# 제네릭
# (Generic)

# ▶ 제네릭(Generic)

데이터 타입 어떤게 들어와도 돼
반환자료형이 어떤게 올지 모르지만 어떤게 들어와도 돼
-> 데이터 타입 어떤걸 넘겨주냐에 따라서 알아서 다르게 동작

내부에서 사용할 데이터 타입을 외부에서 지정하는 기법

jdk 1.5 부터 제공되는 기능

클래스와 인터페이스, 메소드를 정의할 때 타입을 파라미터로 사용할 수 있도록 함

사용하기 전에 미리 정의해놓고 사용해야 하는 애들

타입 파라미터는 코드 작성 시 구체적인 타입으로 대체되어 다양한 코드를 생성하도록 해 줌

이 클래스 이용해서 내가 객체 만들게
이 인터페이스 이용해서 참조변수 만들게
호출해서 사용, 정의할 때 사용할 때
정의하는 당시에는 데이터타입 지정x
사용할 때 데이터타입 지정
내가 지정하는 것에 따라서 알아서 동작

입력받는 개수 같고
내용도 같은데
입력받는 자료형이 다를 때

## ✓ 제네릭 프로그래밍

데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있는 기술에 중점을 두어 재사용성을 높일 수 있는 프로그래밍 방식

<위키백과>

# ▶ 제네릭(Generic)

## ✓ 제네릭 장점

1. 컴파일 시 제네릭 코드에 대해 강한 타입 체크를 함
   - 잘못된 타입이 사용될 수 있는 문제를 컴파일 과정에서 제거할 수 있음
2. 컬렉션, 람다식, 스트림, NIO에서 널리 사용됨
3. 불필요한 타입변환을 제거
   - 제네릭 코드로 타입을 지정해 놓으면 해당 타입으로 사용할 때 타입변환을 하지 않아도 됨

일일이 여러개 만들을 필요도 없고
캐스팅이나 타입변환을 하지 않아도 됨

# ▶ 제네릭 타입

타입을 파라미터로 가지는 클래스와 인터페이스

클래스 또는 인터페이스 이름 뒤에 < 타입 > 기호를 추가

타입 파라미터는 변수와 동일한 규칙에 따라 작성

일반적으로 대문자 알파벳 한글자로 표현

## ✓ 제네릭 타입 정의

```
public class 클래스명<T> { ... }

public interface 인터페이스명<T> { ... }
```

# ▶ 제네릭 타입

## ✓ 제네릭 타입 사용

```
클래스명<사용할 타입> 참조변수명 = new 클래스명<사용할 타입>();
클래스명<사용할 타입> 참조변수명 = new 클래스명(); 객체까지 한줄로 생성
```

이 클래스를 이런 타입으로 쓸게

*** 타입변수선언과 객체생성을 동시에 할 때 객체 생성부분의 타입 파라미터는 생략 가능**

# ▶ 제네릭 멀티 타입 파라미터

제네릭 타입은 두 개 이상의 멀티 타입 파라미터 사용 가능

각각 타입 파라미터를 콤마로 구분

타입 파라미터 = 꺽쇄

## ✓ 제네릭 멀티 타입 정의

```
public class 클래스명<T,M> { ... }

public interface 인터페이스명<T,M> { ... }
```

## ✓ 제네릭 멀티 타입 사용

```
클래스명<사용할타입1,사용할타입2> 참조변수명 = new 클래스명();
```

# ▶ 제네릭 메소드



매개타입과 리턴 타입으로 타입 파라미터를 갖는 메소드

리턴 타입 앞에 < 타입 > 기호를 추가

## ✓ 제네릭 메소드 정의

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

## ✓ 제네릭 메소드 사용

```
<사용할 타입>메소드명(인자값);

메소드명(인자값);
```

* 사용할 타입을 생략하면 인자값의 타입을 보고 타입을 추정함

# ▶ 제네릭 제한된 타입 파라미터

타입 파라미터로 지정되는 타입을 제한하기 위한 용도로 사용

타입 파라미터 뒤에 extends 키워드를 붙이고 상위타입을 명시

상위타입은 클래스와 인터페이스도 가능(모두 extends 키워드 사용)

타입 파라미터 변수로 상위타입의 멤버(필드, 메소드)만 사용 가능

## ✓ 제네릭 제한된 타입 정의

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

상위타입 & 상위타입 아래 애들만 가능

Animal 이라고 적으면 Animal을 상속받는 자식들만 가능

우리가 자료형을 실제 사용하는 놈아 니가 자료 저장해라 하니까 아무거나 저장함

내가 원하는 타입말고 다른 타입들까지 막 입력할 수 있음

-> 기능을 구현하는데 문제가 생길 수 있음

이런것들을 막기위해 생긴 제한된 타입 파라미터

# ▶ 제네릭 와일드 카드 타입 파라미터

KH정보교육원

코드에서 ?를 일반적으로 와일드 카드라고 부름
제네릭 타입을 매개변수나 리턴 타입으로 사용할 때 구체적인 타입대신 와일드 카드를 사용 가능

## ✓ 제네릭 와일드 카드 타입 사용 범위

제네릭 타입 **<?>** : 제한없음
(모든 클래스나 인터페이스 타입 사용 가능)
제네릭 타입 **<? extends 상위타입>** : 상위클래스 제한
(명시된 상위타입이나 하위타입만 사용 가능)
제네릭 타입 **<? super 하위타입>** : 하위클래스 제한
(명시된 하위타입이나 상위타입만 사용 가능)

