JAVA Collections

프로그램은 사람이 이해하는 코드를 작성.

느려도 꾸준하면 경기에서 이긴다.

작성자 : 홍효상
이메일 : hyomee@naver.com
소스 : https://github.com/hyomee/JAVA_EDU

Content

7. Collections

1. Collections
2. List<E>
3. Stack<E>
4. Queue<E>
5. Set<E>
6. Map<E>

1. Collections

" 다수의 데이터를 쉽고 효과적으로 처리할 수 있는 표준화된 방법을 제공하는 클래스의 집합 "

JCF (Java Collections Framework)

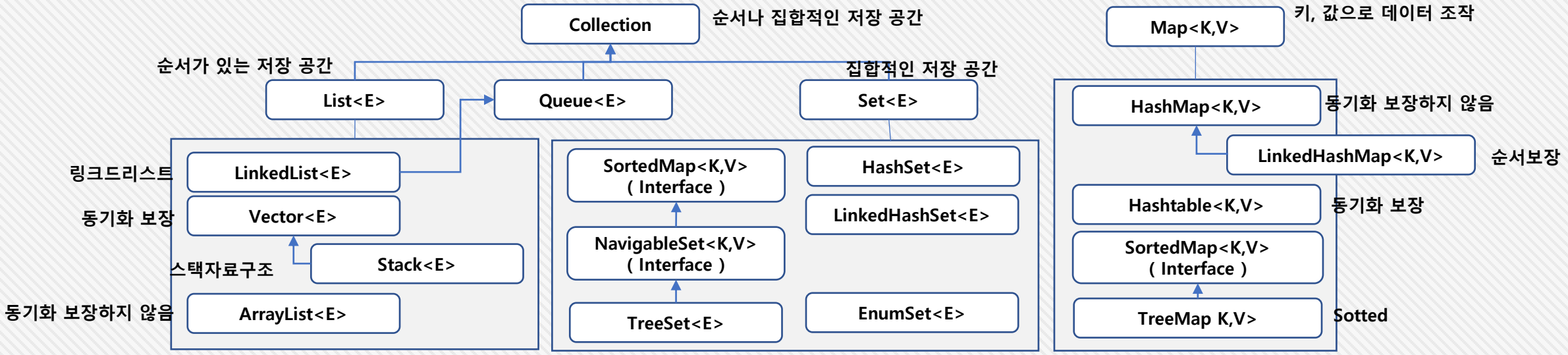
- Collection : 여러 데이터를 모아 놓은 자료 구조 (동일한 자료형)으로 데이터의 크기에 따라서 저장 공간이 동적으로 변환 한다.
- 데이터를 저장하는 자료 구조와 데이터를 처리하는 알고리즘을 구조화하여 클래스로 구현해 놓은 것으로 인터페이스(interface)를 사용하여 구현

➢ Library & Framework

Library : 필요한 기능을 사용 목적에 따라서 만들어 놓은 함수들의 집합으로 재사용을 하기 위한 것
- 내 방식대로 코딩을 하면서 이미 만들어진 기능을 라이브러리로부터 가져 다가 쓰는 것

Framework : Library와 같이 함수들의 집합이지만, 특정한 프로그램 개발에 필요한 필수 함수 및 개발에 대한 방법을 제공 하는 것
- 프레임워크가 정해준 방법을 따라가며 나의 코드를 작성하는 것

JCF 상속 구조



1. Collections

"데이터의 크기에 따라서 저장 공간이 동적으로 변환 "

배열과 차이점

- 배열은 저장 공간의 크기가 고정 되어 있고, Collection은 동적으로 변한다.

```
public static void main(String... args) {
  // 배열과 비교
  DifferenceArray differenceArray = new DifferenceArray();
  differenceArray.arrayMethod();
  differenceArray.collectionMethod();
}
```

```
str 길이 : 4
alist 길이 : 4
alist 길이 : 3
```

```
public class DifferenceArray {

  public void arrayMethod() {
    String[] str = new String[] {"A", "B","C", "D"};
    str[2] = null;
    System.out.println("str 길이 : " + str.length);
  }

  public void collectionMethod() {
    List<String> alist = new ArrayList<>();
    alist.add("A");
    alist.add("B");
    alist.add("C");
    alist.add("D");
    System.out.println("alist 길이 : " + alist.size());
    alist.remove("C");
    System.out.println("alist 길이 : " + alist.size());
  }
}
```

요소에 값을 null로 변경 해도 길이는 변하지 않음

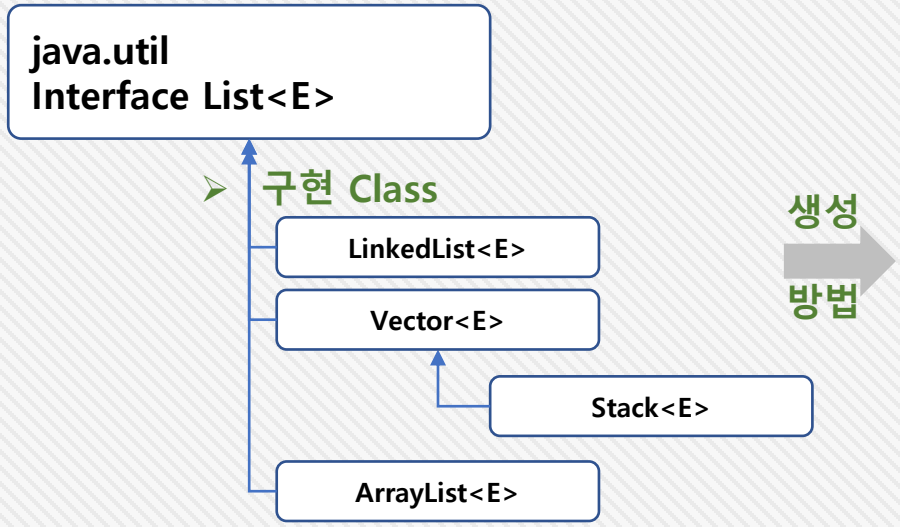
요소에 값을 삭제 하면 크기는 변함

1. List

"순서가 있는 저장 공간으로 배열과 비슷한 자료 구조"

List

- List<E>는 인터페이스이기 때문에 객체를 스스로 생성 할 수 없다.
- 기본 저장 공간은 (int : 10)으로 ( ) 안에 지정을 하지 않으면 적용 된다. – 초기 용량을 지정 해야 할 경우 () 안에 저장 크기를 정수로 기재 (20)
  - private static final int DEFAULT_CAPACITY = 10;
- 기본 저장 공간이 없는 객체 : LinkedList로 초기 저장 공간을 기재 하면 오류 발생



생성 방법

```
List<제너릭타입지정> 참조변수 = new ArrayList<제너릭타입지정>();
List<제너릭타입지정> 참조변수 = new Vector<제너릭타입지정>();
List<제너릭타입지정> 참조변수 = new Stack<제너릭타입지정>();
List<제너릭타입지정> 참조변수 = new LinkedList<제너릭타입지정>();

ArrayList<제너릭타입지정> 참조변수 = new ArrayList<제너릭타입지정>();
Vector<제너릭타입지정> 참조변수 = new Vector<제너릭타입지정>();
Stack<제너릭타입지정> 참조변수 = new Stack<제너릭타입지정>();
LinkedList<제너릭타입지정> 참조변수 = new LinkedList<제너릭타입지정>();

// 정적 Collection 객체 생성 -> new ArrayList<> Wrapping 되어 있음
List<제너릭타입지정> 참조변수 = Arrays.asList(제너릭타입저장데이터);
```

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

1. List


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

// 초기 저장 공간 할당 : 실제 데이터 갯수의 size은 아님
// Capacity 는 생략 가능 하면 생략 이때 기본은 10
List<Integer> asArrayCapacity = new ArrayList<>(20);
List<Integer> asVectors = new Vector<>();
List<Integer> asStack = new Stack<>();

//LinkedList는 Capacity가 없는 객체로 지정시 오류 발생
//List<Integer> asLinkedList = new LinkedList<>(10);
List<Integer> asLinkedList = new LinkedList<>();
```

```
ArrayList<Integer> asArray01 = new ArrayList<>();
Vector<Integer> asVectors01 = new Vector<>();
Stack<Integer> asStack01 = new Stack<>();
LinkedList<Integer> asLinkedList01 = new LinkedList<>();


// 정적 메서드 활용 -> new ArrayList<> Wrapping 되어 있음
List<Integer> asArrays = Arrays.asList(10,20,20,40);
```

2. 주요 메서드

주요 메서드

반환 타입	메서드	설명
boolean	add(E e)	매개변수로 입력된 요소를 마지막에 추가
void	add(int index, E element)	Index 위치에 매개변수(element)를 생성
boolean	addAll(Collection<? extends E> c)	매개변수로 입력된 Collection 전체를 마지막에 추가
boolean	addAll(int index, Collection<? extends E> c)	Index 위치에 매개변수로 입력된 Collection 전체를 생성
void	clear()	전체 삭제
boolean	contains(Object o)	매개변수로 입력한 Object의 포함 되어 있는지 확인
boolean	containsAll(Collection<?> c)	매개변수로 입력한 Collection의 모든 요소가 포함 되어 있는지 확인
boolean	equals(Object o)	매개변수로 입력한 Object가 같은 지 체크
E	get(int index)	Index 위치에 있는 요소 값 반환
int	hashCode()	List에 있는 hashcode 반환
int	indexOf(Object o)	지정된 요소가 처음 나타나는 인덱스를 반환, 요소가 없으면 -1을 반환
boolean	isEmpty()	요소가 없으면 true를 반환
Iterator<E>	iterator()	요소의 순서 반환
int	lastIndexOf(Object o)	지정된 요소가 마지막으로 발생한 인덱스를 반환하거나 이 목록에 요소가 없으면 -1

2. 주요 메서드

주요 메서드

반환 타입	메서드	설명
ListIterator<E>	listIterator()	요소에 대한 목록 순서 반환
ListIterator<E>	listIterator(int index)	지정된 위치에서 시작하여 이 목록의 요소에 대한 목록 반복자를 적절한 순서로 반환
E	remove(int index)	지정된 위치에 있는 요소를 제거
boolean	remove(Object o)	존재하는 경우 지정된 요소의 첫 번째 발생을 제거
boolean	removeAll(Collection<?> c)	지정된 컬렉션에 포함된 모든 요소를 제거
default void	replaceAll(UnaryOperator<E> operator)	각 요소를 해당 요소에 연산자를 적용한 결과로 바꿉니다
boolean	retainAll(Collection<?> c)	지정된 컬렉션에 포함된 이 목록의 요소만 유지
E	set(int index, E element)	Index의 위치에 갓을 요소로 받은 값으로 변경
int	size()	요소 수를 반환
default void	sort(Comparator<? super E> c)	지정된 Comparator에 의해 유도된 순서에 따라 정렬
default Spliterator<E>	spliterator()	요소 분해
List<E>	subList(int fromIndex, int toIndex)	fromIndex 에서 toIndex의 값을 List로 반환
Object[]	toArray()	배열로 반환
<T> T[]	toArray(T[] a)	입력 매개변수로 전달한 타입의 배열로 반환

3. ArrayList<E>

"배열 처럼 인덱스로 관리, 저장 용량 동적 관리"

ArrayList<E>

- List<E> 인터페이스를 구현한 클래스.

