08 Java Collections Framework

Instructor: Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

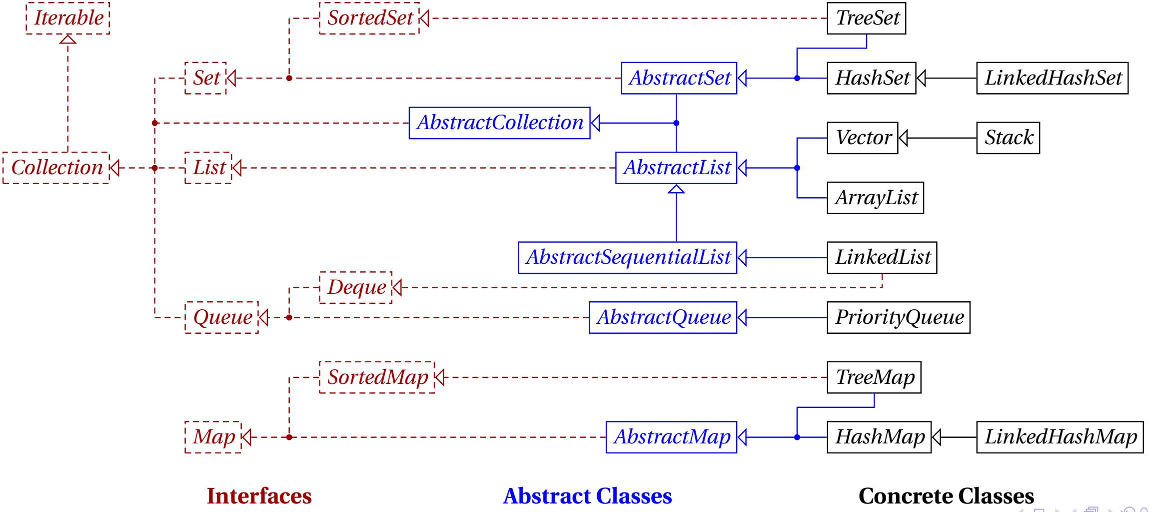
October 17, 2019

Outline

Java Collections Framework

- A collection is a container object that represents a group of inner objects, often referred to as *elements*.
- Some collections allow duplicate elements and others do not.
- Some collections are ordered and others are unordered.
- A map (also *associative array*) represents a group of objects, each of which is associated with a key. You can get the object from a map by a key, and you must use a key to put the object into the map.
- The Java Collections Framework supports three types of collections, namely *sets*, *lists*, and *maps*, grouped in the $java.util$ package.

Java Collections Framework Hierarchy



The *Collection*⟨*E*⟩ Interface

- *clear*() — removes all elements from `this`.
- *isEmpty*() — checks if `this` contains no elements.
- *size*() — returns the number of elements in `this`.
- *add*(e) — adds element e to `this`, returns if `this` has been modified.
- *addAll*(c) — adds all of the elements in collection c to `this`.
- *contains*(o) — checks if `this` contains an element that *equals* object o.
- *containsAll*(c) — checks if `this` contains all of the elements in collection c.
- *remove*(o) — removes one element that *equals* object o from `this`, if present.
- *removeAll*(c) — removes all the elements of `this` that are also contained in collection c.
- *retainAll*(c) — retains only the elements of `this` that are also contained in collection c.

The *AbstractCollection*$\langle E \rangle$ Class

- The *AbstractCollection* class is a convenience class that provides partial implementation for the *Collection* interface.
- *AbstractCollection* implements all the methods in *Collection* except the *size* and *iterator* methods. These are implemented in appropriate subclasses.
- To implement an unmodifiable collection, the programmer needs only to extend *AbstractCollection* and provide implementations for the *iterator* and *size* methods. The iterator returned by the *iterator* method must implement *hasNext* and *next*.
- To implement a modifiable collection, the programmer must additionally override the *add* method, and the iterator returned by the *iterator* method must additionally implement its *remove* method.

Implementing a String Collection with Fixed Capacity

```
public class StrColl extends AbstractCollection<String> {
    private String[] a;
    private int n;
    public StrColl(int capa) { a = new String[capa]; n = 0; }
    @Override public int size() { return n; }
    @Override public boolean add(String e) { a[n++] = e; return true; }
    @Override public Iterator<String> iterator() {
        return new Iterator<String>() {
            int i = n;
            @Override public boolean hasNext() { return i > 0; }
            @Override public String next() { return a[--i]; }
        };
    }
}
```

The $Set\langle E\rangle$ Interface and the $AbstractSet\langle E\rangle$ Class

- The *Set* interface extends the *Collection* interface.
- It does *not* introduce new methods or constants, but it stipulates that an instance of *Set* contains no duplicate elements.
- An implementation of *Set* must ensure that no duplicate elements (by *equals*) can be added to the set.

```
@Override public boolean add(String e) {
    if ( contains(e) ) return false;
    else { a[n++] = e; return true; }
}
```

- The *AbstractSet* class is a convenience class that extends *AbstractCollection* and implements *Set*.
- The *AbstractSet* class provides concrete implementations for the *equals* and *hashCode* methods on sets.
- A concrete set class based on *AbstractSet* must implement *add*, *size* and *iterator*.