# ▶ 제네릭 타입의 상속과 구현

제네릭 타입의 클래스나 인터페이스를 상속 받을 경우 자식 클래스도 제네릭 타입으로 정의

부모가 제네릭 타입이면 자식도 제네릭타입인게 좋다

자식 제네릭 타입은 부모타입 외 추가로 타입 지정 가능

public class Tiger<T,A> extends Animal<T>{

## ✓ 사용 예시

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

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

# 컬렉션
# (Collection)

# ▶ 컬렉션(Collection)

메모리상에서 자료를 구조적으로 처리하는 방법을 자료구조라 일컫는데

컬렉션(Collection)은 자바에서 제공하는 자료구조를 담당하는 프레임워크

데이터를 사용하기 위한 방법
어떻게 저장하고, 어떻게 접근해서 사용할것이고,
어떻게 수정하고 삭제하고 변경할 것인지 정리해놓은 것

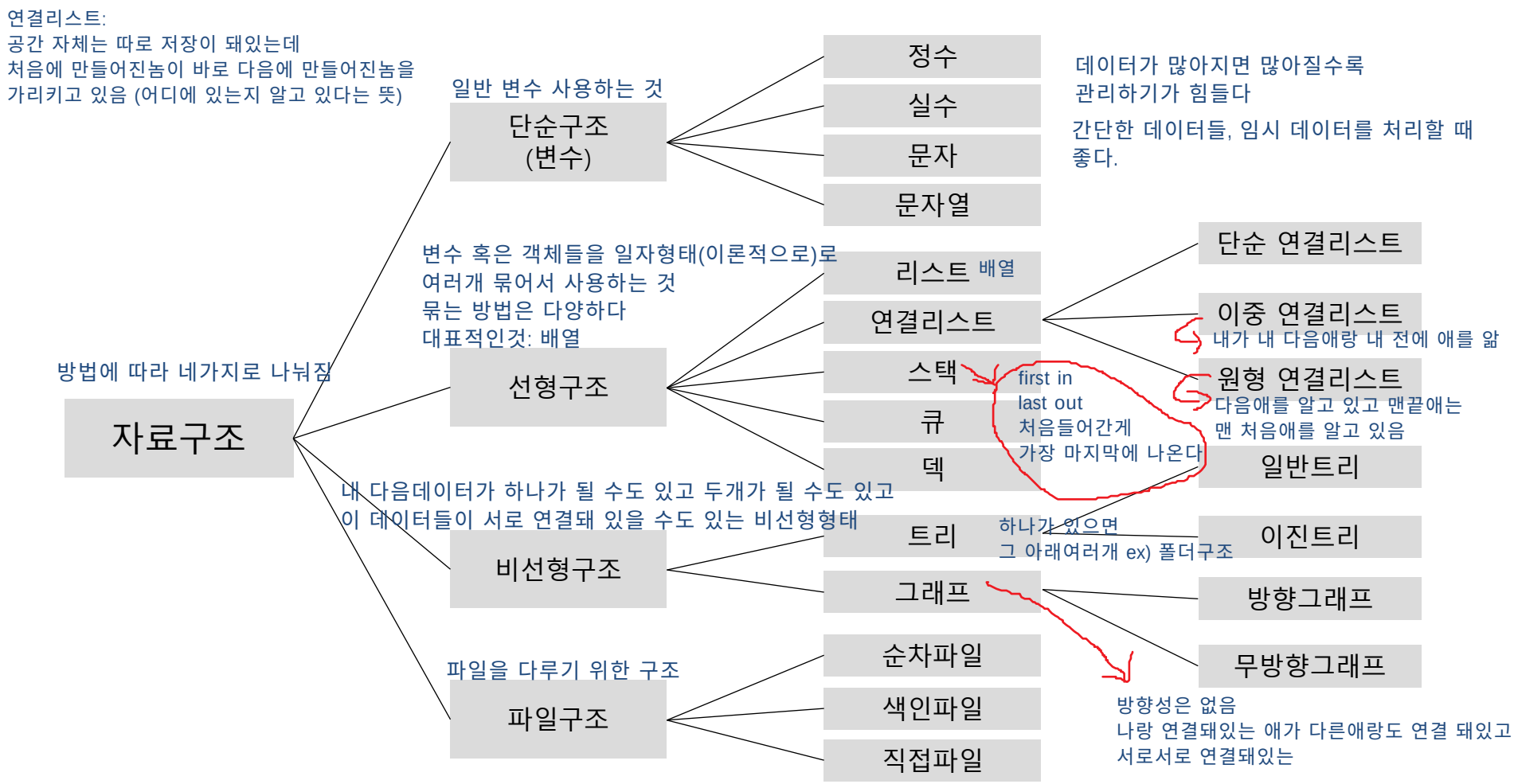
추가, 삭제, 정렬 등의 기능처리가 간단하게 해결 되어 자료구조적 알고리즘을 구현할 필요 없음

java.util 패키지에 포함되며, 인터페이스를 통해 정형화된 방법으로

다양한 컬렉션 클래스 이용 가능

# 자료구조

데이터(자료)를 메모리에서 구조적으로 처리하는 방법론



# ▶ 배열의 문제점 & 컬렉션의 장점

KH정보교육원

## ✓ 배열의 문제점

1. **한 번 크기를 지정하면 변경할 수 없다.**
   필요에 따라 공간을 늘리거나 줄일 수 없음
   공간 크기가 부족하면 에러가 발생
   → 할당 시 넉넉한 크기로 할당하게 됨 (메모리 낭비)

2. **기록된 데이터에 대한 중간 위치의 추가, 삭제가 불편하다.**
   추가, 삭제할 데이터부터 마지막 기록된 데이터까지 하나씩
   뒤로 밀어내고 추가해야 함 (복잡한 알고리즘)

