Algorithms and Data Structures

The Collection Framework in Java

Dr. Bernhard Anrig

HS 2012/13

Bibliography: java.oracle.com, Collections Framework
http://download.oracle.com/javase/tutorial/collections/

Outline

Outline

Java 1.4.2 — Java 1.5–7

Major difference in Java 1.5–7
with respect to the collection framwork:

`Collections are Generics`

In the sequel: **No focus on generics!**

Simple Example of Using Generics

Generic stack:

```
Stack<String> s = new Stack<String>();
s.push("hi");
String greeting = s.pop(); //no cast required here
```

Non-generic stack:

```
Stack s = new Stack();
s.push("hi");
String greeting = (String)s.pop(); //cast required here
```

Learn to write generic classe: yourself!

Outline

Need for efficient Data Structures in Java

Defaults of "Vectors":

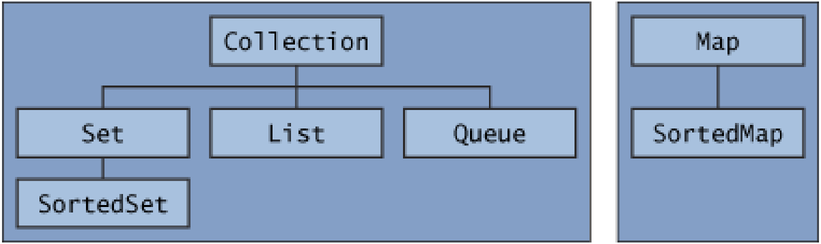
- Growable array
- Not designed for its efficiency
- Complexity hard to evaluate

Need for efficient Data Structures in Java

Replaced by a coherent Framework

- *Collection:* a group of objects, duplicates allowed;
  1. *List:* extends Collection
     allows duplicates and introduces positional indexing
  2. *Set:* extends Collection but forbids duplicates;
     - *SortedSet*
  3. *Queue:* extends Collection, ordered elements
     (for example as FIFO)
- *Map:* Extends neither Set nor Collection
  1. *SortedMap*

Hierarchy Overview



Outline

The Collection Interface

Goal: Represent any group of objects or elements.

```
public interface Collection<E> extends Iterable<E> {
   // Basic Operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element); // Optional
   boolean remove(Object element); // Optional
   Iterator<E> iterator();
```

Methods (Cont'd)

```
    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c); // Optional
    boolean retainAll(Collection<?> c); // Optional
    void clear(); // Optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Focus on some methods:

- The `ContainsAll` method allows you to test if the current collection contains all the elements of another collection, a *subset*.
- The `addAll` method ensures all elements from another collection are added to the current collection, usually a `union`.
- The `clear` method removes all elements from the current collection.
- The `removeAll` method is like `clear` but only removes a subset of elements.
- The `retainAll` method is similar to the `removeAll` method, but does what might be perceived as the opposite. Usually called `intersection`.

Outline

Iterator

With the `Iterator` interface, you can traverse a collection from the beginning to end and safely remove elements from the underlying `collection`.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional -> !!!
}
```

http://download.oracle.com/javase/7/docs/api/java/util/Iterator.html

Example

```
Collection<String> collection = ...;

Iterator<String> iterator = collection.iterator();

while (iterator.hasNext()){
    String element = iterator.next();
    if (removalCheck(element)){
        iterator.remove();
    }
}
```

"For-each" Construct

Simple and easy way to traverse a collection:

```
for (Object o : collection)
    System.out.println(o);
```

Example:

```
Collection<String> x = new HashSet<String>();
x.add("1");
x.add("2");
x.add("3");
for (String s : x)
    System.out.println(s);
```

Iterators

Different behaviour of iterators:

- "Snapshot iterator"
- "Fail fast iterator"
- Iterators that "follow" changes in the underlying structure

Outline

The Set interface

- The `Set` interface extends the `Collection` interface and forbids duplicates within the collection.
- Very useful as we need not to worry about duplicates!
- All original methods are present and no new methods are introduced.
- The concrete `set` implementation classes rely on the `equals()` method of the object added to check for equality.

HashSet and TreeSet Classes

- The Collection Framework provides three general-purpose implementations of the `Set` interface.
- The `HashSet` class will be used for storing duplicate-free collection.
- For efficiency, objects added to a `HashSet` need to implement the `hashCode()` method in a manner that properly distributes the hash codes.
- The `TreeSet` class is useful when you need to extract elements from a collection is a sorted manner.
- Elements added to a `TreeSet` must be sortable.
- `LinkedHashSet` somewhat in-between the two above

Set Usage Example

```
import java.util.*;
public class SetExample{
  public static void main(String args[]){
    Set<String> set = new HashSet<String>();
    set.add("Christian");
    set.add("Marion");
    set.add("Daniel");
    set.add("Marta");
    set.add("Bruno");
    set.add("Marion");
    System.out.println(set);
    Set<String> sortedSet = new TreeSet<String>(set);
    System.out.println(sortedSet);
  }
```

Set Usage Example (Cont'd)

Running the program produces the following output.

```
java SetExample

[Marta, Bruno, Marion, Daniel, Christian]
[Bruno, Christian, Daniel, Marion, Marta]
```

Outline

The List interface

- The `List` interface extends the `Collection` interface to define an ordered collection, permitting duplicates.
- The interface adds position-oriented operations, as well as the ability to work with just a part of the list.

