Java

Collections

Presented by Bartosz Sakowicz

Overview

- A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group, like a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a collection of name-to-phone-number mappings).

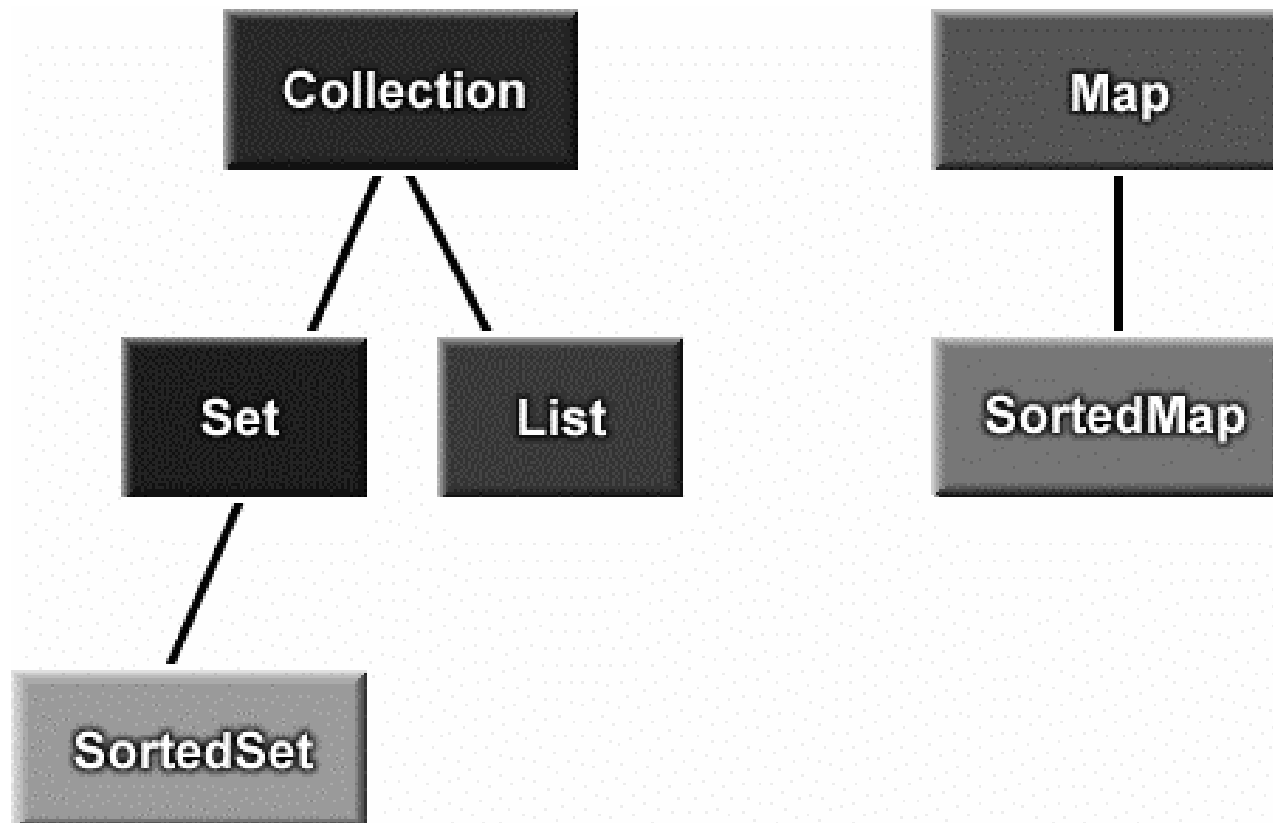
Collections framework

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain three things:

- **Interfaces**: abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.
- **Implementations**: concrete implementations of the collection interfaces.
- **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

Core collections interfaces

The **core collection interfaces** are the interfaces used to manipulate collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation:



The Collection interface

A Collection represents a group of objects, known as its elements. The primary use of the Collection interface is to pass around collections of objects where maximum generality is desired. The Collection interface is shown below:

```
public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);
    Iterator iterator();
```

The Collection interface(2)

boolean containsAll(Collection c);

boolean addAll(Collection c);

boolean removeAll(Collection c);

boolean retainAll(Collection c);

// Removes from the target Collection all of its elements that are not also contained in the specified Collection.

void clear();

Object[] toArray();

Object[] toArray(Object a[]); }

Iterator

Iterator is very similar to an Enumeration , but allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

The Iterator interface is shown below:

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Iterator - example

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (!cond(i.next()))
                i.remove();
    } //Another example:
java.util.Map result; //Creation somwhere else...
  if (result!=null) {
            java.util.Iterator i=result.entrySet().iterator();
            while(i.hasNext()) {
               java.util.Map.Entry entry=(java.util.Map.Entry)i.next();
               debug(entry.getKey()+" => "+entry.getValue()); }}
```

The Set Interface

A Set is a Collection that cannot contain duplicate elements. Set models the mathematical set abstraction. The Set interface extends Collection and contains no methods other than those inherited from Collection. It adds the restriction that duplicate elements are prohibited.Two Set objects are equal if they contain the same elements.

Usage of Set example:

Suppose you have a Collection, c, and you want to create another Collection containing the same elements, but with all duplicates eliminated. The following one-liner does the trick:

Collection noDups = new HashSet(c);

The Set Interface usage example

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
    Set s = new HashSet();
    for (int i=0; i<args.length; i++)
        if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);
    System.out.println(s.size()+" distinct words detected: "+s);
} }
```

The List Interface

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements.