3. **한 타입의 데이터만 저장 가능하다.**

# ▶ 배열의 문제점 & 컬렉션의 장점

## ✓ 컬렉션의 장점

1. **저장하는 크기의 제약이 없다.**

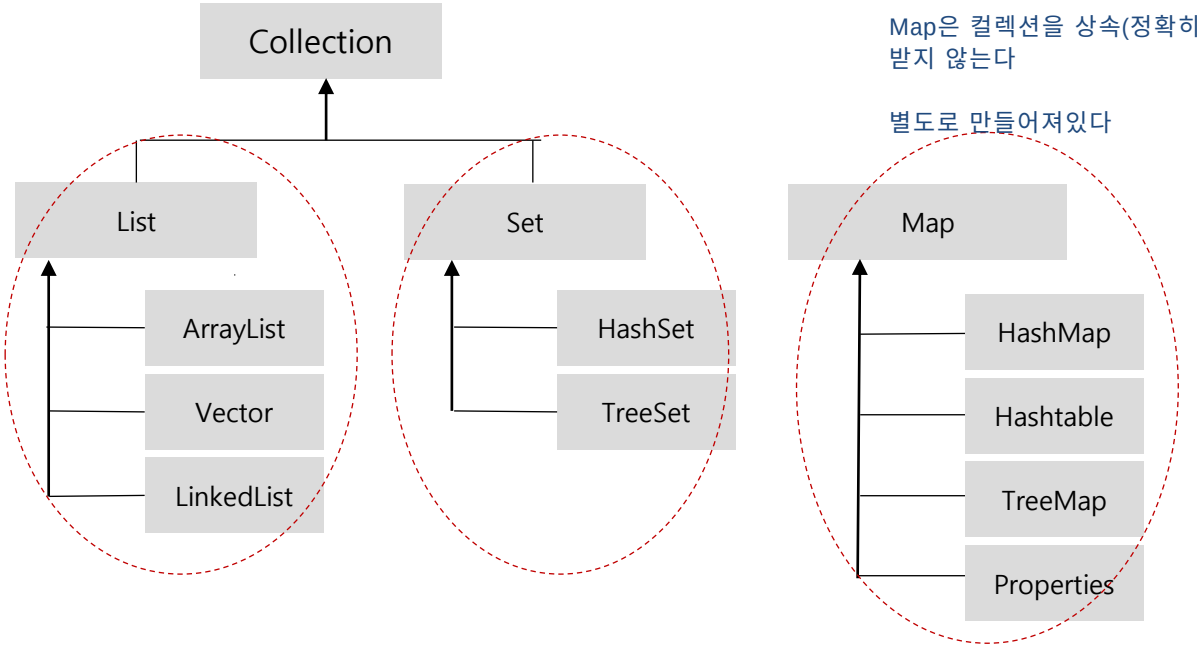
2. **추가, 삭제, 정렬 등의 기능 처리가 간단하게 해결된다.**

   자료를 구조적으로 처리 하는 자료구조가 내장되어 있어
   알고리즘 구현이 필요 없음

3. **여러 타입의 데이터가 저장 가능하다.**

   객체만 저장할 수 있기 때문에 필요에 따라 기본 자료형을
   저장해야 하는 경우 Wrapper클래스 사용

# 컬렉션의 주요 인터페이스




Map은 컬렉션을 상속(정확히는 인터페이스 구현) 받지 않는다

별도로 만들어져있다

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

인덱스를 이용해서 접근

10, 20,30 넣었는데
20, 30,10 으로 나올 수 있다

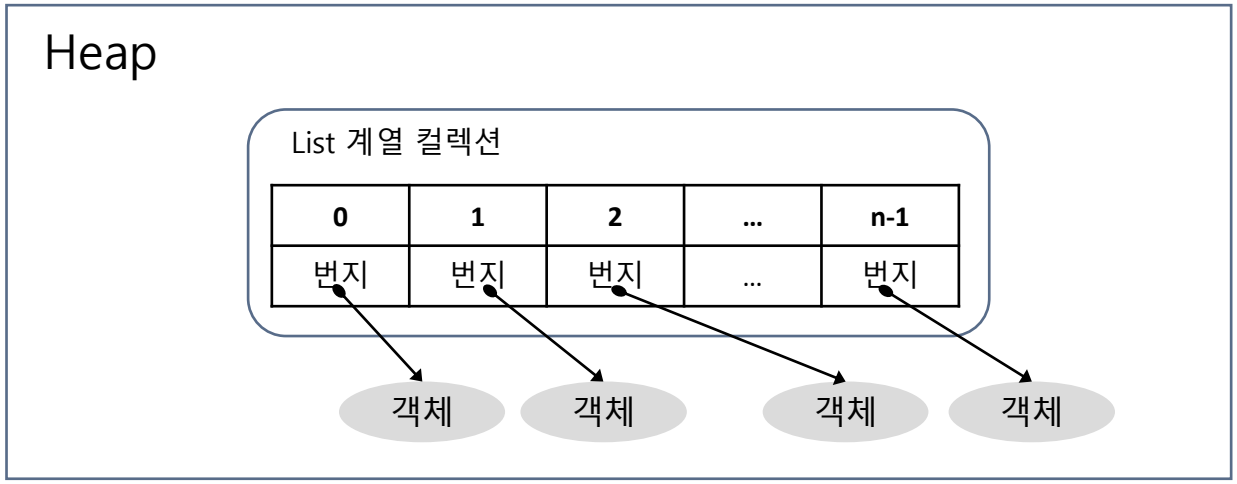
list에 있는거 중복 제거할 때 사용 할 수 있다.

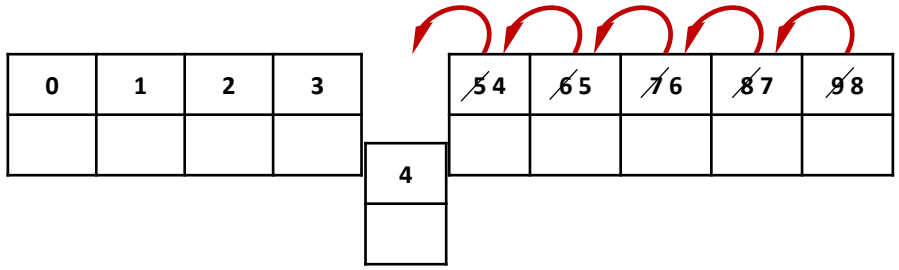
키와 값으로 저장됨
순서는 상관 없음

키 : 값을 찾아가기 위한 이름

# List

자료들을 순차적으로 나열한 자료구조로 인덱스로 관리되며,
중복해서 객체 저장 가능
구현 클래스로 ArrayList와 Vector, LinkedList가 있음



# List

## ✓ List 계열 주요 메소드

| 기능 | 메소드 | 리턴타입 | 설명 |
|---|---|---|---|
| 객체 추가 | add(E e) | boolean | 주어진 객체를 맨 끝에 추가 |
| | add(int index, E element) | void | 주어진 인덱스에 객체를 추가 |
| | addAll(Collection<? extends E> c) | boolean | 주어진 Collection타입 객체를 리스트에 추가 |
| | set(int index, E element) | E | 주어진 인덱스에 저장된 객체를 주어진 객체로 바꿈 |
| 객체 검색 | contains(Object o) | boolean | 주어진 객체가 저장되어 있는지 여부 |
| | get(int index) | E | 주어진 인덱스에 저장된 객체를 리턴 |
| | iterator() | Iterator<E> | 저장된 객체를 한번씩 가져오는 반복자 리턴 |
| | isEmpty() | boolean | 컬렉션이 비어 있는지 조사 |
| | size() | int | 저장되어 있는 전체 객체 수 리턴 |
| 객체 삭제 | clear() | void | 저장된 모든 객체를 삭제 |
| | remove(int index) | E | 주어진 인덱스에 저장된 객체 삭제 특정 데이터 삭제 |
| | remove(Object o) | boolean | 주어진 객체를 삭제 |

# List

## ✓ ArrayList

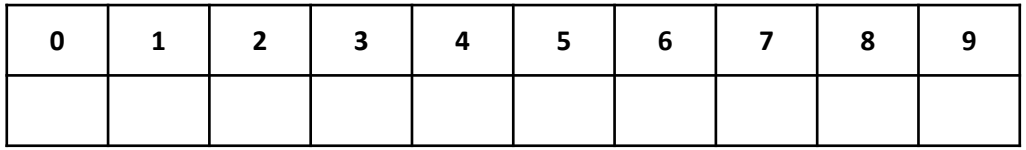
배열의 형태로 리스트를 만든 것

List의 후손으로 초기 저장용량은 10으로 자동 설정 / 따로 지정 가능
저장 용량을 초과한 객체들이 들어오면 자동으로 증가 / 고정도 가능
동기화(Synchronized)를 제공하지 않음

쓰레드에 대해서 안전하지 않다.
백터는 동기화를 제공한다.

예) List<E> list = new ArrayList<E>();

ArrayList




E 타입의 객체 10개를 저장할 수 있는 공간 생성(배열)

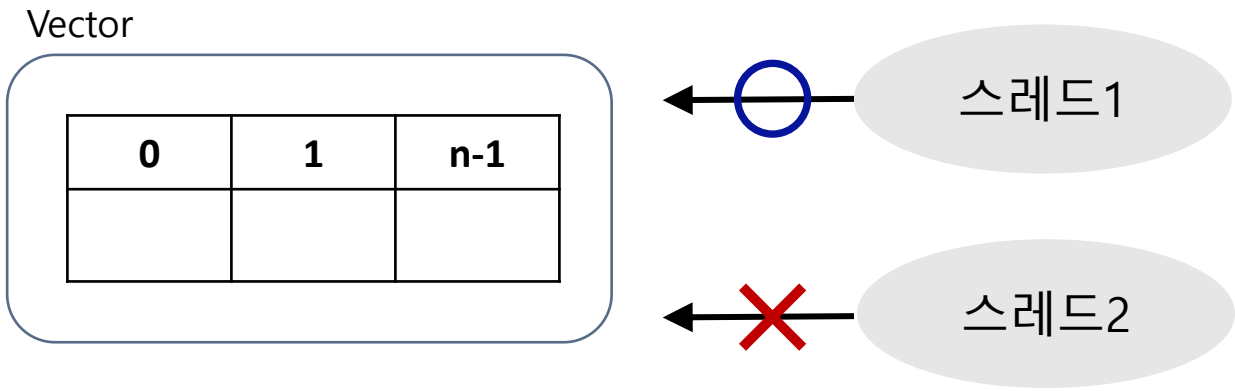
* **동기화 - 하나의 자원(데이터)에 대해 여러 스레드가 접근 하려 할 때 한 시점에서 하나의 스레드만 사용할 수 있도록 하는 것**

