COMP6700/2140 Abstract Data Types: Queue, Set, Map

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

19 April 2017

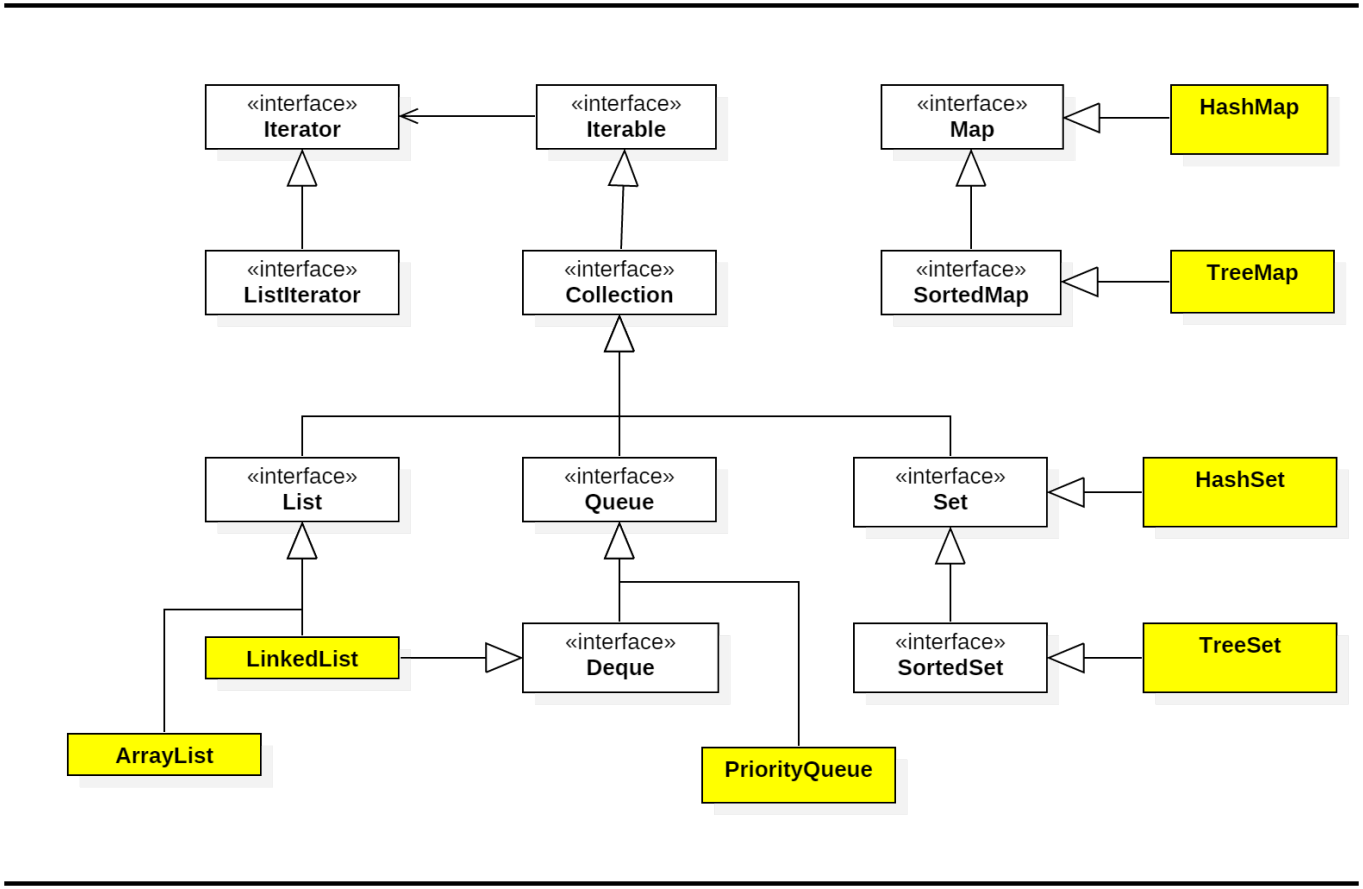
Java Collection Interfaces

The Java Collections Framework (JCF) includes interfaces (ADTs), implementations (data structures), and algorithms. The main interfaces are:

