컬렉션 프레임 워크

: 다수의 객체들을 사용함에 있어 자바가 배열의 문제점을 해결하고, 널리 알려져 있는 자료구조를 바탕으로 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 java.util 패키지에 컬렉션과 관련된 인터페이스와 클래스들을 포함시켜 놓은 것

\*프레임워크

: 사용방법을 미리 정해 놓은 라이브러리

컬렉션의 종류

* List

1. ArrayList
2. Vector
3. LinkedList

* Set

1. HashSet
2. TreeSet

* Map(예외, 컬렉션 아님)

1. HashMap
2. Hashtable
3. TreeMap
4. Properties

List : 순서를 유지하고 저장, 중복 저장 가능

Set : 순서를 유지X 저장, 중복 저장 X

Map : 키와 값의 쌍으로 저장, 키는 중복 저장 X

List 컬렉션

: 객체를 일렬로 늘어놓은 구조, 객체를 인덱스로 관리하기 때문에 객체를 저장하면 자동 인덱스가 부여되고 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공

객체 자체를 저장하는 것이 아닌, 객체의 번지를 참조한다.

(동일한 객체를 중복 저장할 수 있는데, 이 경우 동일한 번지를 참조한다.)

null도 저장이 가능한데, 이 경우 해당 인덱스는 객체를 참조하지 않는다.

메소드 종류

1. 객체추가

Boolean add(E e) : 주어진 객체를 맨 끝에 추가

Void add(int index, E element) : 주어진 인덱스에 객체를 추자

E set(int index, E element) : 주어진 인덱스에 저장된 객체를 주어진 객체로 바꾼다.

1. 객체 검색

Boolean contains(Object o) : 주어진 객체가 저장되어 있는 지 여부

E get(int index) : 주어진 인덱스에 저장된 객체를 리턴

Int size() : 저장되어 있는 전체 객체 수를 리턴

1. 객체 삭제

Void clear() : 저장된 모든 객체를 삭제

E remove(int index) : 주어진 인덱스에 저장된 객체를 삭제

Boolean remove(Object o) : 주어진 객체를 삭제

\* Vector

: ArrayList와 동일한 내부 구조를 가지고 있다 Vector를 생성하기 위해서는 저장할 객체 타입을 타입 파라미터로 표기하고 기본 생성자를 호출한다.

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

ArrayList와 다른 점은 동기화된 메소드로 구성되어 있기 때문에 멀티 스레드가 동시에 이 메소드들을 실행할 수 없고, 하나의 스레드가 실행을 완료해야 한다.(Thread safe List)

ArrayList와 LinkedList와의 차이점

: ArrayList가 순차적 추가/삭제 및 검색 측면에서 빠른데 반해, 중간 검색에서 LinkedList가 강점을 보인다.

Set컬렉션

: 순서와 중복이 없는 공간, 일종의 수학의 집합에 비유된다.

* 메소드

Boolean add(E e) : 주어진 객체를 저장, 객체가 성공적으로 저장되면 true, 실패 시 false

Boolean contains(Object o) : 주어진 객체가 저장되어 있는지 여부

Boolean isEmpty() : 컬렉션이 비어있는 지 조사

Iterator<E> iterator() : 저장된 객체를 한 번씩 가져오는 반복자 리턴

Int size() : 저장되어 있는 전체 객체 수 리턴

Void clear() : 모두 삭제

Boolean remove(Object o) : 주어진 객체 삭제

\*\*Iterator 메소드

: Iterator란 위의 Arraylist 등의 집합체로부터 정보를 얻어내는데 사용한다. 아래의 메소드들을 활용하여 어떤 컬랙션일지라도 동일한 방식으로 접근이 가능하여 그 안에 잇는 항목들에게 접근할 수 있다.(다형성)

Boolean hasNext() : 가져올 객체가 있으면 true를 리턴하고, 없으면 false를 리턴

E next() : 컬렉션에서 하나의 객체를 가져온다

Void remove() : Set 컬렉션에서 객체를 제거한다.

HashSet

: Set인터페이스의 구현 클래스이며,

포함하는 객체의 hashCode를 판별하여 같을 경우 한 쪽을 삭제한다.(주로 내부 데이터 종류가 같으면 hashcode도 같다)

Map 컬렉션

: 키와 값으로 구성된 Entry 객체를 저장하는 구조, 여기서 키와 값은 모두 객체이다

키는 중복 저장될 수 없다. 만약 기존에 저장된 키와 동일한 키로 값을 저장하면 기존의 값은 없어지고 새로운 값으로 대치된다.