Methods

```
public interface List<E> extends Collection<E> {

    // Positional Access
    E get(int index);
    E set(int index, E element); // Optional
    boolean add(E element); // Optional
    void add(int index, E element); // Optional
    E remove(int index); // Optional
    boolean addAll(int index,
        Collection<? extends E> c); //Optional
```

Methods (Cont'd)

```
    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Lists Methods

- `E get(int index)`
  Returns the element at the specified position in this list.
- `E set(int index, E element)`
  Replaces the element at the specified position in this list with the specified element (optional operation).
- `void add(int index, E element)`
  Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Lists Methods (Cont'd)

- `E remove(int index)`
  Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.
- `List<E> subList(int from, int to)`
  Returns a view of the portion of this list between the specified `from`, inclusive, and `to`, exclusive. (If `from` and `to` are equal, the returned list is empty.)
  The returned list is backed by this list, so changes in the returned list are reflected in this list, and vice-versa. The returned list supports all of the optional list operations supported by this list.

The `ListIterator` interface

- The `ListIterator` interface extends the `Iterator` interface to support bi-directional access, as well as adding or changing elements in the underlying collection.
- Normally, one does not use a `ListIterator` to alternate between going forward and backward in one iteration through the elements of a collection.

Methods

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); // Optional
    void set(E o); // Optional
    void add(E o); // Optional
}
```

The `ListIterator` interface (Cont'd)

- `void add(E o)` Inserts the specified element into the list (optional operation). The element is inserted immediately before the next element that would be returned by next, if any, and after the next element that would be returned by previous, if any. (If the list contains no elements, the new element becomes the sole element on the list.)
  The new element is inserted before the implicit cursor: a subsequent call to next would be unaffected, and a subsequent call to previous would return the new element. (This call increases by one the value that would be returned by a call to nextIndex or previousIndex.)

The `ListIterator` interface (Cont'd)

- `void remove()`
  Removes from the list the last element that was returned by next or previous (optional operation). This call can only be made once per call to next or previous. It can be made only if ListIterator.add has not been called after the last call to next or previous.
- `void set(E o)`
  Replaces the last element returned by next or previous with the specified element (optional operation). This call can be made only if neither ListIterator.remove nor ListIterator.add have been called after the last call to next or previous.

Example

```
List<String> list = ...;

ListIterator<String> iterator
        = list.listIterator(list.size());

while (iterator.hasPrevious()){
    String element = iterator.previous();
    // Process element
}
```

ArrayList class and LinkedList class

There are two general-purpose `List` implementations in the Collections Framework: `ArrayList` and `LinkedList`.

- `ArrayList`: The underlying data structure is an Array. Thus, any insertion requires a shift of the elements.
  And when the array is full, one need a copy of the array.
- `LinkedList`: The underlying data structure is a doubly linked list. Thus it is inefficient to access to elements using their index. It is much more efficient to use a `ListIterator`.

LinkedList class

Additional methods available in `LinkedList`:

```
void addFirst(E o)
void addLast(E o)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

etc.

ArrayList and LinkedList (Cont)

Complexity with:	*ArrayList*	*LinkedList*
`size`, `isEmpty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`add(E o)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove(E o)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`addAll(COLL c)`	$\mathcal{O}(m)$	$\mathcal{O}(m)$
`removeAll(COLL c)`	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(n \cdot m)$
`retainAll(COLL c)`	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(n \cdot m)$
`clear`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`addAll(int index, COLL c))`	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$
`get(int index)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`set(int index, E elem)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`add(int index, E elem)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`remove(int index)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`indexOf(Object o)`, `lastIndexOf`	$\mathcal{O}(n)$	$\mathcal{O}(n)$

ArrayList and LinkedList (Cont)

Complexity with:	*ArrayList*	*LinkedList*
`iterator`, `listIterator`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`listIterator(int index)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`sublist(int i, int j)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`addFirst(E o)`, `addLast(E o)`	n.a.	$\mathcal{O}(1)$
`getFirst`, `getLast`	n.a.	$\mathcal{O}(1)$
`removeFirst`, `removeLast`	n.a.	$\mathcal{O}(1)$

where $n =$ size of `this` and m size of the collection argument)

COLL = `Collection<? extends E>`

n.a.: not available

Lists: Example