# List

## ✓ Vector

List의 후손으로 ArrayList와 동등하지만 동기화(Synchronized)를 제공한다는 점이 ArrayList와의 차이점

→ List 객체들 중에서 가장 성능이 좋지 않음

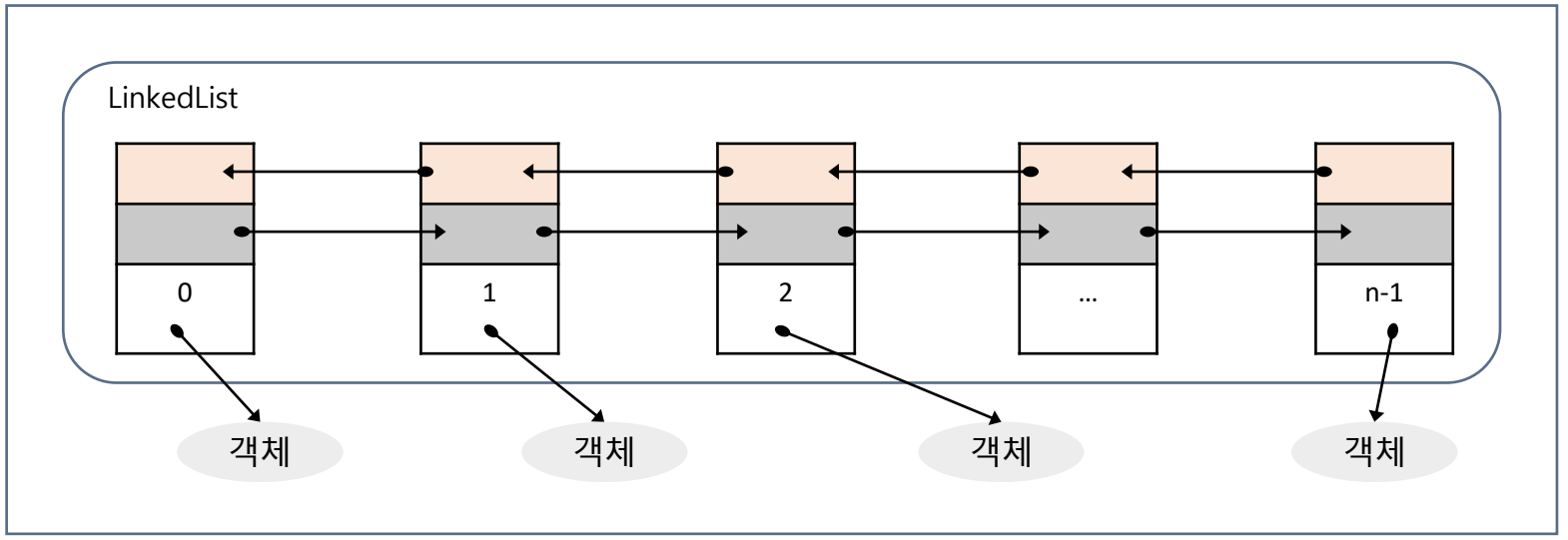


# List

## ✓ LinkedList

List의 후손으로, 인접 참조를 링크해 체인처럼 관리
특정 인덱스에서 객체를 제거하거나 추가하게 되면 바로 앞 / 뒤
링크만 변경하면 되기 때문에 객체 삭제와 삽입이 빈번하게
일어나는 곳에서는 ArrayList보다 성능이 좋음

Heap



# ▶ List

어떤걸 기준으로 정렬할 것인지
기준을 명확하게 정하기

## ✓ Comparable, Comparator

Comparator를 상속 받은
새로운 객체를 생성해서
사용해야 한다

| | Comparable | Comparator |
|---|---|---|
| 패키지 | java.lang | java.util |
| 사용 메소드 | compareTo() 기본 오름차순 기본 내림차순 | compare() |
| 정렬 | 기존의 정렬기준을 구현하는데 사용 | 그외 다른 여러 기준으로 정렬하고자 할 때 사용 |
| 사용법 | 정렬하고자 하는 객체에 Comparable를 상속받아 compareTo() 메소드를 오버라이딩해 기존의 정렬 기준 재정의 어떤 값을 기준으로 정렬할 것인지만 알려주면 됨 → 한 개의 정렬만 가능 | vo 패키지 안에 필요한 정렬 기준에 맞춘 클래스들을 생성하고 Comparator를 상속받아 compare() 메소드를 오버라이딩해 기존의 정렬 기준 재정의 → 여러 개의 정렬 가능 |

## ✓ Collections.sort()

정렬

Collections.sort(List<T> list) → T객체에 **Comparable**을 상속받아 compareTo 메소드 재정의를 통해 정렬 구현 (한 개의 정렬)

오름차순으로

Collections.reverse -> 내림차순으로

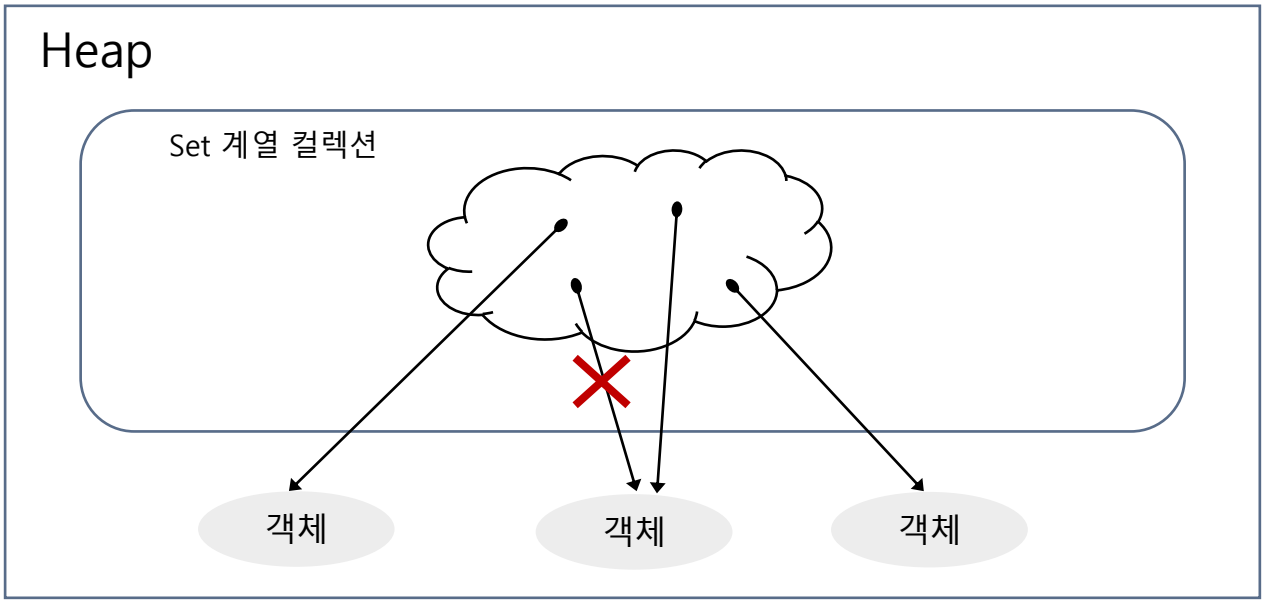
Collections.sort(List<T> list, Comparator<T> c) → 지정한 **Comparator**클래스에 의한 정렬 (여러 개의 정렬)

