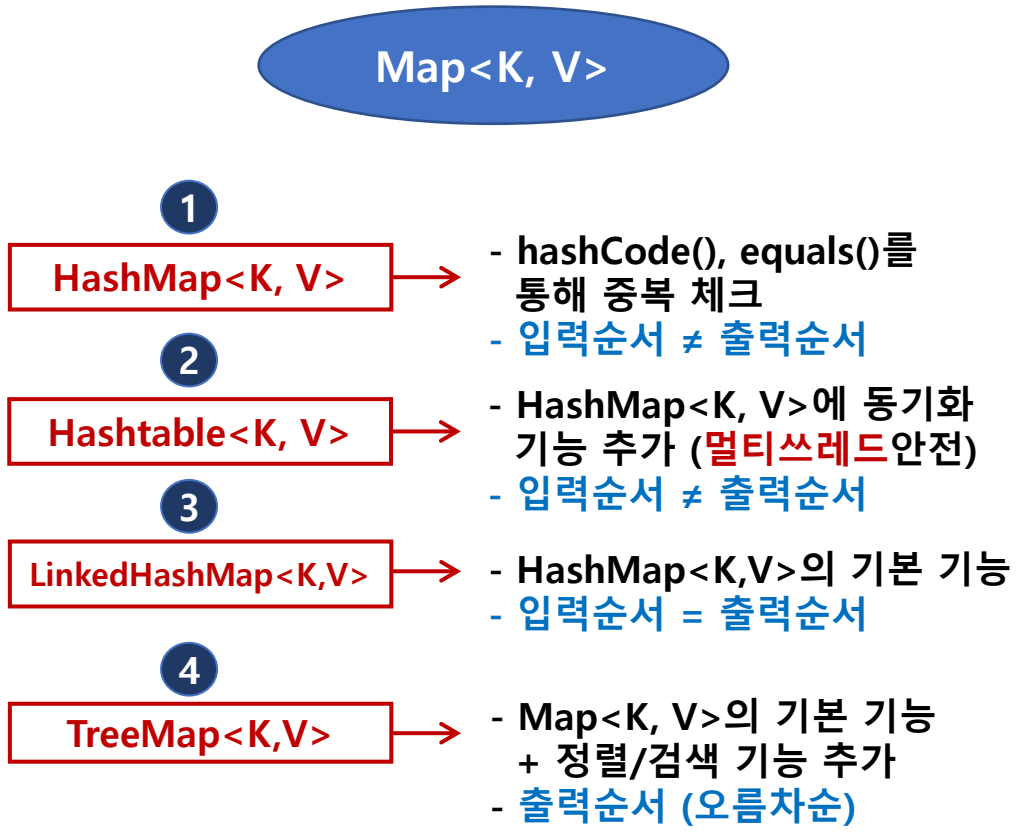
→ myClass1 < myClass2

```
2
3
```

Map<K,V> 컬렉션 – Summary

Map<K, V> 컬렉션

☞ Map<K, V>의 구현객체 정리




```java
//#1. HashMap
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("다", 30);
hashMap.put("마", 40);
hashMap.put("나", 50);
hashMap.put("가", 60);
System.out.println(hashMap.toString());//{가=60,다=30,마=40,나=50}

//#2. HashTable
Map<String, Integer> hashtable = new Hashtable<>();
hashtable.put("다", 30);
hashtable.put("마", 40);
hashtable.put("나", 50);
hashtable.put("가", 60);
System.out.println(hashtable.toString());//{가=60,나=50,마=40,다=30}
```

5

6

```
<terminated> HashMapHashtableLinkedHashMa
{가=60, 다=30, 마=40, 나=50}
{가=60, 나=50, 마=40, 다=30}
```

