Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java: Collections framework - 2/5b76d7b24227cf4f24a32d68

TUTORIAL

Java: Collections framework - 2

Topics

Set<E> Interfaces & Implementations

The Set<E> interface models a mathematical set, where no duplicate elements are allowed (e.g., playing cards). It may contain a single null element. It declares the following abstract methods. The insertion, deletion and inspection methods returns false if the operation fails, instead of throws an exception.

```
boolean add(E o)           // add the specified element if it is
not already present
boolean remove(Object o)   // remove the specified element if it
is present
boolean contains(Object o) // return true if it contains o

// Set operations
boolean addAll(Collection<? extends E> c) // Set union
boolean retainAll(Collection<?> c)        // Set intersection
```

The implementations of Set<E> interface include:

- **HashSet<E>:** Stores the elements in a hash table (hashed via the hashcode()). HashSet is is the best all-round implementation for Set.
- **LinkedHashSet<E>:** Stores the elements in a linked-list hash table for better efficiency in insertion and deletion. The element are

hashed via the hashCode() and arranged in the linked list according to the insertion-order.

- **TreeSet<E>:** Also implements sub-interfaces NavigableSet and SortedSet. Stores the elements in a red-black tree data structure, which are sorted and navigable. Efficient in search, add and remove operations (in O(log(n))).

For example, let's write a Book class, and create a Set of Book objects.

Java

```
import java.util.HashSet;
import java.util.Set;
class Book
{
  private int id;
  private String title;

  public Book(int id, String title) {
    this.id = id;      this.title = title;
  }

  public String toString() {
    return id + ": " + title;
  }

  // Two book are equal if they have the same id
  public boolean equals(Object o) {
    if (!(o instanceof Book)) {
      return false;
    }
    return this.id == ((Book)o).id;
  }
  // Consistent with equals(). Two objects which are equal have the same hash code.
  @Override
  public int hashCode() {
    return id;
  }
}

```

```
class Main
{
  public static void main(String[] args)
  {
    Book book1 = new Book(1, "Coding fun CodeQuotient");
    Book book2 = new Book(2, "Coding magic CodeQuotient");
    Book book2Dup = new Book(2, "Coding more magic CodeQuotient"); // same id as above
    Book book3 = new Book(3, "Learn Coding CodeQuotient");

    Set<Book> set1 = new HashSet<Book>();
    set1.add(book2);
    set1.add(book2Dup); // duplicate id, not added
    set1.add(book2);    // added twice, not added
    set1.add(book3);
    set1.add(null);     // Set can contain a null
    set1.add(null);     // but no duplicate
    set1.add(book1);
    System.out.println("Set1=" + set1);
    set1.remove(book1);
    set1.remove(book3);
    System.out.println("Set1 after removals =" + set1);

    Set<Book> set2 = new HashSet<Book>();
    set2.add(book3);
    System.out.println("Set2=" + set2);
    set2.addAll(set1);        // union with set1
    System.out.println("Set2 after Union with Set1=" + set2);

    set2.remove(null);
    System.out.println("Set2 after removing null= " + set2);
    set2.retainAll(set1);     // "intersection" with set1
    System.out.println("Set2 after intersection with Set1=" + set2);
  }
}
```

We need to provide an equals() method, so that the Set implementation can test for equality and duplication. In this

example, we choose the id as the distinguishing feature. We override equals() to return true if two books have the same id. We also override the hashCode() to be consistent with equals(). A Set cannot hold duplicate element. The elements are check for duplication via the overridden equal(). A Set can hold a null value as its element (but no duplicate too). The addAll() and retainAll() perform set union and set intersection operations, respectively. Also note that the arrangement of the elements is arbitrary, and does not correspond to the order of add().

LinkedHashSet<E>

Unlike HashSet, LinkedHashSet builds a link-list over the hash table for better efficiency in insertion and deletion (in the expense of more complex structure). It maintains its elements in the insertion-order (i.e., order of add()).

Java

```
class Main
{
  public static void main(String[] args)
  {
    Book book1 = new Book(1, "Coding fun CodeQuotient");
    Book book2 = new Book(2, "Coding magic CodeQuotient");
    Book book2Dup = new Book(2, "Coding more magic CodeQuotient"); // same id as above
    Book book3 = new Book(3, "Learn Coding CodeQuotient");

    Set<Book> set1 = new LinkedHashSet<Book>();
    set1.add(book2);
    set1.add(book2Dup); // duplicate id, not added
    set1.add(book2);    // added twice, not added
    set1.add(book3);
    set1.add(null);     // Set can contain a null
    set1.add(null);     // but no duplicate
    set1.add(book1);
    System.out.println("Set1=" + set1);
  }
}
```