```
List<제너릭타입지정> 참조변수 = new ArrayList<제너릭타입지정>();
ArrayList<제너릭타입지정> 참조변수 = new ArrayList<제너릭타입지정>();
List<제너릭타입지정> 참조변수 = Arrays.asList(제너릭타입저장데이터);
```

01. 인스턴스 변수로 ArrayList 선언

```
public class ArrayListTest {
  private final List<String> strList;

  public ArrayListTest() {
    strList = new ArrayList<>();
  }
```

생성자에서 ArrayList 객체 생성

02. 요소 추가 (add, addAll)

예제 : ArrayListTest.addtMethod()

```
// 요소 추가
strList.add("현대자동차");
strList.add("기아자동차");
System.out.println("strList " + strList.toString());

// 특정 요소에 데이터 추가
strList.add(0, "삼성자동차");
System.out.println("strList " + strList.toString());

// 컬렉션을 요소에 추가
List<String> strCollection =  new ArrayList<>();
strCollection.add("포드자동차");
strCollection.add("도시바자동차");
System.out.println("strCollection " + strCollection.toString());
```

마지막 요소에 추가

Index 를 지정 해서 해당 요소에 추가

Collection을 마지막 요소에 추가

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

03. 요소 삭제 (remove, removeAll, clear)

```
// 지정한 위치  요소 삭제
strList.remove(2);
System.out.println("removeMethod : strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 포르쉐자동차, BMW자동차, 포드자동차, 도시바자동차]


List<String> strCollection01 =  new ArrayList<>();
strCollection01.add("포르쉐자동차");
strCollection01.add("BMW자동차");
System.out.println("removeMethod : strCollection01 " + strCollection01.toString());
// 결과 : [포르쉐자동차, BMW자동차]

// Collection 객체에 있는 모든 요소 삭제
strList.removeAll(strCollection01);
System.out.println("removeMethod : removeAll : strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 포드자동차, 도시바자동차]
```

지정한 요소에서 삭제

예제 : ArrayListTest.removeMethod()

Collection 에 있는 요소 삭제

```
strList.add("포드자동차");
System.out.println("removeMethod :  strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 포드자동차, 도시바자동차, 포드자동차]

// 중복되는 요소 값의 첫번째 삭제
strList.remove("포드자동차");
System.out.println("removeMethod :  remove : strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 도시바자동차, 포드자동차]

// 전체 삭제
strList.clear();
System.out.println("removeMethod : strList " + strList.toString());
// 결과 : []
```

중복되는 요소의 첫번째 삭제

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

04. 정보 가져 오기 (size, get, indexof, contains)

예제 : ArrayListTest.getMethod()

List 크기 구하기

```
// 순회 하면서 정보 읽기
int size = strList.size();
for ( int i = 0; i < size; i++) {
  // i 변수는 지역변수로 개발자 실수에 의해서 오염 될 수 있음
  // i = i + 4; // List에 값이 없어서 오류 발생 할수 있음
  // java.lang.IndexOutOfBoundsException: 발생
  System.out.println("getMethod : strList i " + i + " , 정보 : " + strList.get(i));
}
/*
 * 결과 :
 * getMethod : strList i 0 , 정보 : 삼성자동차
 * getMethod : strList i 1 , 정보 : 현대자동차
 * getMethod : strList i 2 , 정보 : 기아자동차
 * getMethod : strList i 3 , 정보 : 포르쉐자동차
 * getMethod : strList i 4 , 정보 : BMW자동차
 * getMethod : strList i 5 , 정보 : 포드자동차
 * getMethod : strList i 6 , 정보 : 도시바자동차
 */
```

지역 변수 오염이 되어서 잘못 된 결과 초래 -> 잠재적 오류

```
// foreach 로 표현
for(String str: strList) {
  System.out.println("getMethod : strList " + str);
}
/*
 * 결과 :
 * getMethod : strList 삼성자동차
 * getMethod : strList 현대자동차
 * getMethod : strList 기아자동차
 * getMethod : strList 포르쉐자동차
 * getMethod : strList BMW자동차
 * getMethod : strList 포드자동차
 * getMethod : strList 도시바자동차
 */
```

단어 위치 찾기

```
// 특정 요소가 있는지 체크
// index는 존재 하면 해당 index를 돌려 주고 없으면 -1
int isExistIndex = strList.indexOf("현대자동차");
if (isExistIndex > 0) {
  System.out.println("현대자동차는 존재 합니다.");
  strList.add(isExistIndex, "현대자동차대신");
}
// 결과 : 현대자동차는 존재 합니다.
```

단어 포함 되어 있는지 체크

```
if (!strList.contains("현대02자동차")) {
  System.out.println("현대자동차는 존재 하지 않습니다.");
}
// 결과 : 현대자동차는 존재 하지 않습니다.
```

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

05. 정렬 하기 (sort)

예제 : ArrayListTest.sortMethod()

```
strList.add("A");
strList.add("Z");
strList.add("a");

// 오름차순으로 정렬
strList.sort(Comparator.naturalOrder());
System.out.println("오름차순 : " + strList);
// 결과 : [A, BMW자동차, Z, a, 기아자동차, 도시바자동차, 삼성자동차, 포드자동차, 포르쉐자동차, 현대자동차, 현대자동차대신]

// 내림차순으로 정렬
strList.sort(Comparator.reverseOrder());
System.out.println("내림차순 : " + strList);
// 결과 : [현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, a, Z, BMW자동차, A]

// 대소문자 구분없이 오름차순 정렬
strList.sort(String.CASE_INSENSITIVE_ORDER);
System.out.println("대소문자 구분없이 오름차순 : " + strList);
// 결과 : [a, A, BMW자동차, Z, 기아자동차, 도시바자동차, 삼성자동차, 포드자동차, 포르쉐자동차, 현대자동차, 현대자동차대신]

// 대소문자 구분없이 내림차순 정렬
strList.sort(Collections.reverseOrder(String.CASE_INSENSITIVE_ORDER));
System.out.println("대소문자 구분없이 내림차순 : " + strList);
// 결과 : [현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, Z, BMW자동차, a, A]
```

오름차순으로 정렬

내림차순으로 정렬

대소문자 구분없이 오름차순 정렬

대소문자 구분없이 내림차순 정렬

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

예제 : ArrayListTest.copyMethod()

06. 복사 하기


```
List<String> strListTemp = strList;
System.out.println("복사 : strListTemp : " + strListTemp);
// 결과 : [현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, Z, BMW자동차, a, A]
strListTemp.add(0, "추가");
```

복사 하기 : shallowCopy : 변수의 주소만 틀리지 실제 참조 하고 있는 주소는 같아서 발생

```
System.out.println("원본 : strList : " + strList);
// 결과 : [추가, 현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, Z, BMW자동차, a, A]
System.out.println("복사 : strListTemp : " + strListTemp);
// 결과 : [추가, 현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, Z, BMW자동차, a, A]
```


```
List<String> strListTemp01 = new ArrayList<>();;
for (String str : strList) {
  strListTemp01.add(str);
}
```

복사 하기 : deepCopy

```
strListTemp01.add(0, "더추가");
System.out.println("원본 : strList : " + strList);
// 결과 : [추가, 현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, Z, BMW자동차, a, A]
System.out.println("복사 : strListTemp01 : " + strListTemp01);
// 결과 : [더추가, 추가, 현대자동차대신, 현대자동차, 포르쉐자동차, 포드자동차, 삼성자동차, 도시바자동차, 기아자동차, Z, BMW자동차, a, A]
```

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

07. 복사 하기 (Class - shallowCopy)

예제 : ArrayListTest.shallowCopyObjectMethod()

```
ArrayList<Customer> customerArrayList = new ArrayList<>();
customerArrayList.add(new Customer("홍길동", 18));
customerArrayList.add(new Customer("홍당무", 20));
ArrayList<Customer> copyOfcustomerArrayList = (ArrayList<Customer>) customerArrayList.clone();
System.out.println("원본 : customerArrayList : " + customerArrayList);
// 결과 : [Customer{name='홍길동', age=18}, Customer{name='홍당무', age=20}]
System.out.println("복사 : copyOfcustomerArrayList : " + copyOfcustomerArrayList);
// 결과 : [Customer{name='홍길동', age=18}, Customer{name='홍당무', age=20}]

Customer customer = copyOfcustomerArrayList.get(0);
customer.setName("김순길");
customer.setAge(25);
System.out.println("원본 : customerArrayList : " + customerArrayList);
// 결과 : [Customer{name='김순길', age=25}, Customer{name='홍당무', age=20}]
System.out.println("복사 : copyOfcustomerArrayList : " + copyOfcustomerArrayList);
// 결과 : [Customer{name='김순길', age=25}, Customer{name='홍당무', age=20}]
```

```
class Customer {

  private String name;
  private int age;

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

  public void setName(String name) {
    this.name = name;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @Override
  public String toString() {
    return "Customer{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
  }
}
```

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

08. 복사 하기 (Class - deepCopy)

예제 : ArrayListTest.deepCopyObjectMethod()

```
ArrayList<CustomerClone> customerArrayList = new ArrayList<>();
customerArrayList.add(new CustomerClone("홍길동", 18));
customerArrayList.add(new CustomerClone("홍당무", 20));
ArrayList<CustomerClone> copyOfcustomerArrayList = new ArrayList<>();
for ( CustomerClone customerClone : customerArrayList) {
  copyOfcustomerArrayList.add(customerClone.clone());
}