Map<K, V> 컬렉션

☞ Map<K, V>의 구현객체 정리

HashMap<K, V> →
- hashCode(), equals()를 통해 중복 체크
- 입력순서 ≠ 출력순서

Hashtable<K, V> →
- HashMap<K, V>에 동기화 기능 추가 (멀티쓰레드안전)
- 입력순서 ≠ 출력순서

LinkedHashMap<K,V> →
- HashMap<K,V>의 기본 기능
- 입력순서 = 출력순서

TreeMap<K,V> →
- Map<K, V>의 기본 기능 + 정렬/검색 기능 추가
- 출력순서 (오름차순)

```
//#3. LinkedHashMap
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("다", 30);
linkedHashMap.put("마", 40);                                    ①
linkedHashMap.put("나", 50);
linkedHashMap.put("가", 60);
System.out.println(linkedHashMap.toString());
                                        //{다=30, 마=40, 나=50, 가=60}

//#4. TreeMap
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("다", 30);
treeMap.put("마", 40);                                          ②
treeMap.put("나", 50);
treeMap.put("가", 60);
System.out.println(treeMap.toString());
                                        //{가=60, 나=50, 다=30, 마=40}
```

```
<terminated> HashMapHashtableLinkedHashMap
{다=30, 마=40, 나=50, 가=60}
{가=60, 나=50, 다=30, 마=40}
```

The End

Stack<E> 컬렉션

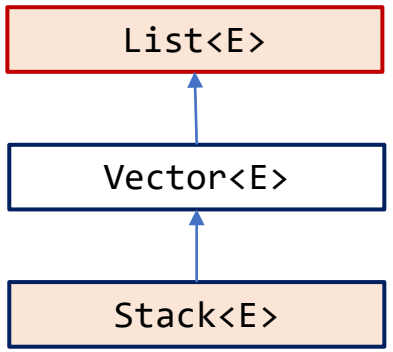
Stack<E> 컬렉션

☞Stack<E> 컬렉션의 특징

1
- Vector<E> 클래스의 자식클래스
- List<E>의 기본 메서드의 사용과 더불어 **LIFO(Last in First Out)** 구조
- Vector<E> 클래스의 기본 메서드와 더불어 LIFO 구조를 위한 **5개의 메서드 추가 정의**