# ▶ Set

저장 순서가 유지되지 않고, 중복 객체도 저장하지 못하게 하는 자료 구조

null도 중복을 허용하지 않기 때문에 1개의 null만 저장

구현 클래스로 HashSet, LinkedHashSet, TreeSet이 있음



# ▶ Set

set이라는 클래스가 있고
그 안에 여러가지 메소드가 정의 돼있고
HashSet, LinkedHashSet, TreeSet 이런 애들이 상속받아서 사용하는 것

## ✓ Set 계열 주요 메소드

| 기능 | 메소드 | 리턴타입 | 설명 |
|---|---|---|---|
| 객체 추가 | add(E e) | boolean | 주어진 객체를 맨 끝에 추가 |
| | addAll(Collection<? extends E> c) | boolean | 주어진 Collection타입 객체를 리스트에 추가 |
| 객체 검색 | contains(Object o) | boolean | 주어진 객체가 저장되어 있는지 여부 |
| | iterator() | Iterator<E> | 저장된 객체를 한번씩 가져오는 반복자 리턴 |
| | isEmpty() | boolean | 컬렉션이 비어 있는지 조사 |
| | size() | int | 저장되어 있는 전체 객체수를 리턴 |
| 객체 삭제 | clear() | void | 저장된 모든 객체를 삭제 |
| | remove(Object o) | boolean | 주어진 객체를 삭제 |

* 전체 객체 대상으로 한 번씩 반복해서 가져오는 반복자(Iterator)를 제공
  인덱스로 객체에 접근할 수 없음

컬렉션에 있는 데이터를 추출, 꺼내 쓸수 있게 해주는 메소드

# ▶ Set

## ✓ HashSet

Set에 객체를 저장할 때 hash함수를 사용하여 처리 속도가 빠름

동일 객체 뿐 아니라 동등 객체도 중복하여 저장하지 않음

## ✓ LinkedHashSet

HashSet과 거의 동일하지만

Set에 추가되는 순서를 유지한다는 점이 다름

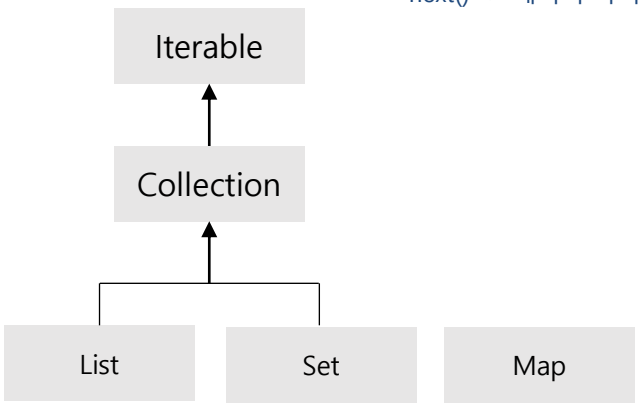
# Enumeration, Iterator, ListIterator

컬렉션에 저장된 요소를 접근하는데 사용되는 인터페이스

- Enumeration : Iterator의 구버전
- ListIterator : Iterator를 상속받아 양방향 특징

[그림 1]의 상속구조 때문에 iterator() 메소드는 List와 Set 계열에서만 사용

→ Map의 경우 Set 또는 List화 시켜서 iterator()를 사용해야 함




[그림1]

hasNext() -> 데이터 있니?
next() -> 데이터 가져와

*** 주요 메소드**

제일 위에 있는 것 부터
하나하나씩 꺼내줌

| | | |
|---|---|---|
| **Iterator<E>** | boolean hasNext() | 앞에서부터 검색 |
| | E next() | |
| **ListIterator<E>** | boolean hasNext() | 앞에서부터 검색 |
| | E next() | |
| | boolean hasPrevious() | 뒤에서부터 검색 |
| | E previous() | |

맨 밑에서 하나씩 꺼내주는 기능도 있다.

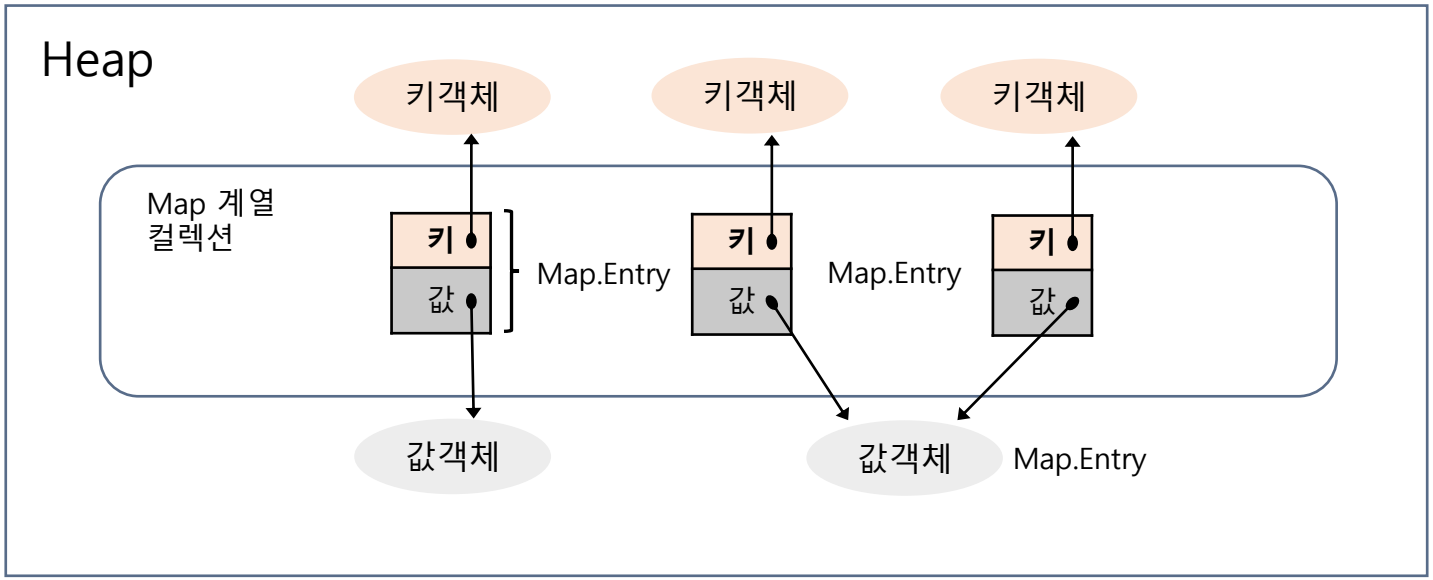
# ▶ Map

키(key)와 값(value)으로 구성되어 있으며, 키와 값은 모두 객체

키는 중복 저장을 허용하지 않고(Set방식), 값은 중복 저장 가능(List방식)