The JDK contains two general-purpose List implementations. ArrayList, which is generally the best-performing implementation, and LinkedList which offers better performance under certain circumstances.

Two List objects are equal if they contain the same elements in the same order.

The List Interface(2)

```
public interface List extends Collection {
    Object get(int index);
    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);
    abstract boolean addAll(int index, Collection c);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator listIterator();
    ListIterator listIterator(int index);
    List subList(int from, int to); }
```

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. Two Map objects are equal if they represent the same key-value mappings.

The most useful methods:

Object put(Object key, Object value);

Object get(Object key);

Object remove(Object key);

boolean containsKey(Object key);

boolean containsValue(Object value);

public Set keySet();

public Collection values();

Object ordering

If the list consists of String elements, it will be sorted into lexicographic (alphabetical) order. If it consists of Date elements, it will be sorted into chronological order.

String and Date both implement the Comparable interface. The Comparable interfaces provides a natural ordering for a class, which allows objects of that class to be sorted automatically.

If you try to sort a list whose elements do not implement Comparable, Collections.sort(list) will throw a ClassCastException .

If you try to sort a list whose elements cannot be compared to one another, Collections.sort will throw a ClassCastException.

The Comparable Interface

The Comparable interface consists of a single method:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

The **compareTo** method compares the receiving object with the specified object, and returns a negative integer, zero, or a positive integer as the receiving object is less than, equal to, or greater than the specified Object.

If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

The Comparator Interface

A Comparator is an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method:

```
public interface Comparator {
 int compare(Object o1, Object o2);
}
```

The compare method compares its two arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

SortedSet and SortedMap

A SortedSet is a Set that maintains its elements in ascending order, sorted according to the elements natural order, or according to a Comparator provided at SortedSet creation time.

A SortedMap is a Map that maintains its entries in ascending order, sorted according to the **keys** natural order, or according to a Comparator provided at SortedMap creation time.

Implementations

JDK provides two implementations of each interface (with the exception of Collection).

All implementations permit null elements, keys and values.

All are Serializable, and all support a public clone method.

Each one is unsynchronized.

If you need a synchronized collection, the **synchronization wrappers** allow any collection to be transformed into a synchronized collection.

HashSet and TreeSet

- The two general purpose Set implementations are HashSet and TreeSet .
- HashSet is much faster but offers no ordering guarantees.
- If in-order iteration is important use TreeSet.
- Iteration in HashSet is linear in the sum of the number of entries and the capacity. It's important to choose an appropriate initial capacity if iteration performance is important. The default initial capacity is 101. The initial capacity may be specified using the int constructor. To allocate a HashSet whose initial capacity is 17:

  Set s= new HashSet(17);

ArrayList and LinkedList

The two general purpose List implementations are ArrayList and LinkedList . **ArrayList** offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method System.arraycopy when it has to move multiple elements at once.

If you frequently add elements to the beginning of the List, or iterate over the List deleting elements from its interior, you might want to consider **LinkedList**. These operations are constant time in a LinkedList but linear time in an ArrayList. Positional access is linear time in a LinkedList and constant time in an ArrayList.

HashMap and TreeMap

The two general purpose Map implementations are HashMap and TreeMap .

The situation for Map is exactly analogous to Set.

If you need SortedMap operations you should use TreeMap; otherwise, use HashMap.

Synchronization wrappers

The **synchronization wrappers** add automatic synchronization (thread-safety) to an arbitrary collection. There is one static factory method for each of the six core collection interfaces:

public static Collection synchronizedCollection(Collection c);

public static Set synchronizedSet(Set s);

public static List synchronizedList(List list);

public static Map synchronizedMap(Map m);

public static SortedSet synchronizedSortedSet(SortedSet s);

public static SortedMap synchronizedSortedMap(SortedMap m);

Each of these methods returns a synchronized (thread-safe) Collection backed by the specified collection.

Unmodifiable wrappers

Unmodifiable wrappers take away the ability to modify the collection, by intercepting all of the operations that would modify the collection, and throwing an **UnsupportedOperationException.** The unmodifiable wrappers have two main uses:

- To make a collection immutable once it has been built.
- To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper. In this way, the second-class citizens can look but not touch, while you maintain full access.

Unmodifiable wrappers(2)

There is one static factory method for each of the six core collection interfaces:

public static Collection unmodifiableCollection(Collection c);

public static Set unmodifiableSet(Set s);

public static List unmodifiableList(List list);

public static Map unmodifiableMap(Map m);

public static SortedSet unmodifiableSortedSet(SortedSet s);

public static SortedMap unmodifiableSortedMap(SortedMap m);

Java 1.0/1.1 containers

A lot of code was written using the Java 1.0/1.1 containers, and even new code is sometimes written using these classes. So although you should never use the old containers when writing new code, you'll still need to be aware of them. Here are older container classes:

Vector (~=ArrayList)

Enumeration (~= Iterator)

Hashtable (~=HashMap)

Stack (~=LinkedList)

All of them are synchronized (and slower).