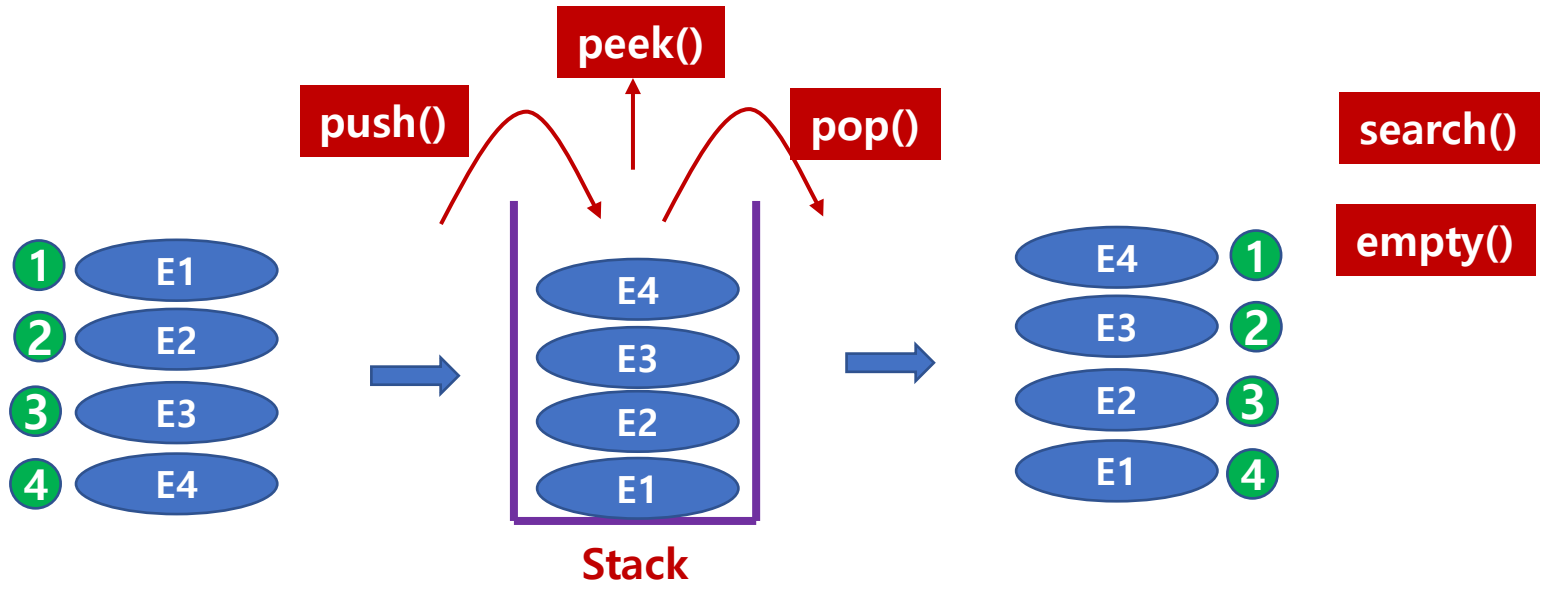
이들 메서드를 호출하기 위해서는
Stack<E> 타입으로 선언해야 함

2



3 **Stack<E>** = **Vector<E>의 기본기능** + **LIFO 기능추가 (5개 메서드)**

4



Stack<E> 컬렉션

☞ Stack<E> 컬렉션의 주요 메서드

구분	리턴타입	메서드이름	기능
데이터 추가	E	push(E item)	매개변수의 item을 Stack에 추가
데이터 보기	E	peek()	가장 상위에 있는 원소 값 리턴 (데이터는 변화가 없음)
데이터위치검색	int	search(Object o)	Stack 원소들의 위치값을 리턴 (최상위의 위치 1을 기준으로 아래로 내려갈수록 1씩 증가)
데이터꺼내기	E	pop()	최상위 데이터 꺼내기 (데이터 개수 줄어듬)
empty 검사	boolean	empty()	Stack 객체가 비워져 있는지 여부를 리턴

1

Stack<E> 컬렉션

☞ Stack<E> 컬렉션의 주요 메서드

- **데이터 추가/보기/검색**

```
Stack<Integer> stack = new Stack<Integer>();
//#1. E push(E element)
stack.push(2);
stack.push(5);
stack.push(3);
stack.push(7);

//#2. E peek()
System.out.println(stack.peek()); //7
System.out.println(stack.size()); //4
System.out.println();

//#3. int search(Object o)
System.out.println(stack.search(7)); //1
System.out.println(stack.search(3)); //2
System.out.println(stack.search(5)); //3
System.out.println(stack.search(2)); //4
System.out.println(stack.search(9)); //-1
```

1 2 3

```
7
4


1
2
3
4
-1
```

Stack<E> 컬렉션

☞ Stack<E> 컬렉션의 주요 메서드

- 데이터 꺼내기 / empty 검사

1
2

```
//#4. E pop()
System.out.println(stack.pop()); //7
System.out.println(stack.pop()); //3
System.out.println(stack.pop()); //5
System.out.println(stack.pop()); //2
System.out.println();

//#5. boolean empty()
System.out.println(stack.empty()); //true
```

```
7
3
5
2

true
```