System.out.println("원본 : customerArrayList : " + customerArrayList);
// 결과 : [Customer{name='홍길동', age=18}, Customer{name='홍당무', age=20}]
System.out.println("복사 : copyOfcustomerArrayList : " + copyOfcustomerArrayList);
// 결과 : [Customer{name='홍길동', age=18}, Customer{name='홍당무', age=20}]

CustomerClone customerClone = copyOfcustomerArrayList.get(0);
customerClone.setName("김순길");
customerClone.setAge(25);

System.out.println("원본 : customerArrayList : " + customerArrayList);
// 결과 : [Customer{name='홍길동', age=18}, Customer{name='홍당무', age=20}]
System.out.println("복사 : copyOfcustomerArrayList : " + copyOfcustomerArrayList);
// 결과 : [Customer{name='김순길', age=25}, Customer{name='홍당무', age=20}]
```

```
class CustomerClone implements Cloneable {
  private String name;
  private int age;

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

  public void setName(String name) {
    this.name = name;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @Override
  public String toString() {
    return "Customer{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
  }

  @Override
  public CustomerClone clone() {
    try {
      CustomerClone clone = (CustomerClone) super.clone();
      // TODO: copy mutable state here, so the clone can't change the internals of the original
      return clone;
    } catch (CloneNotSupportedException e) {
      throw new AssertionError();
    }
  }
}
```

Cloneable 상속

clone 재 정의 해야 함

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

3. ArrayList<E>

09. 배열 변환

예제 : ArrayListTest. toArray()

```
List<String> strCars =  new ArrayList<>();
strCars.add("포르쉐자동차");
strCars.add("BMW자동차");

Object[] ob = strCars.toArray();
System.out.println("strCars.toArray : " + Arrays.toString(ob));
// 결과 : [포르쉐자동차, BMW자동차]

String[] strings01 = strCars.toArray(new String[0]);
System.out.println("strCars.toArray(new String[0] : " + Arrays.toString(strings01));
// 결과 : [포르쉐자동차, BMW자동차]

String[] strings02 = strCars.toArray(new String[4]);
System.out.println("strCars.toArray(new String[4] : " + Arrays.toString(strings02));
// 결과 : [포르쉐자동차, BMW자동차, null, null]
```

배열로 변환 시 원본 보다 적으면 원본 크기로 전환

배열로 변환 시 원본 보다 크게 지정 하면 null 로 들어감

예제 : https://github.com/hyomee/JAVA_EDU/blob/main/Collection/src/com/hyomee/collection/arrayList/ArrayListTest.java

4. Vector<E>

"List의 공통적인 특성을 가지고 있으면서 동기화 Method"

Vector<E>

- List<E> 인터페이스를 구현한 클래스로 ArrayList<E>와 기능이 동일
- 동기화 메서드(Synchronized Method)는 하나의 공유 객체를 2개의 쓰레드에서 동시에 사용 할 수 없도록 만든 메서드
- 멀티 쓰레드에서 적합함

```
List<제너릭타입지정> 참조변수 = new Vector<제너릭타입지정>();
Vector<제너릭타입지정> 참조변수 = new Vector<제너릭타입지정>();
```

01. 인스턴스 변수로 Vector 선언

```
private final List<String> strList;

public VectorTest() {
  strList = new Vector<>();
}
```

02. 요소 추가 (add, addAll)

```
// 특정 요소에 데이터 추가
strList.add(0, "삼성자동차");
System.out.println("strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 기아자동차]

// 컬렉션을 요소에 추가
List<String> strCollection =  new Vector<>();
strCollection.add("포드자동차");
strCollection.add("도시바자동차");
System.out.println("strCollection " + strCollection.toString());
// 결과 : [포드자동차, 도시바자동차]
```

5. LinkedList<E>

"List의 공통적인 특성을 가지고 있으면서 저장용량을 지정 할 수 없다"

LinkedList<E>

- List<E> 인터페이스를 구현한 클래스로 ArrayList<E>와 기능이 동일
- ArrayList는 요소의 index값으로 저장 하다면 LinkedList는 요소가 서로 연결 되어 있디.
- 싱글 쓰레드에서 적합함

```
List<제너릭타입지정> 참조변수 = new LinkedList<제너릭타입지정>();
LinkedList<제너릭타입지정> 참조변수 = new LinkedList<제너릭타입지정>();
```

➢ **ArrayList**

성능 빠름 →

➢ **LinkedList**

0	1	2	3
온	달	편	강

△
과 (삽입)

요소를 밀면서 진행

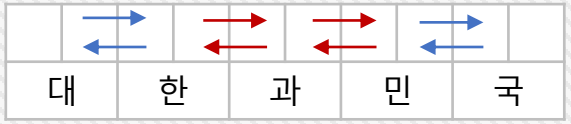
0	1	2	2	3
대	한	과	민	국

0	1	2	3	3
대	한	과	민	국

0	1	2	3	4
대	한	과	민	국

△
과 (삽입)

Link만 변경



5. LinkedList<E>

01. 인스턴스 변수로 LinkedList 선언

```
private final List<String> strList;

public LinkedListTest() {
  strList = new LinkedList<>();
}
```

02. 요소 추가 (add, addAll)

```
// 요소 추가
strList.add("현대자동차");
strList.add("기아자동차");

System.out.println("strList " + strList.toString());
// 결과 : [현대자동차, 기아자동차]

// 특정 요소에 데이터 추가
strList.add(0, "삼성자동차");
System.out.println("strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 기아자동차]

// 컬렉션을 요소에 추가
List<String> strCollection =  new LinkedList<>();
strCollection.add("포드자동차");
strCollection.add("도시바자동차");
System.out.println("strCollection " + strCollection.toString());
// 결과 : [포드자동차, 도시바자동차]

// 컬렉션을 마지막 요소에 추가
strList.addAll(strCollection);
System.out.println("strList " + strList.toString());
// 결과 : [삼성자동차, 현대자동차, 기아자동차, 포드자동차, 도시바자동차]
```

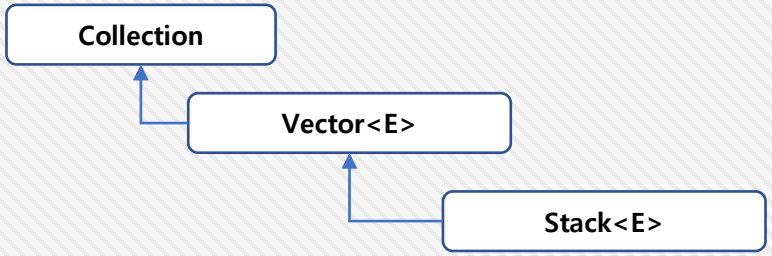
1. Stack<E>

"LIFO(Last in First out)로 Vector<E>의 모든 기능 + 추가 메서드"

Stack<E>

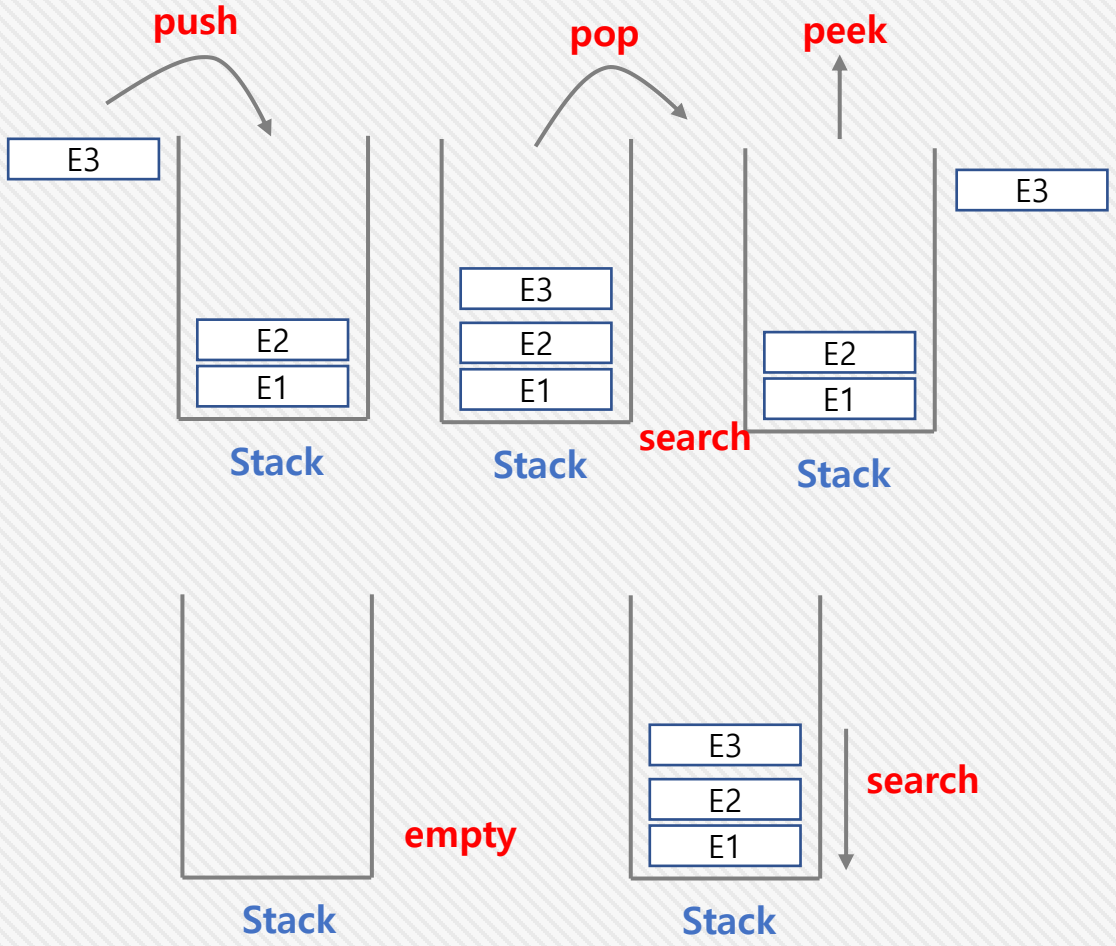
- Vector<E> Class를 상속 받았으므로 Vector의 모든 기능 사용

```
Stack<제너릭타입지정> 참조변수 = new Stack<제너릭타입지정>();
```



주요 메서드

메서드	설명
E push(E item)	매개변수로 입력된 요소를 추가 (데이터 증가)
synchronized E pop()	가장 사위에 있는 요소 꺼냄 (데이터 감소)
synchronized E peek()	가장 상위에 있는 요소 리턴 (데이터는 변화 없음)
boolean empty()	비어 있는지 확인
synchronized int search(Object o)	요소의 위치 값 리턴 (가장 상위 1, 아래로 내려 갈 수록 1증가



1. Stack<E>