* 메소드

V put(K key, V value) : 주어진 키로 값을 저장, 새로운 키면 null을 리턴, 동일한 키가 있으면 대치

Boolean containsKey(Object key) : 주어진 키가 있는 지 여부

Boolean containsValue(Object value) : 주어진 값이 있는 지 여부

Set<Map.Entry<K, V>> entrySet() : 키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set에 담아서 리턴

V get(Object key) : 주어진 키가 있는 값을 리턴

Boolean isEmpty() : 컬렉션이 비어 있는지 여부 검사

Set<K> keySet() : 모든 키를 Set 객체에 담아서 리턴

Int size() : 저장된 키의 총 수를 리턴

Collection<V> values() : 저장된 모든 값을 Collection에 담아서 리턴

Void clear() : 모든 Map.Entry<K,V>를 삭제

V remove(Object key) : 주어진 키와 일치하는 Map.Entry를 삭제하고 값을 리턴

HashMap

: hashCode()와 equals() 메소드를 재정의해서 동등 객체가 될 조건을 정해야 한다. 동등 객체의 조건은 hashCode()의 리턴값이 같아야 하고, equals() 메소드가 true를 리턴해야한다.

* Why?? : map에서 key를 바탕으로 값을 받아올 때, 해당하는 키와 동일여부를 검사하기 때문에 해당 코드를 작성해야 한다.

\*\*\*\* Equals와 hashCode를 같이 재정의 해야하는 이유!! \*\*\*\*\*

* 일반적으로 hashCode()를 재정의하지 않으면 디폴트 hashCode()가 계속해서 다른 해시코드를 반환한다.
* Equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다.(소위 말해 어떤 짓을 해도 실행 중인 동안에는 같은 값을 가져야 한다)
* Equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑 같은 값을 반환해야 한다.
* Equals(Object)가 두 객체를 다르게 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시 테이블의 성능이 좋아진다.

ㄴ> 이는, HashMap은 hashCode()를 이용하여, 해시 버킷에서 데이터를 꺼내온다. hashCode가 동일해도 논리적 동치가 아니기에 판단은 가능하지만, hashCode가 동일하기에 모든 데이터를 동일한 해시 버킷에 담고 꺼내기에 LinkedList와 다를게 없게 된다(비효율적)

ㄴ 참고링크

<https://donghyeon.dev/%EC%9D%B4%ED%8E%99%ED%8B%B0%EB%B8%8C%EC%9E%90%EB%B0%94/2021/01/09/eqauls%EC%99%80-hashCode%EB%A5%BC-%EA%B0%99%EC%9D%B4-%EC%9E%AC%EC%A0%95%EC%9D%98-%ED%95%B4%EC%95%BC%ED%95%98%EB%8A%94-%EC%9D%B4%EC%9C%A0/>

Hashtable

: HashMap과 동일한 내부 구조를 가지고 있다. Hashtable도 키로 사용할 객체는 hashCode() 및 equals() 메소드를 재정의하여 조건을 정해야 한다. HashMap 및 Hashtable의 차이점은 Hashtable은 동기화된 메소드로 구성되어 있기 때문에 멀티 스레드가 동시에 이 메소드를 실행할 수 없어 thread safe 환경을 만들 수 있다는 점에 있다.

Properties

: Properties는 Hashtable의 하위 클래스이기 때문에 Hashtable의 모든 특징을 그대로 가지고 있다.

차이점은 Hashtable은 키와 값을 다양한 타입으로 지정이 가능한데 비해 Properties는 키와 값을 String 타입을 제한한 컬렉션이다.

애플리케이션의 옵션 정보, 데이터베이스 연결 정보 그리고 국제화(다국어\_ 정보가 저장된 프로퍼티(~.properties)파일을 읽을 때 주로 사용한다.

프로퍼티 파일은 키와 값이 = 기호로 연결되어 있는 텍스트 파일로 ISO 8859-1 문자셋으로 저장된다. 이 문자셋으로 직접 표현할 수 없는 한글은 유니코드로 변환되어 저장된다.

프로퍼티 파일을 읽기 위해서는 Properties 객체를 생성하고, load() 메소드를 통해 호출하면된다. Load()메소드는 프로퍼티 파일로부터 데이터를 읽기 위해 FileReader 객체를 매개값으로 받는다.