키가 중복되는 경우, 기존에 있는 키에 해당하는 값을 덮어 씌움

구현 클래스로 HashMap, Hashtable, LinkedHashMap, Properties, TreeMap 이 있음



# ▶ Map

## ✓ Map 계열 주요 메소드

| 기능 | 메소드 | 리턴타입 | 설명 |
|---|---|---|---|
| 객체 추가 | put(K key, V value) (키, 값) | V | 주어진 키와 값을 추가, 저장이 되면 값을 리턴 |
| 객체 검색 | containsKey(Object key) | boolean | 주어진 키가 있는지 확인하여 결과 리턴 |
| | containsValue(Object value) | boolean | 주어진 값이 있는지 확인하여 결과 리턴 |
| | entrySet() | Set<Map.Entry <K,V>> | 키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 set에 담아서 리턴 (키&값) |
| | get(Object key) | V | 주어진 키의 값을 리턴 |
| | isEmpty() | boolean | 컬렉션이 비어있는지 여부 |
| | keySet() | Set<K> | 모든 키를 Set 객체에 담아서 리턴 |
| | size() | int | 저장된 키의 총 수를 리턴 |
| | values() | Collection<V> | 저장된 모든 값을 Collection에 담아서 리턴 |
| 객체 삭제 | clear() | void | 모든 Map.Entry를 삭제함 |
| | remove(Object key) | V | 주어진 키와 일치하는 Map.Entry 삭제, 삭제가 되면 값을 리턴한다. |

# ▶ Map

## ✓ HashMap

키 객체는 hashCode()와 equals()를 재정의해 동등 객체가 될 조건을 정해야 함, 때문에 키 타입은 hashCode와 equals()메소드가 재정의되어 있는 String타입을 주로 사용

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

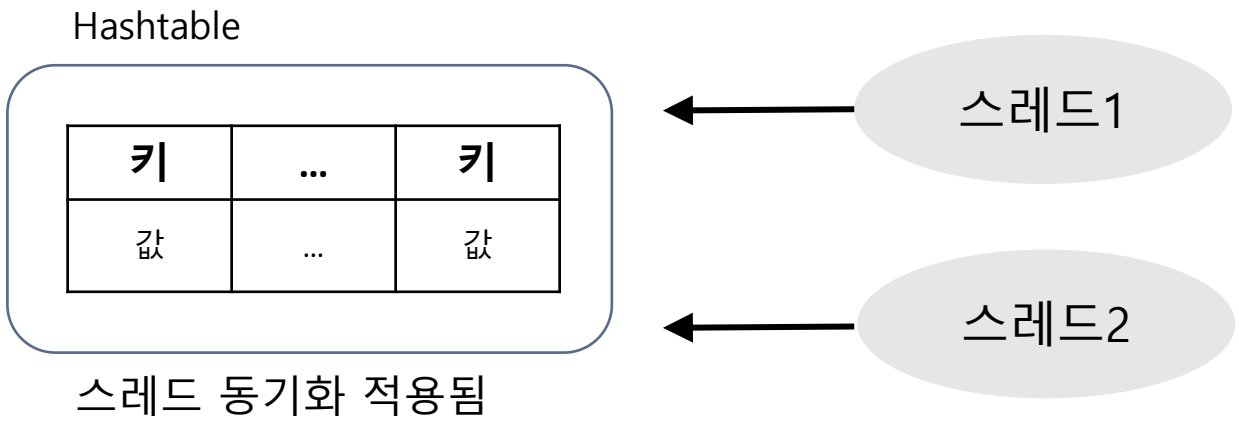
# Hashtable

KH정보교육원

마치 ArrayList와 Vector처럼

키 객체 만드는 법은 HashMap과 동일하나 Hashtable은 스레드 동기화가 된 상태이기 때문에, 복수의 스레드가 동시에 Hashtable에 접근해 객체를 추가, 삭제 하더라도 스레드에 안전 (Thread safe)

예) Map<K, V> map = new HashTable<K, V>();



# ▶ Properties

키와 값을 String타입으로 제한한 Map컬렉션

주로 Properties는 프로퍼티(*.properties)파일을 읽어 들일 때 주로 사용

## ✓ 프로퍼티(*.properties) 파일

- 옵션정보, 데이터베이스 연결정보, 국제화(다국어)정보를 기록하여 텍스트 파일로 활용
- 애플리케이션에서 주로 변경이 잦은 문자열을 저장하여 관리하기 때문에 유지보수를 편리하게 만들어 줌
- 키와 값이 '='기호로 연결되어 있는 텍스트 파일로 ISO 8859-1 문자셋으로 저장되고, 한글은 유니코드(Unicode)로 변환되어 저장