```
public class StackTestMain {
  public static void main(String... args) {
    Stack<Integer> stack = new Stack<>();
    System.out.println(stack.empty());          // true
    // STACK에 넣기
    System.out.println(stack.push(10));     // 10
    System.out.println(stack.push(20));     // 20
    System.out.println(stack.push(30));     // 30
    System.out.println(stack.toString());       // [10, 20, 30]
    System.out.println(stack.peek());           // 30
    System.out.println(stack.search(10));     // 1
    System.out.println(stack.empty());          // false
    System.out.println(stack.pop());            // 30
    System.out.println(stack.pop());            // 20
    System.out.println(stack.pop());            // 10
    System.out.println(stack.empty());          // true
  }
}
```

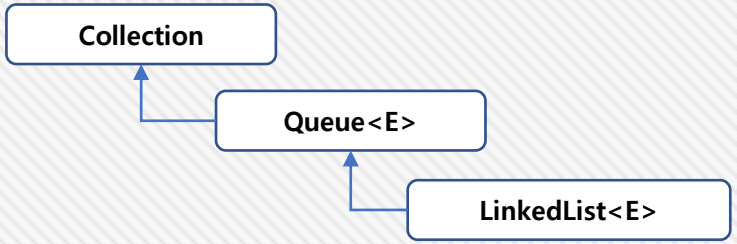
1. Queue<E>

" LinkedList<E>가 Queue<E> 인터페이스의 구현체 "

Queue<E>

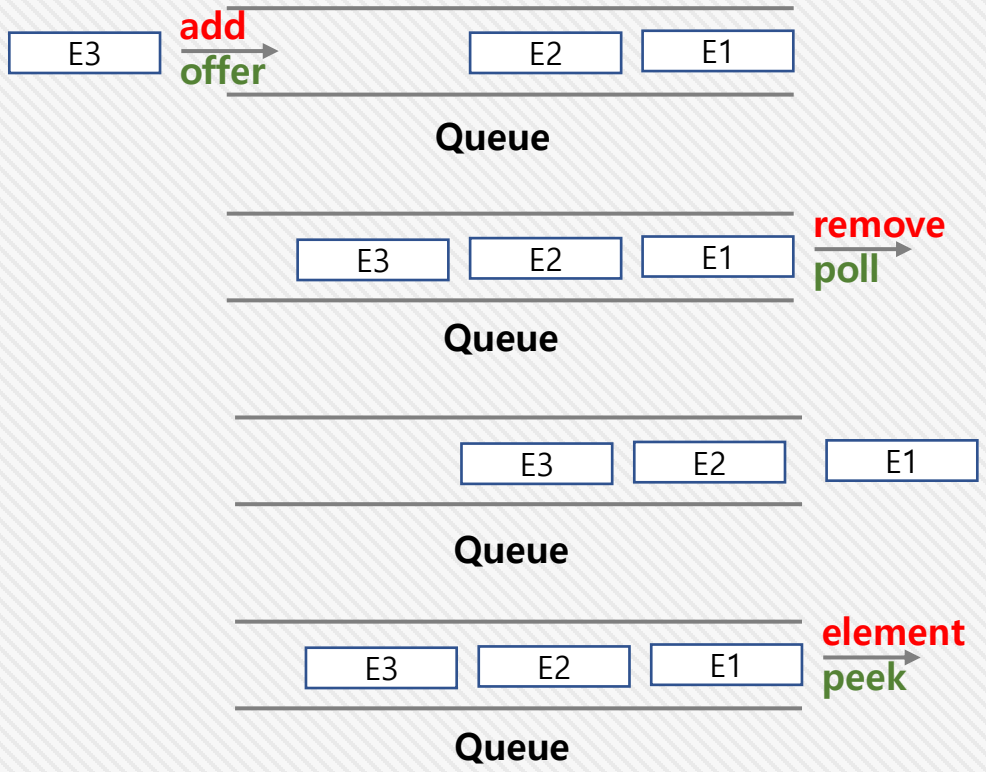
- LinkedList<E>가 Queue<E> 인터페이스의 구현체

```
Queue<제너릭타입지정> 참조변수 = new LinkedList<제너릭타입지정>();
```



주요 메서드

메서드	설명
boolean add(E e)	매개변수로 입력된 요소를 추가
boolean offer(E e)	매개변수로 입력된 요소를 추가 (예외 처리 포함)
E remove()	가장 상위에 있는 요소 꺼내기 (없으면 예외 발생)
E poll()	가장 상위에 있는 요소 꺼내기 (예외 처리 포함)
E element()	가장 상위에 있는 요소 리턴 (데이터는 변화 없음, 없으면 예외 발생)
E peek()	가장 상위에 있는 요소 리턴 (데이터는 변화 없음, 예외 처리 포함)



1. Queue<E>

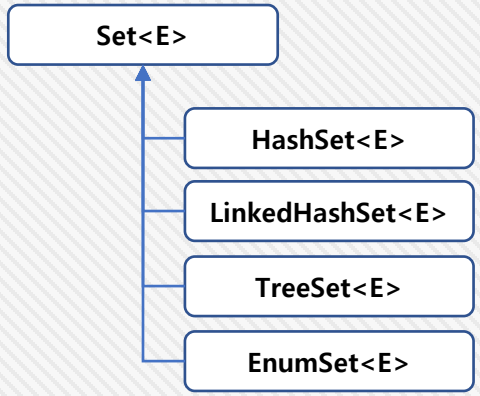

```
public static void main(String... args) {
  Queue<Integer> queue = new LinkedList<Integer>();

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

  System.out.println(queue.add(10) + " :" + queue.toString());  // true :[10]
  System.out.println(queue.add(20) + " :" + queue.toString());  // true :[10, 20]
  System.out.println(queue.add(30) + " :" + queue.toString());  // true :[10, 20, 30]
  System.out.println(queue.element());                           // 10
  System.out.println(queue.remove() + " :" + queue.toString()); // 10 :[20, 30]
  System.out.println(queue.remove() + " :" + queue.toString()); // 20 :[30]
  System.out.println(queue.poll() + " :" + queue.toString());   // 30 :[]
  System.out.println(queue.peek());                              // null
}
```

1. Set<E>

"데이터의 구분을 데이터 그 자체로 하므로 중복을 허용 하지 않음"

Set<E>

- List<E>는 요소(index)를 통해서 중복된 데이터를 찾을 수 있지만 Set<E>는 데이터의 구분을 데이터 그 자체로 찾는다.



➢ 주요 메서드

메서드	설명
int size()	객체내에 포함된 요소의 개수
boolean isEmpty()	비어 있는지 여부
boolean contains(Object o) boolean containsAll(Collection<?> c)	매개변수로 입력된 Object 존재 여부 매개변수로 입력된 Collection에 있는 모든 요소 존재 여부
Iterator<E> iterator()	객체 내의 데이터를 꺼내기 위한 Iterator객체
Object[] toArray();	배열로 변환
boolean add(E e) boolean addAll(Collection<? extends E> c)	요소 추가 Collection에 있는 모든 요소 추가
boolean remove(Object o) boolean removeAll(Collection<?> c)	요소 삭제 Collection에 있는 모든 요소 삭제
void clear()	요소 전체 삭제
boolean equals(Object o)	Index 위치에 있는 요소 값 반환

2. HashSet<E>

HashSet<E>

- 저장 용량 동적 관리 (기본 : 16)

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

01. 추가 : add

```
void add() {
  strSet.add("대한");
  strSet.add("민국");
  strSet.add("대한");
  System.out.println("* add : " + strSet.toString());
}
```

중복 허용 하지 않음 : 데이터 자체

```
* add : [민국, 대한]
```

02. Collection 전체 추가 : addAll

```
void addAll() {
  Set<String> strSet01 = new HashSet<String>();
  strSet01.add("우리");
  strSet01.add("나라");
  strSet.addAll(strSet01);
  System.out.println("* addAll : " +strSet.toString());
}
```

객체 전부 합침

```
* addAll : [나라, 우리, 민국, 대한]
```

03. 삭제 : remove

```
void remove() {
  strSet.remove("우리");
  System.out.println("* remove : " + strSet.toString());
}
```

```
* remove : [나라, 민국, 대한]
```

04. 객체 전체 삭제 : removeAll

```
void removeAll() {
  Set<String> strSet01 = new HashSet<String>();
  strSet01.add("대한");
  strSet01.add("민국");
  strSet.removeAll(strSet01);
  System.out.println("* removeAll : " +strSet.toString());
}
```

여러 건을 동시에 삭제

```
* removeAll : [나라]
```

2. HashSet<E>

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

05. 객체 데이터 존재 여부 체크 및 전체 삭제 : isEmpty, clear

```
void isEmptyAndClear() {
  if (strSet.isEmpty()) {
    System.out.println("* strSet 데이터 없음");
  } else {
    System.out.println("* strSet 데이터 있음");
    strSet.clear();
    System.out.println("* strSet clear 후 isEmpty : " + strSet.isEmpty());
  }
  add();
  addAll();
}
```

```
* strSet clear 후 isEmpty : true
```

06. 객체 데이터 크기 : size

```
void size() {
  System.out.println(String.format("* size :  %s 이 크기는 %d입니다",
          strSet.toString(), strSet.size()));
}
```

```
* size :  [나라, 우리, 민국, 대한] 이 크기는 4입니다
```

07. 객체 데이터 포함 여부 : contains

```
void contains() {
  if (strSet.contains("우리")) {
    System.out.println("* contains : '우리'가 존재 합니다");
  } else {
    System.out.println("* contains : '우리'가 존재 하지 않습니다.");
  }
}
```

```
* contains : '우리'가 존재 합니다
```

08. 객체 데이터 전체 포함 여부 : containsAll

```
void containsAll() {
  Set<String> strSet01 = new HashSet<String>();
  strSet01.add("우리");
  strSet01.add("나라");
  if (strSet.containsAll(strSet01)) {
    System.out.println(String.format("* containsAll : %s에 %s가 존재 합니다",
            strSet.toString(), strSet01.toString()));
  } else {
    System.out.println(String.format("* containsAll : %s에 %s가 존재 하지 않습니다",
            strSet.toString(), strSet01.toString()));
  }
}
```