The *HashSet⟨E⟩* Class

- The *HashSet* class is a concrete class that implements *Set*. It can be used to store duplicate-free elements.
- For efficiency, objects added to a hash set need to implement the *hashCode* method in a manner that properly disperses the hash code.

```java
Set<String> set = new HashSet<String>(); // create a hash set
set.add("London"); // add strings to the set
set.add("Paris");
set.add("New York");
System.out.println(set); // display the hash set as an object
for ( String elem : set ) // display the elements in the hash set
    System.out.print(elem + " ");
```

The *LinkedHashSet*⟨ *E* ⟩ Class

- *LinkedHashSet* differs from *HashSet* in that it maintains a doubly-linked list running through all of its entries.
- The linked list defines the iteration ordering, which is the order in which elements were inserted into the set.

```
Set<String> set = new LinkedHashSet<String>();
set.add("London");
set.add("Paris");
set.add("New York");
for ( String elem: set )
    System.out.print(elem + " ");
```

- The above code prints “London”, “Paris” and “New York” in their insertion order.

The *SortedSet*⟨*E*⟩ Interface and the *TreeSet*⟨*E*⟩ Class

- *SortedSet* is a subinterface of *Set*, which guarantees that the elements in the set are sorted. *TreeSet* is a concrete class that implements the *SortedSet* interface.
- You can use an iterator to traverse the elements in the sorted order.
- The elements can be sorted in two ways.
- One way is to use the *Comparable*⟨*E*⟩ interface.
- The other way is to specify a *comparator* for the elements in the set if
    1. the class for the elements does not implement the *Comparable*⟨*E*⟩ interface, or
    2. you don't want to use the *compareTo* method in the class that implements the *Comparable*⟨*E*⟩ interface. This approach is referred to as *order by comparator*.

Example: Using *TreeSet* to Sort Elements in a Set

- All the collection classes have at least two constructors.
- One is the default constructor that constructs an empty collection.
- The other copies elements from another collection.

```
Set<String> set = new HashSet<String>();
set.add("London");        set.add("Paris");
set.add("New York");      set.add("San Francisco");
set.add("Beijing");       set.add("New York");

TreeSet<String> treeSet = new TreeSet<String>(set);
System.out.println(treeSet);
```

- The above code prints: `[Beijing, London, New York, Paris, San Francisco]`

The $List\langle E\rangle$ Interface

A list not only stores duplicate elements, but also associates each element with a position (*index*). The user can access an element e by its index i.

- $get(i)$ — returns the element at index i in `this`.
- $set(i,\ e)$ — replaces the element at index i in `this` with element e.
- $indexOf(o)$ — returns the index of the first element (with the lowest index) that *equals* o in `this`, or -1 if the element is not found.
- $lastIndexOf(o)$ — returns the index of the last element (with the highest index) that *equals* o in `this`, or -1 if the element is not found.
- $listIterator()$ — returns a list iterator over the elements in `this`.
- $listIterator(i)$ — returns a list iterator over the elements in `this`, starting at index i.
- $subList(fromIdx,\ toIdx)$ — returns a view of the portion of `this` (as a sub-list) between the specified *fromIdx*, inclusive, and *toIdx*, exclusive.

The *AbstractList*$\langle E \rangle$ and *AbstractSequentialList*$\langle E \rangle$

These classes provide partial implementations of the *List* interface.

- An implementation backed by a "random access" data store (such as an array) can be based on *AbstractList*.
- For sequential access data (such as a linked list), *AbstractSequentialList* should be used in preference to this class.
- To implement an unmodifiable list, we need to override the *get*(i) and *size* methods.
- To implement a modifiable list, we must also override the *set*(i, e), *add*(i, e) and *remove*(i) methods.
- Unlike the other abstract collection implementations, we don't have to provide an iterator implementation.

ArrayList$\langle E \rangle$ and *LinkedList*$\langle E \rangle$

- The *ArrayList* class and the *LinkedList* class are concrete implementations of the *List* interface.
- If you need to support random access through an index without inserting or removing elements from any place other than the end, *ArrayList* offers the most efficient collection.
- If, however, you require the insertion or deletion of elements from any place in the list, you should choose *LinkedList*.
- A list can grow or shrink dynamically, while an array is fixed once it is created.
- If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

Additional Methods in *ArrayList*⟨*E*⟩ and *LinkedList*⟨*E*⟩

An *ArrayList* has a capacity, which can be expanded or shrinked when needed.

- *ArrayList*(*initialCapacity*) — constructs an empty list with the specified initial capacity.
- *trimToSize*() — trims the capacity of `this` to be the list's current size.

A *LinkedList* is efficient when insertions and deletions are frequent.

- *addFirst*(*e*) — inserts element *e* the beginning of `this`.
- *addLast*(*e*) — appends element *e* to the end of `this`.
- *getFirst*() — returns the first element in `this`.
- *getLast*() — returns the last element in `this`.
- *removeFirst*() — removes and returns the first element from `this`.
- *removeLast*() — removes and returns the last element from `this`.