# TreeSet과 TreeMap

비선형 구조 중 하나

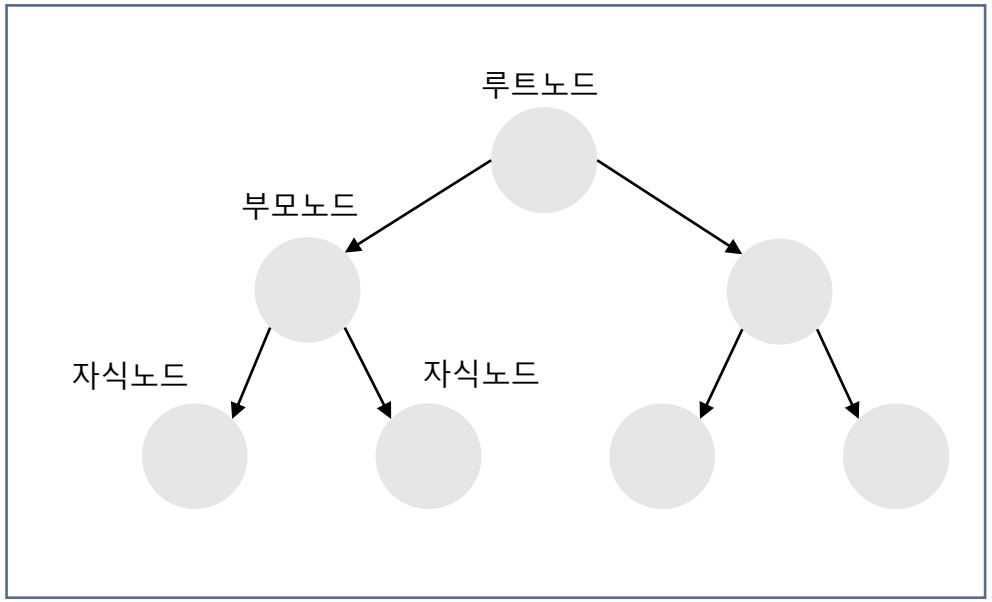
KH정보교육원

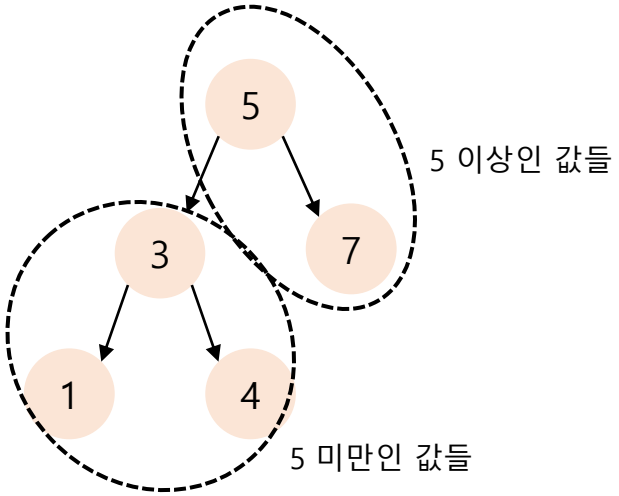
검색 기능을 강화시킨 컬렉션으로, 계층 구조를 활용해 이진 트리 자료구조를 구현하여 제공

* 트리 : 각 노드 간 연결된 모양이 나무와 같다고 해서 붙여진 이름

## Tree 구조

하나하나의 데이터를 노드라고 부른다.



# TreeSet

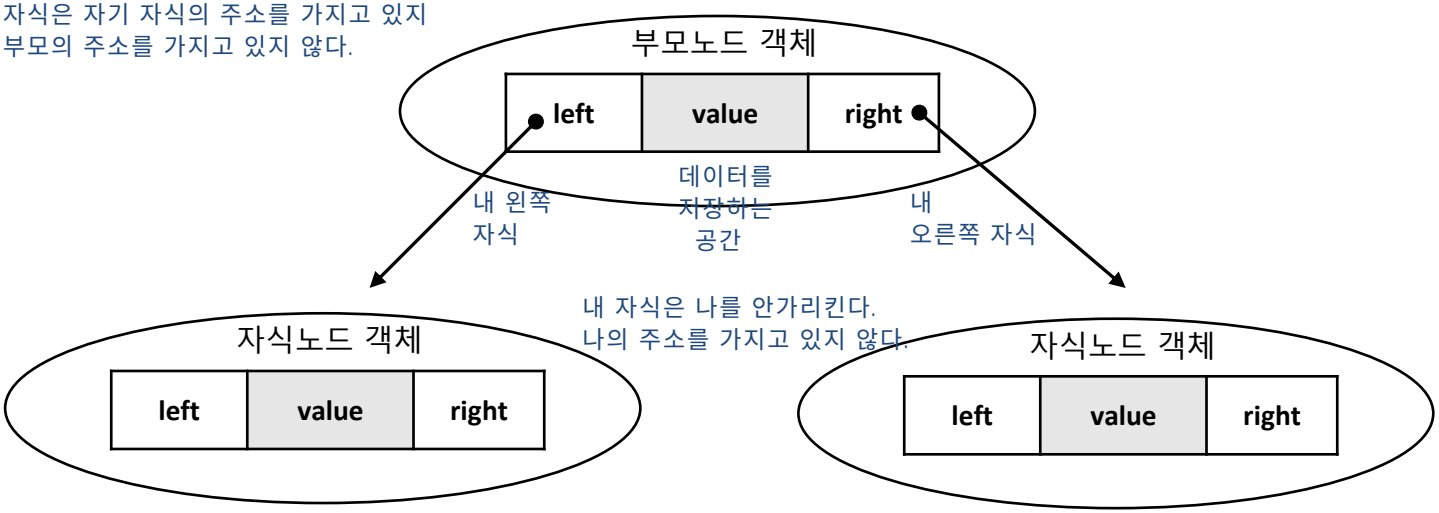
자기 자식이 최대 2명까지 올 수 있는 트리
(자식이 없을 수도 있다.)

Set을 이용해 트리를 구성
기본적으로 중복데이터 저장 안되고
각각의 공간에 접근하기 위해 인덱스,키 값 사용 안한다.

이진 트리를 기반으로 한 Set컬렉션으로,

왼쪽과 오른쪽 자식 노드를 참조하기 위한 두 개의 변수로 구성

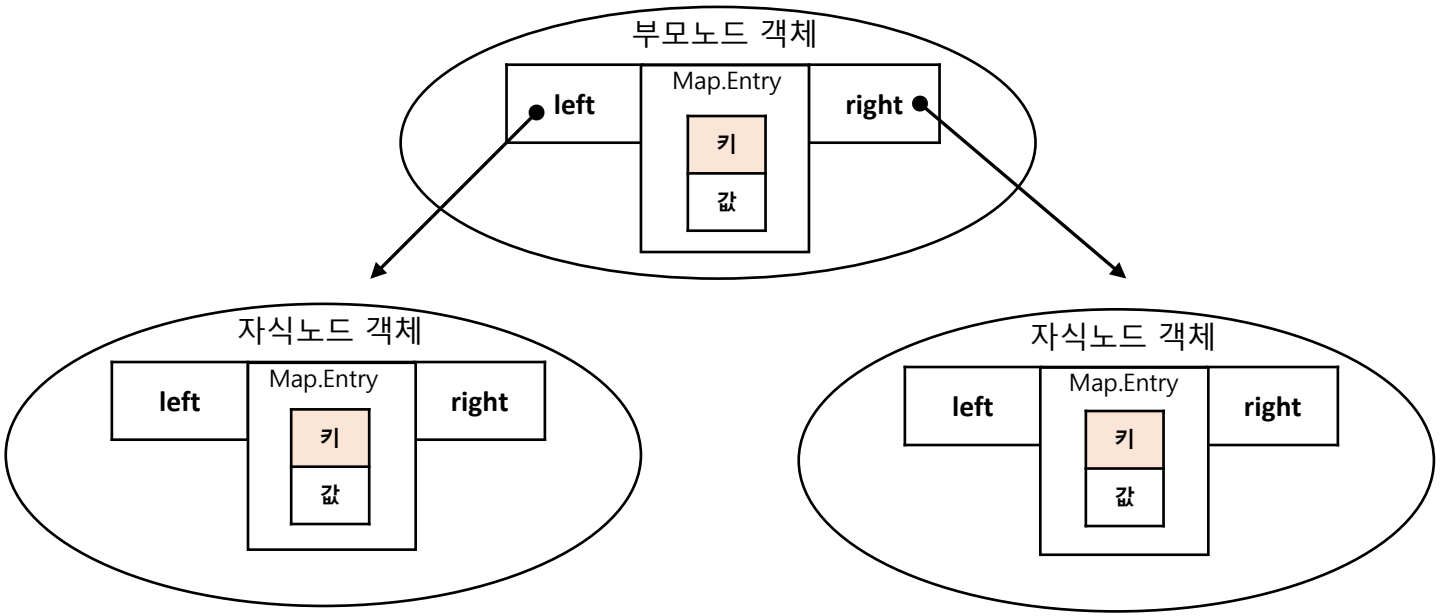
더블 링크드 -> 서로의 주소를 가리키고 있음
TreeSet -> 부모는 자식의 주소를 가지고 있지만
자식은 자기 자식의 주소를 가지고 있지
부모의 주소를 가지고 있지 않다.