```
* containsAll : [나라, 우리, 민국, 대한]에 [나라, 우리]가 존재 합니다
```

2. HashSet<E>

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

09. 객체 요소 반복 : iterator

```
void iterator() {
  Iterator strSetingIterator = strSet.iterator();
  while (strSetingIterator.hasNext() ) {
    System.out.println(String.format("* strSet : 요소 :  %s ",
            strSetingIterator.next()));
  }
}
```

```
* strSet : 요소 :  나라
* strSet : 요소 :  우리
* strSet : 요소 :  민국
* strSet : 요소 :  대한
```

10. 배열 변환 : toArray

변환 시 변환 Type에 0으로 생성 하면 원본의 크기 그대로

변환 시 변환 Type에 원본의 크기 보다 크게 설정 하면 빈 공간은 null

```
void convertArray() {
  String[] strArray = strSet.toArray(new String[0]);
  System.out.println(String.format("* strArray : 요소 :  %s ",
          Arrays.toString(strArray)));
  String[] strArray01 = strSet.toArray(new String[5]);
  System.out.println(String.format("* strArray01 : 요소 :  %s ",
          Arrays.toString(strArray01)));
}
```

```
* strArray : 요소 :  [나라, 우리, 민국, 대한]
```

```
* strArray01 : 요소 :  [나라, 우리, 민국, 대한, null]
```

iterator<E>

- 객체안에 있는 데이터를 1개씩 꺼내 처리
  - ➢ 주요 메서드

메서드	설명
boolean hasNext()	다음 요소가 있는지 체크해서 있으면 참 리턴
E next()	다음 요소 꺼내기

3. LinkedHashSet<E>

LinkedHashSet<E>

- 출력 순서가 입력 순서와 동일 한 순서 보장을 하지만 LinkedList처럼 중간에 삽입은 할 수 없다.
- HashSet과 동일한 기능을 하지만 **출력 순서가 입력 순서와 동일 한 순서 보장**

```
Set<String> strSet = new LinkedHashSet<String>();
```

01. 추가 : add

```
void add() {
  strSet.add("대한");
  strSet.add("민국");
  strSet.add("대한");
  System.out.println("* add : " + strSet.toString());
}
```

02. 삭제 : remove

```
void remove() {
  strSet.remove("우리");
  System.out.println("* remove : " + strSet.toString());
}
```

03. 객체 데이터 포함 여부 : contains

```
void contains() {
  if (strSet.contains("우리")) {
    System.out.println("* contains : '우리'가 존재 합니다");
  } else {
    System.out.println("* contains : '우리'가 존재 하지 않습니다.");
  }
}
```

04. 객체 데이터 존재 여부 체크 및 전체 삭제 : isEmpty, clear

```
void isEmptyAndClear() {
  if (strSet.isEmpty()) {
    System.out.println("* strSet 데이터 없음");
  } else {
    System.out.println("* strSet 데이터 있음");
    strSet.clear();
    System.out.println("* strSet clear 후 isEmpty : " + strSet.isEmpty());
  }
  add();
  addAll();
}
```

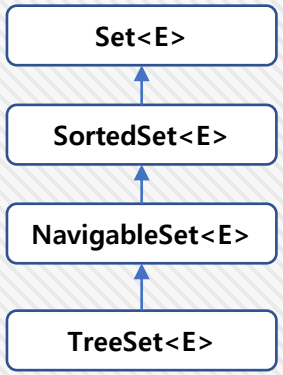
05. 객체 데이터 크기 : size, 객체 요소 반복 : iterator

```
void size() {
  System.out.println(String.format("* size :  %s 이 크기는 %d입니다",
          strSet.toString(), strSet.size()));
}

void iterator() {
  Iterator strSetingIterator = strSet.iterator();
  while (strSetingIterator.hasNext() ) {
    System.out.println(String.format("* strSet : 요소 :  %s ",
            strSetingIterator.next()));
  }
}
```

4. TreeSet<E>

TreeSet<E>

- 데이터의 크기 순으로 출력
- Set<E>의 성질을 그대로 유지 하면서 정렬과 검색이 추가됨
- HashSet과는 달리 TreeSet은 이진 탐색 트리(BinarySearchTree) 구조이므로 데이터 추가/삭제는 시간이 걸리지만 정렬, 검색에는 유리함
- 이진 탐색 트리(BinarySearchTree) 구조로 기본적으로 Nature Ordering을 지원
- 생성자의 매개변수로 Comparator객체를 입력하여 정렬 방법을 임의로 지정해 줄 수 있다.
- HashSet<E>는 객체 비교 시 같은지/다른지를 비교, TreeSet<E>는 두 객체의 크기 비교



Set<제너릭타입지정> 참조변수 = new TreeSet<제너릭타입지정>();
-> Set<E>에 있는 메서드만 사용 가능
TreeSet<제너릭타입지정> 참조변수 = new TreeSet <제너릭타입지정>();
-> Set<E>, SortedSet<E>, NavigableSet<E>에 있는 메서드 사용 가능 (정렬, 검색 ...)

➢ TreeSet 선언

```
TreeSet<Integer> set1 = new TreeSet<Integer>();                    // TreeSet생성 : 오름차순 정렬
TreeSet<Integer> set2 = new TreeSet<>();                           // 타입 파라미터 생략가능
TreeSet<Integer> set3 = new TreeSet<>(Collections.reverseOrder()); // 내림차순 정렬
TreeSet<Integer> set3 = new TreeSet<Integer>(set1);                // set1의 모든 값을 가진 TreeSet생성
TreeSet<Integer> set4 = new TreeSet<Integer>(Arrays.asList(1,2,3)); // 초기값 지정
```

4. TreeSet<E>

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

➢ 주요 메서드

메서드	설명
int size()	객체내에 포함된 요소의 개수
boolean isEmpty()	비어 있는지 여부
boolean contains(Object o)	매개변수로 입력된 Object 존재 여부
Iterator<E> iterator()	객체 내의 데이터를 꺼내기 위한 Iterator객체
Object[] toArray();	배열로 변환
boolean add(E e)	요소 추가
boolean remove(Object o)	요소 삭제
void clear()	요소 전체 삭제
boolean equals(Object o)	Index 위치에 있는 요소 값 반환

01. 추가:add

```
void insertTreeSet() {
  for ( int i = 0 ; i < 10; i++) {
    this.treeSet.add(i);
  }
  System.out.println("* TreeSet 전체 : " + treeSet.toString());
}
```

```
* TreeSet 전체 : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

02. set Method

```
void setMethod() {
  System.out.println("* TreeSet size : " + treeSet.size());
  System.out.println("* TreeSet contains : " + treeSet.contains(8));
  System.out.println("* TreeSet contains : " + treeSet.contains(10));
}
```

```
* TreeSet size : 8
* TreeSet contains : true
* TreeSet contains : false
```

4. TreeSet<E>

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

➢ 주요 메서드

메서드	설명
E first()	가장 작은 요소 값 리턴 (데이터 변화 없음)
E last()	자장 큰 요소 값 리턴 (데이터 변화 없음)
E lower(E e)	입력된 값보다 작은 값 중 가장 큰 요소 값 리턴 (데이터 변화 없음)
E higher(E e)	입력된 값보다 큰 값 가장 작은 요소 값 리턴 (데이터 변화 없음)
E floor(E e)	입력된 값보다 같거나 작은 값 중 가장 큰 요소 값 리턴 (데이터 변화 없음)

메서드	설명
E ceiling(E e)	입력된 값보다 같거나 큰 값 중 가장 작은 요소 값 리턴 (데이터 변화 없음)
E pollFirst()	가장 작은 요소 값 리턴 (데이터 변화 있음)
E pollLast()	자장 큰 요소 값 리턴 (데이터 변화 있음)

03. 데이터 검색 : first, last, lower, higher, floor, pollFirst, pollLast

```
void getTreeSet() {
  System.out.println("* TreeSet first : " + treeSet.first());
  System.out.println("* TreeSet last :  " + treeSet.last());
  System.out.println("* TreeSet lower :  " + treeSet.lower( e: 6));
  System.out.println("* TreeSet higher :  " + treeSet.higher( e: 6));
  System.out.println("* TreeSet floor :  " + treeSet.floor( e: 6));
  System.out.println("* TreeSet ceiling :  " + treeSet.ceiling( e: 4));
  System.out.println("* TreeSet pollFirst 이전 : " + treeSet.toString());
  System.out.println("* TreeSet pollFirst :  " + treeSet.pollFirst());
  System.out.println("* TreeSet pollFirst 이후, pollLast 이전 : " + treeSet.toString());
  System.out.println("* TreeSet pollLast :  " + treeSet.pollLast());
  System.out.println("* TreeSet pollLast 이후 : " + treeSet.toString());
}
```

```
* TreeSet first : 0
* TreeSet last :  9
* TreeSet lower :  5
* TreeSet higher :  7
* TreeSet floor :  6
* TreeSet ceiling :  4
* TreeSet pollFirst 이전 : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
* TreeSet pollFirst :  0
* TreeSet pollFirst 이후, pollLast 이전 : [1, 2, 3, 4, 5, 6, 7, 8, 9]
* TreeSet pollLast :  9
* TreeSet pollLast 이후 : [1, 2, 3, 4, 5, 6, 7, 8]
```

4. TreeSet<E>

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

➢ SortedSet 주요 메서드

메서드	설명
SortedSet<E> headSet(E toElement)	입력요소 값 미만인 모든 요소로 구성된 Set 리턴 (toElement 미포함)
SortedSet<E> tailSet(E fromElement)	입력요소 값 이상인 모든 요소로 구성된 Set 리턴 (fromElement 포함)
SortedSet<E> subSet(E fromElement, E toElement)	입력요소 from요소 이상 to요소 미만인 모든 요소로 구성된 Set 리턴 (fromElement 포함, toElement 미포함)

04. 데이터 부분 생성 : headSet, tailSet, subSet

```
SortedSet headSet = treeSet.headSet(5);
System.out.println("* TreeSet treeSet : " + treeSet.toString());
System.out.println("* SortedSet headSet(5) : " + headSet.toString());
SortedSet tailSet = treeSet.tailSet(5);
System.out.println("* SortedSet tailSet(5) : " + tailSet.toString());
SortedSet subSet = treeSet.subSet(3,6);
System.out.println("* SortedSet subSet(3,6) : " + subSet.toString());
```

```
* TreeSet treeSet : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
* SortedSet headSet(5) : [0, 1, 2, 3, 4]
* SortedSet tailSet(5) : [5, 6, 7, 8, 9]
* SortedSet subSet(3,6) : [3, 4, 5]
```