프로퍼티 파일은 일반적으로 클래스 파일(~.class)과 함께 저장된다. 클래스 파일을 기준으로 상대 경로를 이용해 서 프로퍼티 파일의 경로를 얻으려면 Class의 getResource() 메소드를 이용하면 된다.(절대경로는 getPath())

String path = 클래스.class.getResource(“database.properties”).getPath();

path = URLDecoder.decode(path, “utf-8”); // 경로에 한글이 있을 경우 복원

Properties properties = new Properties();

Properties.load(new FileReader(path));

만약 다른 패키지에 있을 경우, 경로 구분자로 “/”를 사용한다.

String path = A.class.getResouce(“config/database.properties”).getPath();

Properties 객체에서 해당 키의 값을 읽으려면 getProperty() 메소드를 사용한다. 물론 Properties도 Map컬렉션이므로 get() 메소드로 값을 얻을 수 있다. But, get()메소드는 값을 Object 타입으로 리턴하므로 강제 타입 변환해서 String을 얻어야 하기 때문에 일반적으로 getProperty() 메소드를 사용한다.

String value = properties.getProperty(“key”);

검색기능을 강화시킨 컬렉션

: 이진 트리 구조, TreeSet, TreeMap

TreeSet, TreeMap 구조를 이용한 이진 트리 구조 생성이 가능하다(일종의 연결리스트)

Comparable과 Comparator

: TreeSet의 객체와 TreeMap의 키는 저장과 동시에 자동 오름차순으로 정렬되는데, 숫자 타입일 경우에는 값으로 정렬하고, 문자열 타입일 경우에는 유니코드로 정렬한다. TreeSet과 TreeMap은 정렬을 위해 java.lang.Comparable을 구현한 객체를 요구하는데, Integer, Double, String 모두 Comparable 인터페이스를 구현하고 있다. 사용자 정의 클래스도 Comparable을 구현한다면 자동 정렬이 가능하다.

Int compareTo(T o) : 주어진 객체와 같으면 0을 리턴, 주어진 객체보다 적으면 음수를 리턴, 크면 양수를 리턴

ㄴ> 해당 메소드를 오버라이딩 해주어야 정렬이 가능

TreeSet의 객체와 TreeMap의 키가 Comparable을 구현하고 있지 않을 격우에는 저장하는 순서간 ClassCastException이 발생한다. Comparable 비구현 객체를 정렬하는 방법은 TreeSet 또는 TreeMap 생성자의 매개값으로 정렬자(Comparator)를 제공하면 Comparable 비구현 객체도 정렬시킬 수 있다.

Int compare(T o1, T o2) o1과 o2가 동등하면 0을 리턴, o1이 o2보다 앞에 오게 하려면 음수를 리턴, 반대는 양수

동기화된 컬렉션

: 컬렉션 프레임워크의 대부분의 클래스들은 싱글 스레드 환경에서 사용할 수 있도록 설계되었다.

Vector와 Hashtable은동기화된 메소드로 구성되어 안전하게 처리할 수 있지만, 이외의 메소드들은 다르다.

비동화된 메소드를 동기화된 메소드로 래핑하는 Collections의 synchronizedXXX()메소드를 제공하고 있다.

List<T> synchronizedList(List<T> list)

Map<K, V> synchronizedMap(Map<K,V> m)

Set<T> synchonizedSet(Set<T> s)

병렬처리를 위한 컬렉션

: 동기화된 컬렉션은 멀티 스레드 환경에서 하나의 스레드가 요소를 안전하게 처리 하도록 도와주지만, 전체 요소를 빠르게 처리하지는 못한다.자바는 멀티 스레드가 컬렉션의 요소를 병렬적으로 처리할 수 있도록 특별한 컬렉션을 제공하고 있다.

Java.util.concurrent 패키지의 ConcurrentHashMap과 ConcurrentLinkedQueue이다.

ConcurrentHashMap을 사용하면 스레드에 안전하면서도 멀티 스레드가 요소를 병렬적으로 처리할 수 있다.

이는 부분(segment) 잠금을 사용하기 때문이다. 컬렉션에 10개의 요소가 저장되어 있을 경우, 1개를 처리할 동안 전체 10개의 요소를 다른 스레드가 처리하지 못하도록 하는 것이 전체 잠금이라면, 처리하는 요소가 포함된 부분만 잠금하고 나머지 부분은 다른 스레드가 변경할 수 있도록 하는 것이 부분 잠금이다.

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

ConcurrnetLinkedQueue는 락-프리 알고리즘을 구현한 컬렉션이다.

이는 여러 개의 스레드가 동시에 접근할 경우, 잠금을 사용하지 않고도 최소한 하나의 스레드가 안전하게 요소를 저장하거나 얻도록 해준다.

Queue<E> queue = new ConcurrnetLinkedQueue<E>();