- Iterable — most basic interface to use a data type for iteration and nothing else
- Collection (proper) — allows adding, removing and testing for elements
- List — collection whose elements are ordered and accessible by their location in the collection
- Set — a collection with no duplicates
- SortedSet — an ordered collection which contains no duplicates
- Queue — a collection where elements are removed according to some order (typically first-in, first-out (FIFO))
- Deque — a double-ended queue
- Map — a collection where elements are stored and retrieved not by an index, but via a key

Collections: Class Diagram

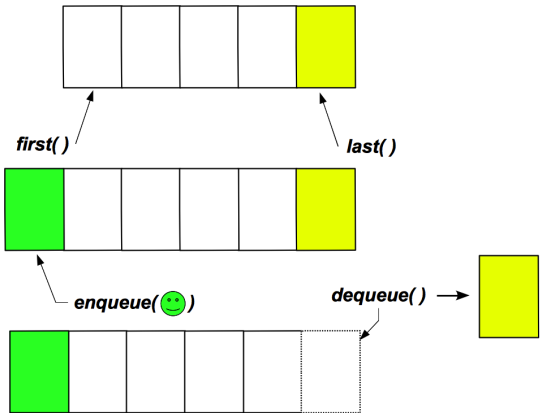
Key interfaces of the Java Collections Framework and *some* implementations:



Queues

A *Queue* is a "first-in, first-out" (FIFO) type which can be implemented by adding objects to the head of a list and removing them from its tail. The interface:

- `int size();`
- `boolean isEmpty();`
- `E first(E element);`
- `E last(E element);`
- `void enqueue(E newEl);`
- `void dequeue();`
- `String toString();`



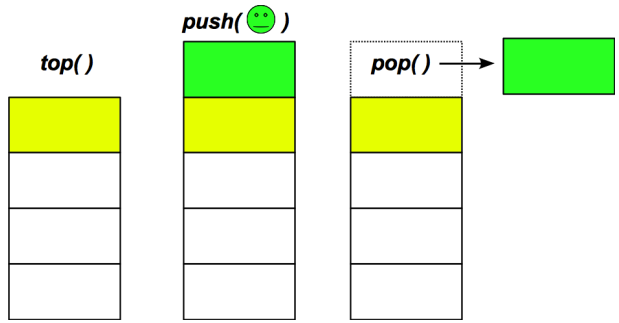
The JCF `Queue` interface has slightly different names for the above operations, as well as allowing different orderings (defined by the constructor with `Comparator` parameter), for instance, in the `PriorityQueue` class.

Stacks

Queues and stacks are widely used in systems level programming (for managing memory and processes) and other applications.

A *Stack* is a "last-in, first-out" (LIFO) collection type which can be implemented by adding to and removing from the head of a list. The interface:

- `int size();`
- `boolean isEmpty();`
- `int search(E el);// similar to indexOf()`
- `E top(E el); // JCF calls it peek()`
- `void push(E newEl);`
- `void pop();`
- `String toString();`

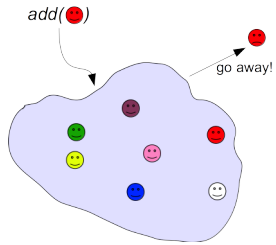


There is a legacy `Stack` class in JCF, however, the recommended implementation is the `Deque` class. `Deque` implements `addFirst`, `removeFirst` and `peekFirst` methods.

Set and *SortedSet*

The extensions of *Collection* type which disallow duplicates (identical elements): a repeated invocation of `add(elem)` with the same element `elem` (or such `elem1`, that `elem.equals(elem1)` returns `true`) returns `false`, and the collection remains unchanged. *Set* types, therefore, are sets in the mathematical sense (some computer scientists call them "bags"). The elements of a set are unordered. The subtype *SortedSet* (extension of the *Set* interface) represents a collection, whose elements can be *compared* — they implement *Comparable* interface. The ordering allows to introduce additional methods:

- `comparator()` — returns the comparator used with this sorted set (`null` in case of *natural ordering*)
- `first()`, `last()` — smallest and largest elements
- `subset(eMin,eMax)`, `headSet(eMax)`, `tailSet(eMin)` — subsets of elements in given ranges

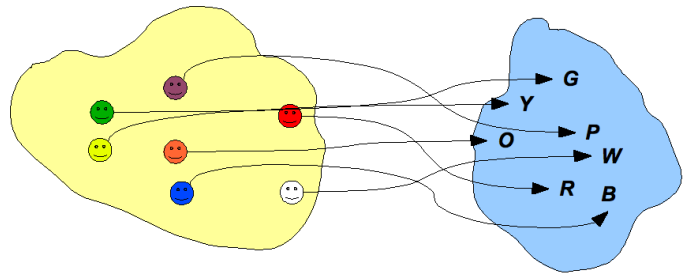


Set is implemented by the *HashSet* class with a hash table as the DS. Content modification (`add`, `set`) and testing (`contains`) are $\mathcal{O}(1)$ operations. *SortedSet* is implemented by the *TreeSet* class with a binary tree as the DS. If the implementation can maintain the *balanced tree*, the search and modify operations are $\mathcal{O}(\log_2 N)$. Example — SetTest.java.