```
import java.util.*;

public class ListExample{
  public static void main(String args[]){
    List<String> list = new ArrayList<String>();
    list.add ("A"); list.add ("B");
    list.add ("C"); list.add ("D");
    list.add ("E");
    System.out.println(list);
    System.out.println("2: "+ list.get(2));
    System.out.println("0: "+ list.get(0));
```

Lists: Example (Cont'd)

```
    LinkedList<String> queue = new LinkedList<String>();
    queue.addFirst("AA");
    queue.addFirst("BB");
    queue.addFirst("CC");
    queue.addFirst("DD");
    queue.addFirst("EE");
    queue.addFirst("FF");
    System.out.println(queue);
    queue.removeLast();
    queue.removeLast();
    System.out.println(queue);
}}
```

Lists: Example (Cont'd)

```
/* Output:
[A, B, C, D, E]
2: C
0: A
[FF, EE, DD, CC, BB, AA]
[FF, EE, DD, CC]
*/
```

Outline

The Queue interface

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E o);
    E peek();
    E poll();
    E remove();
}
```

Queue methods have two different forms:

	Throws exception	Returns null if queue is empty
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

PriorityQueue class

- Implementation based on a heap structure
- ordered according to order specified at construction time (see below)

Constructors

- `PriorityQueue()`
  Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering (using Comparable).
- `PriorityQueue(int initialCapacity)`
  Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering (using Comparable).
- ```
  PriorityQueue(int initialCapacity,
      Comparator<? super E> comparator)
  ```
  Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
- etc.
  ```

Outline

Limitations of Lists in the Collection Framework

- A linked list iterator allows the user to work on a list like positions on a Sequence.
- But a linkedList allows only ONE iterator to change the content of the list. All other iterators are disabled when a modification is done.
- One can not store a `ListIterator` for a future use.
- The structure presented in the previous chapter "Sequence" is still interesting. It allows two pointers to simultaneously point on the same linked list.

Limitations of Queues in the Collection Framework

- No “direct” access using structures like positions seen in class
- Orderings cannot be changed (in standard implementation), i.e. Comparators cannot be exchanged

Outline

Threads

- Collection classes are *not* thread-safe
- Idea: If you do not need thread-safeness, this is much faster
- Tools for synchronization are available in the respective collection framework classes
- Note: Making a collection unmodifiable also makes a collection thread-safe, as the collection can't be modified. This avoids the synchronization overhead.

Synchronization

- Unlike when making a collection read-only, you synchronize the collection immediately after creating it.
- You also must make sure you do not retain a reference to the original collection, or else you can access the collection unsynchronized.
- The simplest way to make sure you don't retain a reference is to never create one:

```
Set set = Collections.synchronizedSet(new HashSet());
```

- Problem: this is a single collection-wide lock, hence we have a problem of scalability

Synchronization (contd.)

- `Collection synchronizedCollection(Collection collection)`
- `List synchronizedList(List list)`
- `Map synchronizedMap(Map map)`
- `Set synchronizedSet(Set set)`
- `SortedMap synchronizedSortedMap(SortedMap map)`
- `SortedSet synchronizedSortedSet(SortedSet set)`

All in `java.util.Collections`

Issues

- Important: `synchronizedMap` etc. are only conditionally thread-safe
- i.e. all individual operations are thread-safe
- but sequences of operations where the control flow depends on the results of previous operations may be subject to data races.

Issues – Examples

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());

// put-if-absent idiom -- contains a race condition
// may require external synchronization
if (!map.containsKey(key))
  map.put(key, value);

// ad-hoc iteration -- contains race conditions
// may require external synchronization
for (int i=0; i<list.size(); i++) {
  doSomething(list.get(i));
}
```

Issues – Examples (Cont'd)

```
// normal iteration --
// can throw ConcurrentModificationException
// may require external synchronization
for (Iterator i=list.iterator(); i.hasNext(); ) {
  doSomething(i.next());
}
```

Examples from http://www.ibm.com/developerworks/java/library/j-jtp07233.html

Issues – Examples (contd.)

Hence, as explained in the javadoc of `java.util.Collections`:

It is imperative that the user manually synchronize on the returned collection when iterating over it:

```
Collection c =
    Collections.synchronizedCollection(myCollection);
   ...
synchronized(c) {
    Iterator i = c.iterator(); // this must be in the
                              // synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Concurrent Classes

- Hence synchronization is only partially solved!
- The problem is, that the presented solutions might make the programmer believe that all problems are solved . . .

- If you need concurrent access, consider also
  `java.util.concurrent.ConcurrentHashMap`
  `java.util.concurrent.CopyOnWriteArrayList`
- However: look carefully at the respective advantages and compromises
- These classes are optimized for specific situations, not useful for other purposes