# TreeMap

Map을 트리 형태로
Map의 특징은 키와 값을 이용해서 데이터를 저장

이진 트리를 기반으로 한 Map 컬렉션으로, 키와 값이 저장된 Map.Entry를 저장하고 왼쪽과 오른쪽 자식 노드를 참조하기 위한 두 개의 변수로 구성



# ▶ TreeSet, TreeMap 정렬

따로 정렬할게 해주지 않아도 된다.

## ✓ 오름차순(기본 정렬)

- TreeSet의 객체와 TreeMap의 key는 저장과 동시에 자동으로 오름차순 정렬
- 숫자(Integer, Double) 타입일 경우 값으로 정렬
- 문자열(String) 타입일 경우 유니코드로 정렬
- 정렬을 위해 java.lang.Comparable을 구현한 객체 요구
  그러지 않을 경우 ClassCastException 발생
  (Integer, Double, String 모두 Comparable 인터페이스를 통해 오름차순이 구현되어 있음)

# ▶ TreeSet, TreeMap 정렬

## ✓ 내림차순(따로 구현)

- TreeSet, TreeMap 생성시 매개변수 생성자를 통해 재정렬 가능

ex) TreeSet<E> tSet
= new TreeSet(Comparator<? super E> comparator);

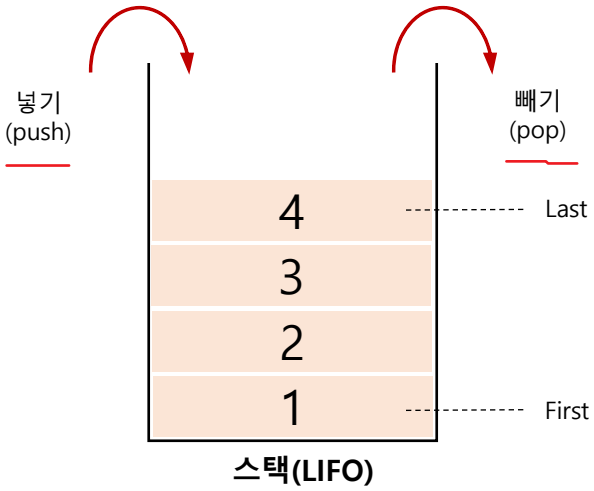
TreeMap<K, V> tMap
= new TreeMap(Comparator<? super K> comparator);

- 또 다른 방법으로 숫자(Integer, Double), 문자열(String) 타입을 제외한 Comparable을 상속 받는 객체인 경우 compareTo() 메소드의 오버라이딩 부분을 내림차순으로 변경

# Stack

후입선출(LIFO : Last In First Out) 구조로 JVM Stack 메모리가 Stack구조로 되어 있음

예) Stack<E> stack = new Stack<E>();



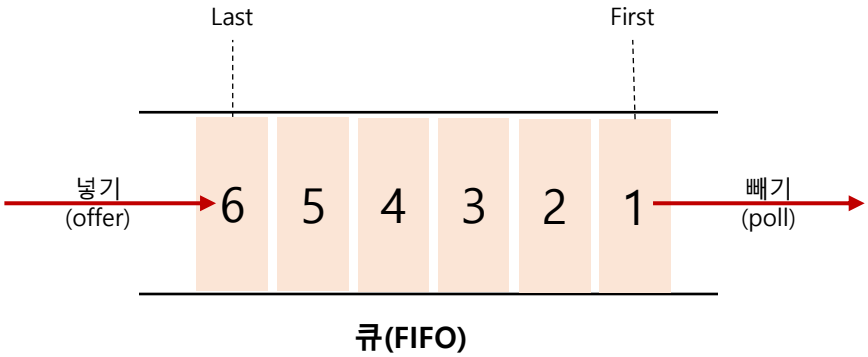

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| E | push(E item) | 주어진 객체를 스택에 넣는다 |
| E | peek() | 스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거하지 않는다. |
| E | pop() | 스택의 맨 위의 객체를 가져온다. 객체를 스택에서 제거한다. |

# Queue

스케쥴링을 위해서 많이 사용하는 구조

선입선출(FIFO : First In First Out) 구조로 작업 큐나 메시지 큐가 Queue구조로 되어 있음

예) Queue() queue = new LinkedList();



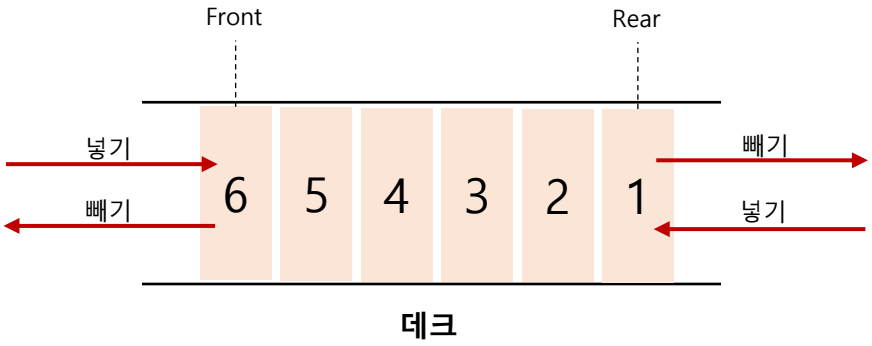

**큐(FIFO)**

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| boolean | offer(E e) | 주어진 객체를 넣는다. |
| E | peek() | 객체를 하나 가져온다. 객체를 큐에서 제거하지 않는다. |
| E | poll() | 객체를 하나 가져온다. 객체를 큐에서 제거한다. |

# Deque

큐랑 똑같이 생겼는데
양쪽으로 넣고 뺄 수 있음

큐와 스택의 성질을 모두 가지고 있는 구조로 검색과 같은 반복적인 문제에 특히 유용한 데이터 구조




데크

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| boolean, void | push, offer, add(E e) | 해당 메소드 뿐만 아니라 메소드 뒤 First, Last를 붙여 앞 뒤에 주어진 객체를 넣는다. |
| E | peek, get() | 해당 메소드 뿐만 아니라 메소드 뒤 First, Last를 붙여 객체를 가져온다. 객체를 큐에서 제거하지 않는다. |
| E | poll, remove() | 해당 메소드 뿐만 아니라 메소드 뒤 First, Last를 붙여 객체를 하나 가져온다. 객체를 큐에서 제거한다. |