Map and *SortedMap*

Unlike all previous types from JCF, the *Map* family of types does not extends the *Collection* interface: their contract is different — to represent not a collection of elements, but a *correspondence between two collections*. So, a map contains **key/value pairs**, with *no duplicate keys* (the keys form a set) and *at most one value* for each key. *Map* is a model of a mathematical abstraction called *function*. The interface Map<K,V> operations are:

- `V put(K key, V value)`
- `V get(K key)` and `V remove(K key)`
- `boolean containsKey(K key)`
- `boolean containsValue(V value)`
- `public Set keySet()` — returns a **view** of the set of keys
- `public Collection values()` — returns a **view** of the collection of stored values



Map of Smiley to Character

The *SortedMap* extension requires the set of keys be sorted. Methods like `firstKey()` and `lastKey()` are added. The *Map* interface has two major implementation classes (similar to *Set*) — *HashMap* which uses a hash table DS for the implementation (with similar $\mathcal{O}(1)$ performance for `put`/`get` operations), and *TreeMap* which implements *SortedMap* in the similar to *TreeSet* way (with $\mathcal{O}(\log_2 N)$ efficiency). Example — MapTest.java.

Bulk operations of *Collection* interface

The basic *Collection* operations were listed above (slide *Collection* Interface). They allow to examine and manipulate the collection element by element. The **bulk operations** allow to manipulate the whole part of the collection in one go:

- `boolean containsAll(Collection<?> c);`
- `boolean addAll(Collection<? extends E> c);` — adds all elements from `c` to `this`
- `boolean retainAll(Collection<?> c);` — retains only those elements that are found in `c`
- `boolean removeAll(Collection<?> c);` — opposite to `retainAll()`
- `List<E> subList(int, int)` — like a `String.substring(int,int)`, figure out the rest...
- `void clear();` — removes all elements ("clean start")

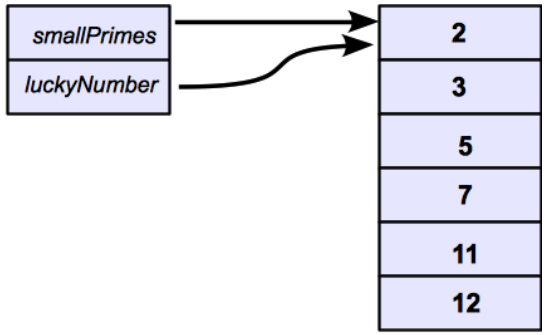
Another category of operations are **array operations**:

- `Object[ ] toArray();` — returns an array of all the elements in this list in the correct order
- `<T> T[ ] toArray(T[ ] dest);` — the list elements are placed in the array `dest`, which is returned (if `dest` isn't big enough to accommodate all elements of the collection, a new, properly sized array is created and returned)

How to copy arrays properly

Arrays are objects, therefore when one array is assigned to another, the two identifiers point to the *same memory location*. How to copy elements of an array into a *different* array?

```
int[] smallPrimes = {2,3,5,7,11,12};
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12;
System.out.println(smallPrimes[5]);// Prints 12
```



To achieve *element-by-element* copying, one has to use the `System.arraycopy()`method: To copy `count` elements from `source` array starting with index `from` to `target` array beginning with index `to`, make the following call: `System.arraycopy(source,from,target,to,count);`

```
int[] smallPrimes = {2,3,5,7,11,13};
int[] luckyNumbers = {1001,1002,1003,1004,1005,1006,1007};
System.arraycopy(smallPrimes,2,luckyNumbers,3,4);
for (int i=0; i<luckyNumbers.length; i++)
    System.out.println(i + ":" + luckyNumbers[i]);
```

Further Reading

- Hortsmann *Core Java for the Impatient*, Ch. 7.3–7.5
- Oracle *The Java Tutorials*: Collections (Sections 1-3)