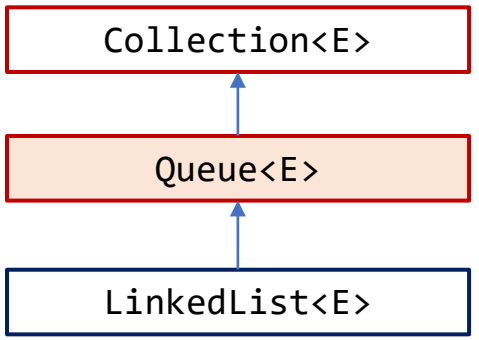
Queue<E> 컬렉션

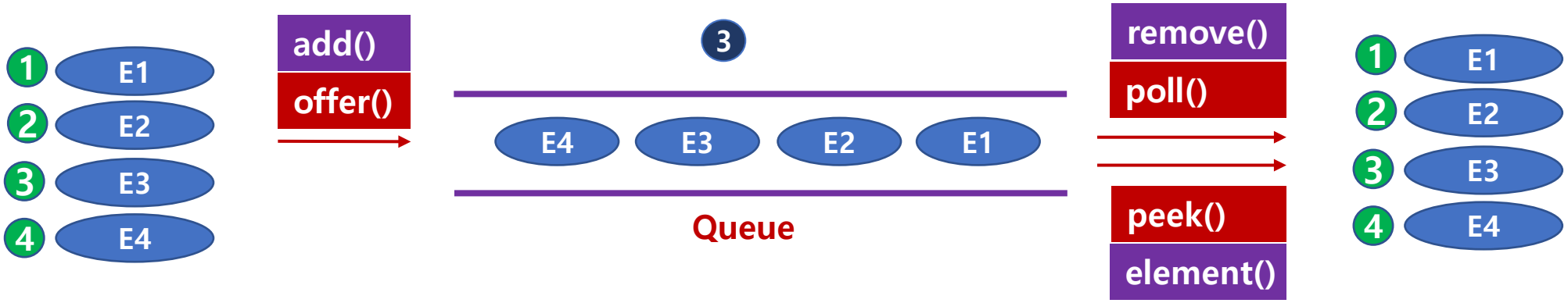
Queue<E> 컬렉션

☞Queue<E> 컬렉션의 특징

1
- Stack<E>과는 달리 별도의 interface로 구성
- 먼저 들어간 데이터가 먼저 나오는 **FIFO(First In First Out) 구조**
- **LinkedList<E> 클래스**는 List<E>와 Queue<E> interface를 구현

2





4

예외처리기능 미포함 메서드	데이터가 없는 경우 예외발생
예외처리기능 포함 메서드	데이터가 없는 경우 기본값 출력

Queue<E> 컬렉션

☞ Queue<E> 컬렉션의 주요 메서드

	구분		리턴타입	메서드이름	기능
1	예외처리 기능 **미포함** 메서드	데이터 추가	boolean	add(E item)	매개변수의 item을 Queue에 추가
		데이터 보기	E	element()	가장 상위에 있는 원소 값 리턴 (데이터는 변화가 없음) (데이터가 하나도 없는 경우 NoSuchElementException 발생)
		데이터 꺼내기	E	remove()	가장 상위에 있는 원소값 꺼내기 (꺼낼 데이터 없는 경우 NoSuchElementException 발생)
2	예외처리 기능 **포함** 메서드	데이터 추가	boolean	offer (E item)	매개변수의 item을 Queue에 추가
		데이터 보기	E	peek()	가장 상위에 있는 원소 값 리턴 (데이터는 변화가 없음) (데이터가 하나도 없는 경우 null을 리턴)
		데이터 꺼내기	E	poll()	가장 상위에 있는 원소값 꺼내기 (꺼낼 데이터 없는 경우 null 발생)

Queue<E> 컬렉션

☞ Queue<E> 컬렉션의 주요 메서드

- **예외처리 기능 미포함 메서드**

```
//#1. 예외처리기능 미포함 메서드
Queue<Integer> queue1 = new LinkedList<Integer>();
//System.out.println(queue.element()); //NoSuchElementException

//@1-1. add(E item)
queue1.add(3);
queue1.add(5);
queue1.add(4);

//@1-2. element()
System.out.println(queue1.element()); //3

//@1-3. remove()
System.out.println(queue1.remove()); //3
System.out.println(queue1.remove()); //5
System.out.println(queue1.remove()); //4
//System.out.println(queue.remove()); //NoSuchElementException
```

```
3

3
5
4
```

Queue<E> 컬렉션

☞ Queue<E> 컬렉션의 주요 메서드

- **예외처리 기능 포함 메서드**

```
//#2. 예외처리기능 포함 메서드
Queue<Integer> queue2 = new LinkedList<Integer>();

System.out.println(queue2.peek()); //null

//@2-1. offer(E item)
queue2.offer(3);
queue2.offer(5);
queue2.offer(4);

//@2-2. peek()
System.out.println(queue2.peek()); //3

//@2-3. poll()
System.out.println(queue2.poll()); //3
System.out.println(queue2.poll()); //5
System.out.println(queue2.poll()); //4
System.out.println(queue2.poll()); //null
```

1
2
3
4

```
null

3

3
5
4
null
```

The End