NavigableSet<E> & SortedSet<E> Interfaces

Elements in the SortedSet<E> are sorted during add(), either using the natural ordering in the Comparable, or given a Comparator object. The NavigableSet<E> is a sub-interface of Set, which declares these additional navigation methods:

```
Iterator<E> descendingIterator()  // Returns an iterator over the
elements in this set in descending order.
Iterator<E> iterator()   // Returns an iterator over the elements
in this set, in ascending order.

// Per-element operation
E floor(E e)    // Returns the greatest element in this set less
than or equal to the given element, or null if there is no such
element.
E ceiling(E e)  // Returns the least element in this set greater
than or equal to the given element, or null if there is no such
element.
E lower(E e)    // Returns the greatest element in this set
strictly less than the given element, or null if there is no such
element.
E higher(E e)   // Returns the least element in this set strictly
greater than the given element, or null if there is no such
element.

// Subset operation
SortedSet<E> headSet(E toElement)            // Returns a view of
the portion of this set whose elements are strictly less than
toElement.
SortedSet<E> tailSet(E fromElement)          // Returns a view of
the portion of this set whose elements are greater than or equal
to fromElement.
SortedSet<E> subSet(E fromElement, E toElement)    // Returns a
view of the portion of this set whose elements range from
fromElement, inclusive, to toElement, exclusive.
```

Queue<E> Interfaces & Implementations

A queue is a collection whose elements are added and removed in a specific order, typically in a first-in-first-out (FIFO) manner. A deque (pronounced "deck") is a double-ended queue that elements can be inserted and removed at both ends (head and tail) of the queue.

Besides basic Collection<E> operations, Queue<E> provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operations). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations

```
// Insertion at the end of the queue
boolean add(E e)   // throws IllegalStateException if no space is
currently available
boolean offer(E e) // returns true if the element was added to
this queue, else false

// Extract element at the head of the queue
E remove()         // throws NoSuchElementException if this queue
is empty
E poll()           // returns the head of this queue, or null if
this queue is empty

// Inspection (retrieve the element at the head, but does not
remove)
E element()        // throws NoSuchElementException if this queue
is empty
E peek()           // returns the head of this queue, or null if
this queue is empty
```

Deque<E> declares additional methods to operate on both ends (head and tail) of the queue.

```
// Insertion
void addFirst(E e)
void addLast(E e)
boolean offerFirst(E e)
boolean offerLast(E e)

// Retrieve and Remove
E removeFirst()
E removeLast()
E pollFirst()
E pollLast()

// Retrieve but does not remove
E getFirst()
E getLast()
```

```
E peekFirst()
E peekLast()
```

A Deque can be used as FIFO queue (via methods add(e), remove(), element(), offer(e), poll(), peek()) or LIFO queue (via methods push(e), pop(), peek()).

The Queue<E> and Deque<E> implementations include:

- **PriorityQueue<E>:** A queue where the elements are ordered based on an ordering you specify, instead of FIFO.
- **ArrayDeque<E>:** A queue and deque implemented as a dynamic array, similar to ArrayList<E>.
- **LinkedList<E>:** The LinkedList<E> also implements the Queue<E> and Deque<E> interfaces, in additional to List<E> interface, providing a queue or deque that is implemented as a double-linked list data structure.

The basic operations of Queue<E> include adding an element, polling the queue to retrieve the next element, or peeking at the queue to see if there is an element available in the queue. The Deque<E> operations are similar except element can be added, polled, or peeked at both ends of the deque.

Map<K,V> Interfaces & Implementations

A map is a collection of key-value pairs (e.g., name-address, name-phone, isbn-title, word-count). Each key maps to one and only value. Duplicate keys are not allowed, but duplicate values are allowed. Maps are similar to linear arrays, except that an array uses an integer key to index and access its elements; whereas a map uses any arbitrary key (such as Strings or any objects). The Map<K,V> interface declares the following abstract methods:

```
V get(Object key)      // Returns the value of the specified key
V put(K key, V value)  // Associate the specified value with the
specified key
boolean containsKey(Object key)  // Is this map has specified key?
boolean containsValue(Object value)

// Views
Set<K> keySet()         // Returns a set view of the keys
```

```
Collection<V> values()  // Returns a collection view of the values
Set entrySet()          // Returns a set view of the key-value
```

The implementations of Map<K,V> interface include:

- **HashMap<K,V>:** Hash table implementation of the Map<K,V> interface. The best all-around implementation. Methods in HashMap is not synchronized.
- **TreeMap<K,V>:** Red-black tree implementation of the SortedMap<K,V> interface.
- **LinkedHashMap<K,V>:** Hash table with link-list to facilitate insertion and deletion.
- **Hashtable<K,V>:** Retrofitted legacy (JDK 1.0) implementations. A synchronized hash table implementation of the Map<K,V> interface that does not allow null key or values, with legacy methods.

For example,

Java

```
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;
import java.io.File;

class Main
{
  public static void main(String[] args) throws Exception
  {
    String f1 = "Code Quotient is to learn Code to solve problems";
    Scanner in=new Scanner(f1);

    Map<String, Integer> map = new HashMap<String, Integer>(); // For storing word count
    while (in.hasNext())
    {
      String word = in.next();
      int freq = (map.get(word) == null) ? 1 : map.get(word) + 1;      // Count frequency
      map.put(word, freq);
    }
    System.out.println(map); // print map
```

```
21     }
22 }
```