Example: Using *ArrayList* and *LinkedList*

- This example creates an array list filled with numbers, and inserts new elements into the specified location in the list.
- The example also creates a linked list from the array list, inserts and removes the elements from the list.
- Finally, the example traverses the list forward and backward.

```
List<Integer> arrayList = new ArrayList<Integer>();
arrayList.add(1); arrayList.add(2);
arrayList.add(3); arrayList.add(1);
arrayList.add(4); arrayList.add(0, 10);
arrayList.add(3, 30);
// Display the the array list
System.out.println(arrayList);                                    // ...
```

Example: Using *ArrayList* and *LinkedList* (2)

```java
LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
linkedList.add(1, "red");
linkedList.removeLast();
linkedList.addFirst("green");
// Display the linked list forward
ListIterator listIterator = linkedList.listIterator();
while (listIterator.hasNext())
    System.out.print(listIterator.next() + " ");
System.out.println();
// Display the linked list backward
listIterator = linkedList.listIterator(linkedList.size());
while (listIterator.hasPrevious())
    System.out.print(listIterator.previous() + " ");
```

The $Map\langle K, V\rangle$ Interface

The *Map* interface maps keys to the elements. The keys are like indices. In lists, the indices are integers. In maps, the keys can be any objects.

- *containsKey*(*key*) — checks if `this` contains a mapping for object *key*.
- *entrySet*( ) — returns a *Set* view of the mappings contained in `this`.
- *get*(*key*) — returns the value to which object *key* is mapped, or null if `this` contains no mapping for the key.
- *keySet*( ) — returns a *Set* view of the keys contained in `this`.
- *put*(*key*, *value*) — associates the *value* with the *key* in `this`.
- *putAll*(*m*) — copies all of the mappings from map *m* to `this`.
- *remove*(*key*) — removes the mapping for object *key* from `this`, if it is present.
- *values*( ) — returns a *Collection* view of the values contained in `this`.

HashMap$\langle K,V \rangle$ and *TreeMap*$\langle K,V \rangle$

- The *HashMap* and *TreeMap* classes are two concrete implementations of the *Map* interface.
- The *HashMap* class is efficient for locating a value, inserting a mapping, and deleting a mapping.
- The *TreeMap* class, implementing *SortedMap*, is efficient for traversing the keys in a sorted order.

LinkedHashMap⟨K,V⟩

- The *LinkedHashMap* class extends *HashMap* with a linked list implementation that supports an ordering of the entries in the map.
- The entries in a *HashMap* are not ordered, but the entries in a *LinkedHashMap* can be retrieved in the order in which they were inserted into the map (known as the insertion order), or the order in which they were last accessed, from least recently accessed to most recently (access order).
- The default constructor constructs a *LinkedHashMap* with the insertion order.
- To construct a *LinkedHashMap* with the access order, use the *LinkedHashMap*(*initialCapacity*, *loadFactor*, `true`).

Example: Using *HashMap* and *TreeMap*

- This example creates a hash map that maps borrowers to mortgages.
- The program first creates a hash map with the borrower's name as its key and mortgage as its value.
- The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

```
Map<String,Integer> hashMap = new HashMap<String,Integer>();
                                   // The type arguments can be omitted here.
hashMap.put("Smith", 30); hashMap.put("Anderson", 31);
hashMap.put("Lewis", 29); hashMap.put("Cook", 29);
// Display entries in HashMap
System.out.println(hashMap);                                              // ...
```

Example: Using *HashMap* and *TreeMap* (2)

```java
Map<String,Integer> treeMap = new TreeMap<>(hashMap);
// Display entries in ascending order of key
System.out.println(treeMap);
Map<String,Integer> linkedHashMap = new LinkedHashMap<>(16,0.75f, true);
linkedHashMap.put("Smith", 30);
linkedHashMap.put("Anderson", 31);
linkedHashMap.put("Lewis", 29);
linkedHashMap.put("Cook", 29);
System.out.println("The age for " + "Lewis is " +
                        linkedHashMap.get("Lewis").intValue());
// Display entries in LinkedHashMap
System.out.println(linkedHashMap);
```

Practice: Implementing a List to Interleave Two Arrays

Let a, b be two integer arrays. We construct a read only wrapper list based on the *AbstractList* class to alternatively return the elements from a and b (interleaving). For example, if $a = [1, 2, 3, 4]$, $b = [-1, -2, -3, -4, -5, -6]$, the interleaving of a, b is

$$[1, -1, 2, -2, 3, -3, 4, -4, -5, -6].$$

1. Define a class that extends *AbstractList*.
2. Define two array fields to record the arrays, and a constructor to initialize the fields.
3. Override the *get*(i) and *size* methods to operate on the arrays.
4. Design a test case to test your implementation.
5. Zip your source files, including `Interleave.java` and `Test.java` into `Interleave.zip`.
6. Upload the `Interleave.zip`.