4. TreeSet<E>

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

➢ NavigableSet 주요 메서드

메서드	설명
NavigableSet<E> headSet(E toElement, boolean inclusive)	입력요소 값 미만인 모든 요소로 구성된 Set 리턴 (inclusive : true 포함, false 미포함)
NavigableSet<E> tailSet(E fromElement, boolean inclusive)	입력요소 값 이상인 모든 요소로 구성된 Set 리턴 (inclusive : true 포함, false 미포함)
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	입력요소 from요소 이상 to요소 미만인 모든 요소로 구성된 Set 리턴 (fromInclusive : true 포함., false 미포함 , toInclusive true 포함., false 미포함)
NavigableSet<E> descendingSet()	현재 정렬 기준의 반대 리턴 (내림 차순 아님)

04. 데이터 부분 생성/정렬 : headSet, tailSet, subSet, descendingSet

```
NavigableSet headSet = treeSet.headSet(5, true);
System.out.println("* TreeSet treeSet : " + treeSet.toString());
System.out.println("* NavigableSet headSet(5, true) : " + headSet.toString());
NavigableSet tailSet = treeSet.tailSet(5, false);
System.out.println("* NavigableSet tailSet(5, false) : " + tailSet.toString());
```

```
* TreeSet treeSet : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
* NavigableSet headSet(5, true) : [0, 1, 2, 3, 4, 5]
* NavigableSet tailSet(5, false) : [6, 7, 8, 9]
```

```
NavigableSet subSetfromInclude = treeSet.subSet(3, true, 6, false);
System.out.println("* NavigableSet subSet(3, true, 6, false) : " + subSetfromInclude.toString());
NavigableSet subSetToInclude = treeSet.subSet(3, false, 6, true);
System.out.println("* NavigableSet subSet(3, false, 6, true) : " + subSetToInclude.toString());
NavigableSet subSetFromToInclude = treeSet.subSet(3, true, 6, true);
System.out.println("* NavigableSet subSet(3, true, 6, true) : " + subSetFromToInclude.toString());
NavigableSet subSetFromTofalse = treeSet.subSet(3, false, 6, false);
System.out.println("* NavigableSet subSet(3, false, 6, false) : " + subSetFromTofalse.toString());
```

```
* NavigableSet subSet(3, true, 6, false) : [3, 4, 5]
* NavigableSet subSet(3, false, 6, true) : [4, 5, 6]
* NavigableSet subSet(3, true, 6, true) : [3, 4, 5, 6]
* NavigableSet subSet(3, false, 6, false) : [4, 5]
```

```
NavigableSet descendingSet = treeSet.descendingSet();
System.out.println("* NavigableSet descendingSet : " + descendingSet.toString());
```

```
* NavigableSet descendingSet : [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

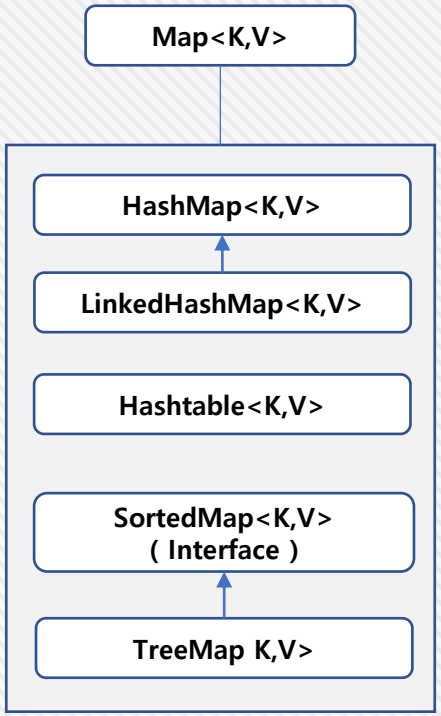
1. Map<K,V>

" Key, Value 구조로 데이터를 저장 "

Map<K,V>

- Key, Value 구조로 데이터를 저장 하는 객체로 K,V의 한 쌍을 Entry(Map.Entry Type)라고 함
- Key는 증복 되지 않지만 Value는 증복

➢ 주요 메서드



메서드	설명
int size()	객체내에 포함된 요소의 개수
boolean isEmpty()	비어 있는지 여부
boolean containsKey(Object key)	매개변수로 입력된 객체의 KEY가 존재 여부
boolean containsValue(Object value)	매개변수로 입력된 객체의 VALUE가 존재 여부
V get(Object key)	매개변수로 입력된 객체의 KEY의 VALUE 리턴
V put(K key, V value)	매개변수로 입력된 KEY, VALUE를 객체에 추가
void putAll(Map<? extends K, ? extends V> m)	매개변수로 입력된 Map 전체를 객체에 저장 (전체 변경)
void clear()	요소 전체 삭제
Set<K> keySet()	Map에 KEY만 모아서 Set으로 리턴
Collection<V> values()	Map의 값만 모아서 Collection으로 리턴
Set<Map.Entry<K, V>> entrySet()	Map의 Entry들을 Set으로 리턴
V remove(Object key)	Map에 있는 key의 Entry 삭제
boolean remove(Object key, Object value)	Map에 있는 key, value Entry 삭제
V replace(K key, V value)	Key값의 Value를 입력된 Value로 변경
boolean replace(K key, V oldValue, V newValue)	Key, oldValue Entry의 Value로 newValue로 변경, entry가 없으면 false

2. HashMap<K,V>

" Key, Value 구조로 데이터를 저장 "

HashMap<K,V>

- Key, Value 구조로 데이터를 저장 하는 객체로 초기 저장 용량 기본값은 16, 16을 넘으면 자동 증가

➢ 주요 메서드

메서드	설명
V put(K key, V value)	매개변수로 입력된 KEY, VALUE를 객체에 추가
void putAll(Map<? extends K, ? extends V> m)	매개변수로 입력된 Map 전체를 객체에 저장 (전체 변경)
V replace(K key, V value)	Key값의 Value를 입력된 Value로 변경
boolean replace(K key, V oldValue, V newValue)	Key, oldValue Entry의 Value로 newValue로 변경, entry가 없으면 false

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

01. 데이터 추가 : put, putAll

```
void putHashMap(){
  strMap.put(1, "한국");
  strMap.put(2, "중국");
  strMap.put(3, "일본");
  strMap.put(4, "대만");
  Map<Integer, String> strMapAll = new HashMap<Integer, String>();
  strMapAll.put(1, "영국");
  strMapAll.putAll(strMap); // 전부 교체
  strMap.put(5, "대만"); // Value 중복
  System.out.println("* HashMap : strMap : " + strMap.toString());
  System.out.println("* HashMap : strMapAll : " + strMapAll.toString());
}
```

전부 교체 됨

```
* HashMap : strMap : {1=한국, 2=중국, 3=일본, 4=대만, 5=대만}
* HashMap : strMapAll : {1=한국, 2=중국, 3=일본, 4=대만}
```

02. 데이터 수정 : replace

```
void replaceMap() {
  strMap.replace(5, "영국");
  strMap.replace(4, "대만", "미국");
  strMap.replace(4, "대만", "영국"); // 처리 하지 않음
  System.out.println("* HashMap : strMap : " + strMap.toString());
}
```

```
* HashMap : strMap : {1=한국, 2=중국, 3=일본, 4=미국, 5=영국}
```

2. HashMap<K,V>

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

➢ 주요 메서드

메서드	설명
V get(Object key)	매개변수로 입력된 객체의 KEY의 VALUE 리턴
boolean containsKey(Object key)	매개변수로 입력된 객체의 KEY가 존재 여부
boolean containsValue(Object value)	매개변수로 입력된 객체의 VALUE가 존재 여부

메서드	설명
Set<K> keySet()	Map에 KEY만 모아서 Set으로 리턴
Set<Map.Entry<K, V>> entrySet()	Map의 Entry들을 Set으로 리턴

03. 데이터 추가: get, containsKey, containsValue

```
void getMap() {
  System.out.println("* HashMap : strMap : strMap.get(2) : "
          + strMap.get(2));
  System.out.println("* HashMap : strMap : strMap.containsKey(2) : "
          + strMap.containsKey(2));
  System.out.println("* HashMap : strMap : strMap.containsKey(10) : "
          + strMap.containsKey(10));
  System.out.println("* HashMap : strMap : strMap.containsValue(\"한국\") : "
          + strMap.containsValue("한국"));
  System.out.println("* HashMap : strMap : strMap.containsValue(\"독일\") : "
          + strMap.containsValue("독일"));
}
```

```
* HashMap : strMap : strMap.get(2) : 중국
* HashMap : strMap : strMap.containsKey(2) : true
* HashMap : strMap : strMap.containsKey(10) : false
* HashMap : strMap : strMap.containsValue("한국") : true
* HashMap : strMap : strMap.containsValue("독일") : false
```

04. 데이터 추가: entrySet, keySet

```
Set<Map.Entry<Integer, String>> entrySet = strMap.entrySet();
for ( Map.Entry entry: entrySet) {
  System.out.println("* HashMap : strMap : entrySet Map.Entry : "
          + entry.toString()
          + ", key : " + entry.getKey()
          + ", Value : " + entry.getValue());
}
```

```
* HashMap : strMap : entrySet Map.Entry : 1=한국, key : 1, Value : 한국
* HashMap : strMap : entrySet Map.Entry : 2=중국, key : 2, Value : 중국
* HashMap : strMap : entrySet Map.Entry : 3=일본, key : 3, Value : 일본
* HashMap : strMap : entrySet Map.Entry : 4=미국, key : 4, Value : 미국
* HashMap : strMap : entrySet Map.Entry : 5=영국, key : 5, Value : 영국
```

```
Set<Integer> keySet = strMap.keySet();
for ( Integer key : keySet) {
  System.out.println("* HashMap : strMap : keySet : "
          + " key : " + key
          + ", value : " + strMap.get(key));
}
```

```
* HashMap : strMap : keySet :  key : 1, value : 한국
* HashMap : strMap : keySet :  key : 2, value : 중국
* HashMap : strMap : keySet :  key : 3, value : 일본
* HashMap : strMap : keySet :  key : 4, value : 미국
* HashMap : strMap : keySet :  key : 5, value : 영국
```

2. HashMap<K,V>

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

➢ 주요 메서드

메서드	설명
Collection<V> values()	Map의 값만 모아서 Collection으로 리턴

메서드	설명
V remove(Object key)	Map에 있는 key의 Entry 삭제
boolean remove(Object key, Object value)	Map에 있는 key, value Entry 삭제

05. 데이터 변환 : values

```
// Map을 Collection으로 변환
Collection valueCollection = strMap.values();
System.out.println("* HashMap : strMap : trMap.values() : "
        + valueCollection.toString());

// Map type 객체를 List Type 객체로 변환
ArrayList countrys =  new ArrayList(strMap.values());
System.out.println("* HashMap : strMap : new ArrayList(strMap.values()) : "
        + countrys.toString());
```

List 로 변경

```
* HashMap : strMap : trMap.values() : [한국, 중국, 일본, 미국, 영국]
* HashMap : strMap : new ArrayList(strMap.values()) : [한국, 중국, 일본, 미국, 영국]
```

06. 데이터 삭제 : entrySet, keySet

```
System.out.println("* HashMap : strMap : remove 이전  : "
        + strMap.toString());
strMap.remove(2);
System.out.println("* HashMap : strMap : strMap.remove(2) 이후  : "
        + strMap.toString());
strMap.remove(3, "일본");
System.out.println("* HashMap : strMap : strMap.remove(3, \"일본\") 이후  : "
        + strMap.toString());
strMap.remove(4, "영국");
System.out.println("* HashMap : strMap : strMap.remove(4, \"영국\") 이후  : "
        + strMap.toString());
strMap.clear();
System.out.println("* HashMap : strMap : sstrMap.clear() 이후  : "
        + strMap.toString());
```

```
* HashMap : strMap : remove 이전  : {1=한국, 2=중국, 3=일본, 4=미국, 5=영국}
* HashMap : strMap : strMap.remove(2) 이후  : {1=한국, 3=일본, 4=미국, 5=영국}
* HashMap : strMap : strMap.remove(3, "일본") 이후  : {1=한국, 4=미국, 5=영국}
* HashMap : strMap : strMap.remove(4, "영국") 이후  : {1=한국, 4=미국, 5=영국}
* HashMap : strMap : sstrMap.clear() 이후  : {}
```

3. HashTable<K,V>

"동기화(synchronized)로 구현이 되어 있어서 멀티 쓰레드에 안전 "

HashTable<K,V>

- 동기화(synchronized)로 구현이 되어 있어서 멀티 쓰레드에 안전 하다.
- HashMap<K,V)한 동일한 기능을 가지고 있다. (싱글 쓰레드에 적합)

01. 데이터 추가: put, putAll

```
strMap.put(1, "한국");
strMap.put(2, "중국");
strMap.put(3, "일본");
strMap.put(4, "대만");
Map<Integer, String> strMapAll = new Hashtable<Integer, String>();
strMapAll.put(1, "영국");
strMapAll.putAll(strMap); // 전부 교체
strMap.put(5, "대만"); // Value 중복
System.out.println("* Hashtable : strMap : " + strMap.toString());
System.out.println("* Hashtable : strMapAll : " + strMapAll.toString());
```

02. 데이터 수정: replace

```
strMap.replace(5, "영국");
strMap.replace(4, "대만", "미국");
strMap.replace(4, "대만", "영국"); // 처리 하지 않음
System.out.println("* Hashtable : strMap : " + strMap.toString());
```

03. 데이터 추가: get, containsKey, containsValue

```
System.out.println("* Hashtable : strMap : strMap.get(2) : "
        + strMap.get(2));
System.out.println("* Hashtable : strMap : strMap.containsKey(2) : "
        + strMap.containsKey(2));
System.out.println("* Hashtable : strMap : strMap.containsKey(10) : "
        + strMap.containsKey(10));
System.out.println("* Hashtable : strMap : strMap.containsValue(\"한국\") : "
        + strMap.containsValue("한국"));
System.out.println("* Hashtable : strMap : strMap.containsValue(\"독일\") : "
        + strMap.containsValue("독일"));
```

3. Hashtable<K,V>

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

04. 데이터 추가 : entrySet, keySet, values

```
// Map 안에 있는 Entry의 key, value
Set<Map.Entry<Integer, String>> entrySet = strMap.entrySet();
for ( Map.Entry entry: entrySet) {
  System.out.println("* Hashtable : strMap : entrySet Map.Entry : "
          + entry.toString()
          + ", key : " + entry.getKey()
          + ", Value : " + entry.getValue());
}

// Map 안에 있는 KeySet를 사용한 Value
Set<Integer> keySet = strMap.keySet();
for ( Integer key : keySet) {
  System.out.println("* Hashtable : strMap : keySet : "
          + " key : " + key
          + ", value : " + strMap.get(key));
}

// Map을 Collection으로 변환
Collection valueCollection = strMap.values();
System.out.println("* Hashtable : strMap : trMap.values() : "
        + valueCollection.toString());

// Map type 객체를 List Type 객체로 변환
ArrayList countrys =  new ArrayList(strMap.values());
System.out.println("* Hashtable : strMap : new ArrayList(strMap.values()) : "
        + countrys.toString());
```

05. 데이터 삭제 : remove

```
System.out.println("* Hashtable : strMap : remove 이전  : "
        + strMap.toString());
strMap.remove(2);
System.out.println("* Hashtable : strMap : strMap.remove(2) 이후  : "
        + strMap.toString());
strMap.remove(3, "일본");
System.out.println("* Hashtable : strMap : strMap.remove(3, \"일본\") 이후
        + strMap.toString());
strMap.remove(4, "영국");
System.out.println("* Hashtable : strMap : strMap.remove(4, \"영국\") 이후
        + strMap.toString());
strMap.clear();
System.out.println("* Hashtable : strMap : sstrMap.clear() 이후  : "
        + strMap.toString());
```

4. LinkedHashMap<K,V>

" HashMap과 기본 특성은 동일 하면서 순서 정보를 추가로 갖고 있는 Collection "

LinkedHashMap<K,V>

- HashMap과 기본 특성은 동일 하면서 순서 정보를 추가로 갖고 있는 Collection
- Key가 서로 Link되어 있는 모습

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

01. 데이터 추가: put, putAll

```
strMap.put(1, "한국");
strMap.put(2, "중국");
strMap.put(3, "일본");
strMap.put(4, "대만");
Map<Integer, String> strMapAll = new LinkedHashMap<Integer, String>();
strMapAll.put(1, "영국");
strMapAll.putAll(strMap); // 전부 교체
strMap.put(5, "대만"); // Value 중복
System.out.println("* LinkedHashMap : strMap : " + strMap.toString());
System.out.println("* LinkedHashMap : strMapAll : " + strMapAll.toString());
```

02. 데이터 수정: replace

```
strMap.replace(5, "영국");
strMap.replace(4, "대만", "미국");
strMap.replace(4, "대만", "영국"); // 처리 하지 않음
System.out.println("* LinkedHashMap : strMap : " + strMap.toString());
```

03. 데이터 추가: get, containsKey, containsValue

```
System.out.println("* LinkedHashMap : strMap : strMap.get(2) : "
        + strMap.get(2));
System.out.println("* LinkedHashMap : strMap : strMap.containsKey(2) : "
        + strMap.containsKey(2));
System.out.println("* LinkedHashMap : strMap : strMap.containsKey(10) : "
        + strMap.containsKey(10));
System.out.println("* LinkedHashMap : strMap : strMap.containsValue(\"한국\") : "
        + strMap.containsValue("한국"));
System.out.println("* LinkedHashMap : strMap : strMap.containsValue(\"독일\") : "
        + strMap.containsValue("독일"));
```

4. LinkedHashMap<K,V>

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

04. 데이터 추가 : entrySet, keySet, values

```
// Map 안에 있는 Entry의 key, value
Set<Map.Entry<Integer, String>> entrySet = strMap.entrySet();
for ( Map.Entry entry: entrySet) {
  System.out.println("* LinkedHashMap : strMap : entrySet Map.Entry : "
          + entry.toString()
          + ", key : " + entry.getKey()
          + ", Value : " + entry.getValue());
}

// Map 안에 있는 KeySet를 사용한 Value
Set<Integer> keySet = strMap.keySet();
for ( Integer key : keySet) {
  System.out.println("* LinkedHashMap : strMap : keySet : "
          + " key : " + key
          + ", value : " + strMap.get(key));
}

// Map을 Collection으로 변환
Collection valueCollection = strMap.values();
System.out.println("* LinkedHashMap : strMap : trMap.values() : "
        + valueCollection.toString());

// Map type 객체를 List Type 객체로 변환
ArrayList countrys =  new ArrayList(strMap.values());
System.out.println("* LinkedHashMap : strMap : new ArrayList(strMap.values()) : "
        + countrys.toString());
```

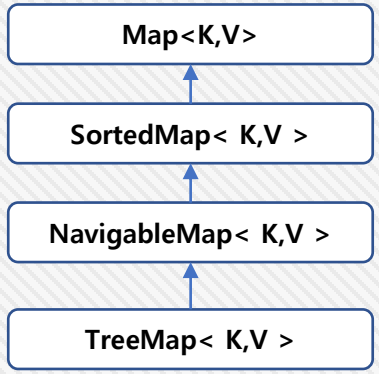
05. 데이터 삭제 : remove

```
System.out.println("* LinkedHashMap : strMap : remove 이전  : "
        + strMap.toString());
strMap.remove(2);
System.out.println("* LinkedHashMap : strMap : strMap.remove(2) 이후  : "
        + strMap.toString());
strMap.remove(3, "일본");
System.out.println("* LinkedHashMap : strMap : strMap.remove(3, \"일본\") 이후  : "
        + strMap.toString());
strMap.remove(4, "영국");
System.out.println("* LinkedHashMap : strMap : strMap.remove(4, \"영국\") 이후  : "
        + strMap.toString());
strMap.clear();
System.out.println("* LinkedHashMap : strMap : sstrMap.clear() 이후  : "
        + strMap.toString());
```

5. TreeMap<K, V>

TreeSet<E>

- 데이터의 크기 순으로 출력
- Map<K,V>의 성질을 그대로 유지 하면서 정렬과 검색이 추가됨



Map<제너릭타입지정> 참조변수 = new TreeMap<제너릭타입지정>();
-> Map<K,V>에 있는 메서드만 사용 가능
TreeMap<제너릭타입지정> 참조변수 = new TreeMap <제너릭타입지정>();
-> Map<K,V>, Sorted Map<K,V>,, Navigable Map<K,V>, 에 있는 메서드 사용 가능 (정렬, 검색 ...)

➢ TreeSet 선언

```
TreeMap<Integer, String> map1 = new TreeMap<Integer, String>();          // TreeMap생성 :
TreeMap<Integer, String> map1 = new TreeMap<>();                          // 타입 파라미터 생략가능
TreeMap<Integer, String> map1 = new TreeMap<Integer, String>(map1);       // map1의 모든 값을 가진 TreeMap 생성
```

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

01. 데이터 추가 :

```
void insertTreeMap() {
  for ( int i = 0 ; i < 10; i++) {
    this.map.put(i, i + "번째");
  }
  System.out.println("* map 전체 : " + map.toString());
}
```

```
* map 전체 : {0=0번째, 1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째, 6=6번째, 7=7번째, 8=8번째, 9=9번째}
```

5. TreeMap<K, V>

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

➢ 주요 메서드

메서드	설명
K firstkey()	가장 작은 요소 값 리턴
K lastkey ()	자장 큰 요소 값 리턴
K lowerkey (K key)	입력된 값보다 작은 값 중 가장 큰 요소 값 리턴
K higherkey (K key)	입력된 값보다 큰 값 가장 작은 요소 값 리턴
K floorkey(K key)	입력된 값보다 같거나 작은 값 중 가장 큰 요소 값 리턴
K ceilingkey(K key)	입력된 값보다 같거나 큰 값 중 가장 작은 요소 값 리턴

02. 데이터 검색 : firstkey, lastkey, lowerkey, higherkey, floorkey, ceilingkey

```
void getTreeMapK() {
  System.out.println("* TreeMap firstKey : " + map.firstKey());
  System.out.println("* TreeMap lastKey :  " + map.lastKey());
  System.out.println("* TreeMap lowerKey :  " + map.lowerKey(6));
  System.out.println("* TreeMap higherKey :  " + map.higherKey(6));
  System.out.println("* TreeMap floorKey :  " + map.floorKey(6));
  System.out.println("* TreeMap ceilingKey :  " + map.ceilingKey(4));
}
```

```
* TreeMap firstKey : 0
* TreeMap lastKey :  9
* TreeMap lowerKey :  5
* TreeMap higherKey :  7
* TreeMap floorKey :  6
* TreeMap ceilingKey :  4
* TreeMap firstEntry : 0=0번째
* TreeMap lastEntry :  9=9번째
* TreeMap lowerEntry :  5=5번째
* TreeMap higherKey :  7=7번째
* TreeMap higherEntry :  6=6번째
* TreeMap ceilingEntry :  4=4번째
```

5. TreeMap<K, V>

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

➢ 주요 메서드

메서드	설명
Map.Entry<K,V> firstEntry()	가장 작은 요소 Entry 리턴
Map.Entry<K,V> lastEntry()	자장 큰 요소 Entry 리턴
Map.Entry<K,V> lowerEntry(K key)	입력된 값보다 작은 값 중 가장 큰 요소 Entry 리턴
Map.Entry<K,V> higherEntry (K key)	입력된 값보다 큰 값 가장 작은 요소 Entry 리턴
Map.Entry<K,V> floorEntry(K key)	입력된 값보다 같거나 작은 값 중 가장 큰 요소 Entry 리턴
Map.Entry<K,V> ceilingkey(K key)	입력된 값보다 같거나 큰 값 중 가장 작은 요소 Entry 리턴
Map.Entry<K,V> pollFirstEntry()	가장 작은 요소 Entry 리턴 (데이터 변화 있음)
Map.Entry<K,V> pollLastEntry()	자장 큰 요소 Entry 리턴 (데이터 변화 있음)

03. 데이터 검색 Entry : firstEntry, lastEntry, lowerEntry, higherEntry, floorEntry, ceilingEntry, pollFirstEntry, pollLastEntry

```
void getTreeMapEntry() {
  System.out.println("* TreeMap firstEntry : " + map.firstEntry());
  System.out.println("* TreeMap lastEntry :  " + map.lastEntry());
  System.out.println("* TreeMap lowerEntry :  " + map.lowerEntry(6));
  System.out.println("* TreeMap higherKey :  " + map.higherEntry(6));
  System.out.println("* TreeMap higherEntry :  " + map.floorEntry(6));
  System.out.println("* TreeMap ceilingEntry :  " + map.ceilingEntry(4));
  System.out.println("* TreeMap pollFirstEntry 이전 : " + map.toString());
  System.out.println("* TreeMap pollFirstEntry :  " + map.pollFirstEntry());
  System.out.println("* TreeMap pollFirstEntry 이후, pollLast 이전 : " + map.toString());
  System.out.println("* TreeMap pollLastEntry :  " + map.pollLastEntry());
  System.out.println("* TreeMap pollLastEntry 이후 : " + map.toString());
}
```

```
* TreeMap firstEntry : 0=0번째
* TreeMap lastEntry :  9=9번째
* TreeMap lowerEntry :  5=5번째
* TreeMap higherKey :  7=7번째
* TreeMap higherEntry :  6=6번째
* TreeMap ceilingEntry :  4=4번째
* TreeMap pollFirstEntry 이전 : {0=0번째, 1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째, 6=6번째, 7=7번째, 8=8번째, 9=9번째}
* TreeMap pollFirstEntry :  0=0번째
* TreeMap pollFirstEntry 이후, pollLast 이전 : {1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째, 6=6번째, 7=7번째, 8=8번째, 9=9번째}
* TreeMap pollLastEntry :  9=9번째
* TreeMap pollLastEntry 이후 : {1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째, 6=6번째, 7=7번째, 8=8번째}
```

5. TreeMap<K, V>

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

➢ SortedMap 주요 메서드

메서드	설명
SortedMap<K,V> headMap(K toKey)	입력요소 값 미만인 모든 요소로 구성된 Map 리턴 (toKey 미포함)
SortedMap<K,V> tailMap(K fromKey)	입력요소 값 이상인 모든 요소로 구성된 Map 리턴 (fromKey 포함)
SortedMap<K,V> subMap(K fromKey, K toKey)	입력요소 from요소 이상 to요소 미만인 모든 요소로 구성된 Map 리턴 (fromKey 포함, toKey 미포함)

04. 데이터 검색: headSet, tailSet, subSet

```
void sortedMap() {
  insertTreeMap();
  SortedMap headMap = map.headMap(5);
  System.out.println("* TreeMap map : " + map.toString());
  System.out.println("* SortedMap headMap(5) : " + headMap.toString());
  SortedMap tailMap = map.tailMap(5);
  System.out.println("* SortedMap tailMap(5) : " + tailMap.toString());
  SortedMap subMap = map.subMap(3,6);
  System.out.println("* SortedMap subMap(3,6) : " + subMap.toString());
}
```

```
* SortedMap headMap(5) : {0=0번째, 1=1번째, 2=2번째, 3=3번째, 4=4번째}
* SortedMap tailMap(5) : {5=5번째, 6=6번째, 7=7번째, 8=8번째, 9=9번째}
* SortedMap subMap(3,6) : {3=3번째, 4=4번째, 5=5번째}
```

5. TreeMap<K, V>

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

➢ NavigableMap 주요 메서드

메서드	설명
NavigableSet<K,V> headMap(E toKey, boolean inclusive)	입력요소 값 미만인 모든 요소로 구성된 Map 리턴 (inclusive : true 포함, false 미포함)
NavigableSet<K,V> tailMap(E fromKey, boolean inclusive)	입력요소 값 이상인 모든 요소로 구성된 Map 리턴 (inclusive : true 포함, false 미포함)
NavigableSet<K,V> subMap(E fromKey, boolean fromInclusive, E toKey, boolean toInclusive)	입력요소 from요소 이상 to요소 미만인 모든 요소로 구성된 Map 리턴 (fromInclusive : true 포함., false 미포함 , toInclusive true 포함., false 미포함)
NavigableSet<K,V> descendingMap()	현재 정렬 기준의 반대 리턴 (내림 차순 아님)

04. 데이터 부분 생성/정렬 : headSet, tailSet, subSet, descendingSet

```
NavigableMap headMap = map.headMap(5, true);
System.out.println("* TreeSet treeSet : " + map.toString());
System.out.println("* NavigableMap headMap(5, true) : " + headMap.toString());
NavigableMap tailMap = map.tailMap(5, false);
System.out.println("* NavigableMap tailMap(5, false) : " + tailMap.toString());
NavigableMap subMapfromInclude = map.subMap(3, true, 6, false);
System.out.println("* NavigableMap subMap(3, true, 6, false) : " + subMapfromInclude.toString());
NavigableMap subMaptToInclude = map.subMap(3, false, 6, true);
System.out.println("* NavigableMap subMap(3, false, 6, true) : " + subMaptToInclude.toString());
NavigableMap subMapFromToInclude = map.subMap(3, true, 6, true);
System.out.println("* NavigableMap subMap(3, true, 6, true) : " + subMapFromToInclude.toString());
NavigableMap subMapFromTofalse = map.subMap(3, false, 6, false);
System.out.println("* NavigableMap subMap(3, false, 6, false) : " + subMapFromTofalse.toString());
NavigableMap descendingMap = map.descendingMap();
System.out.println("* NavigableMap descendingMap : " + descendingMap.toString());
```

```
* map 전체 : {0=0번째, 1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째, 6=6번째, 7=7번째, 8=8번째, 9=9번째}
* TreeMap map : {0=0번째, 1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째, 6=6번째, 7=7번째, 8=8번째, 9=9번째}
* NavigableMap headMap(5, true) : {0=0번째, 1=1번째, 2=2번째, 3=3번째, 4=4번째, 5=5번째}
* NavigableMap tailMap(5, false) : {6=6번째, 7=7번째, 8=8번째, 9=9번째}
* NavigableMap subMap(3, true, 6, false) : {3=3번째, 4=4번째, 5=5번째}
* NavigableMap subMap(3, false, 6, true) : {4=4번째, 5=5번째, 6=6번째}
* NavigableMap subMap(3, true, 6, true) : {3=3번째, 4=4번째, 5=5번째, 6=6번째}
* NavigableMap subMap(3, false, 6, false) : {4=4번째, 5=5번째}
* NavigableMap descendingMap : {9=9번째, 8=8번째, 7=7번째, 6=6번째, 5=5번째, 4=4번째, 3=3번째, 2=2번째, 1=1번째, 0=0번째}
```