

Collections and Algorithms

Everyone needs to implement lists and trees at least once, wouldn't you say? Computer science departments certainly think so and so do I, but not in the workplace. I suppose it didn't hurt me when I wrote binary search trees in FORTRAN at my first job in the late 70's, nor when I wrote a generic list library in C almost ten years ago, but you just can't afford to roll all of your own tools in production. The proliferation of C libraries in the 80's and C++ libraries in the 90's helped a great deal, but we C/C++ folks pride ourselves in writing portable code (or at least we used to, but that's another topic of discussion), and historically few libraries have ported well across platforms. This is a big enough problem that Bjarne Stroustrup, inventor of C++, once said that his biggest mistake with C++ was releasing it before it had a robust foundation library, including data structures.¹

That "mistake" has been more than rectified by the generic containers and algorithms in the now-standard C++ library (also erroneously called "STL"²). C++ programmers now routinely and effectively use containers of any type of object with maximum efficiency on any platform with a standard C++ system. Iterators and function objects are everyday terms now, and yet we're all still learning how far we can go with this brilliantly designed library.

But this is a column about Java, and Java has no STL, nor even templates (although there is talk in that direction). In 1997 Objectspace, Inc. ported STL to Java and called it JGL³, which originally stood for "The Java Generic Library", but Sun put the kibosh on such an official-sounding title, so now it's called "The Generic Collection Library for Java" (but the JGL acronym has stuck). JGL has everything STL-like that a Java programmer could hope for, even without templates. Function objects, for example, by convention use the `execute` method to substitute for overloading `operator ()` in C++. All the standard function objects and adapters are there.

JGL had a growing audience in the days of JDK 1.x, especially since the latter has only four wimpy containers: `Vector`, `Stack`, `Bitset`, and `Hashtable`. With Java 2, Sun has given us a lot less than JGL, but the new collections and their associated algorithms are well designed, easy-to-use, and you may find that they meet most day-to-day production software needs. In this article I survey the collections in both Java 1 and 2, with an emphasis on the latter⁴.

Collections in JDK 1.x

The `Vector` class is pretty much what you would expect: a container that holds an arbitrary number of instances derived from `Object`, which means you can store anything but primitives in a `Vector`. (This is how Java and some other object-oriented languages achieve genericity – by processing instances of a Mother of All Classes such as `Object`). The program in Listing 1 populates a `Vector` with seven `Integers` and uses the `indexOf` method to perform various searches. `indexOf` searches linearly from the beginning for a match (using the `equals` method, of course) and returns the 0-based index of where it first occurs, or `-1` if the object isn't there. There is also a `contains` method which returns a boolean indicating whether or not an object is present (see Listing 2). The program in Listing 2 also illustrates an Enumeration object, which is an iterator of sorts. The `Vector.elements` method returns an Enumeration which points to the first (zeroth) vector element. To traverse the container you query `Enumeration.hasMoreElements`; if returns `true`, you can access the next element via `Enumeration.nextElement`, which as a side effect advances the iterator for the next query. You can't delete an element via an Enumeration, but you can with Java 2's iterators, which you'll see shortly.

A serious drawback to the `Vector` class is that most of its methods are synchronized, which is fine if you need to write thread-safe code, but unacceptable otherwise, since your code has thread overhead you'll never use. To address this problem the Java 2 collections come in both synchronized and non-synchronized flavors (more on this below).

The `Stack` class attempts to give you the typical last-in-first-out container semantics that you expect, but evidently its designers skipped OO Design 101. For some mysterious reason, `Stack` extends `Vector`, instead of just using a `Vector` for implementation. The problem with this approach, of course, is that it lets you use a `Stack` as `Vector`, so you can add or remove elements anywhere you want instead of just at the beginning, which is all a `Stack` should let you do. I'm so appalled that I'm not even going to show an example that uses `Stack` (really!).

The `BitSet` class represents what its name implies: a collection of bits. Actually, a `BitSet` object is an ordered set of bits, beginning with bit 0, just like arrays. You can set, reset, and test individual bits and perform set operations on

BitSets as a whole (e.g., OR, AND, XOR, etc.). Oddly enough, there is no method to toggle a bit directly, so you'll have to do it manually. Bitsets grow as needed to contain whatever bit number you need to refer to. The `length` method returns how many bits are in use (in other words, length return n if $n-1$ is the highest bit number you've referenced). The program in Listing 3 creates two BitSets and computes their intersection relative to the first.

Hashtables are fast lookup tables. A Hashtable object stores key-value pairs by hashing the key into an open hash table, which is basically a dynamic array of linked-lists, each list holding pairs whose keys hash to the same value. Hash tables tend to perform faster than tree-based solutions, but are by nature unordered. As you can see in Listing 4, you add a key-value pair with the `put` method, and perform lookups with the `get` method. If you put a duplicate entry in the table for an existing key, the new value replaces the old (i.e., no duplicate keys, or "multimaps", allowed). If you search for a non-existent key, you get a null back. You can also search for the existence of keys and values independently, and even traverse them via Enumerations.

That's pretty much all you get with the 1.x JDK – no sets, no lists, no queues. And what's worse: no generic algorithms. If you want to sort a Vector, for example, tough luck; you have to write your own sort routine. Java 2 rectifies this situation with a well-defined hierarchy of collections and associated algorithms, and also provides replacements for all the 1.x containers except BitSet.

The Java 2 Collection Hierarchy

Java 2 provides two families of containers, Collections and Maps. Collections are containers that implement the Collection interface, which defines basic collection methods for inserting, removing and visiting elements (see Listing 5). The Collection interface is further refined into List and Set (see Figure 1). Lists are ordered sequences that support direct indexing(!) as well as bi-directional traversal. (For the STL-aware, a List is a sequence – see Listing 6 for its methods). Java 2 comes with LinkedList and ArrayList as implementations for List, as shown in Figure 2 (the leaves of the hierarchy are implementations). Vector is also there for historical reasons, but ArrayList is preferred over Vector in new code. LinkedList implements a doubly-linked list, and is optimal for inserting and removing elements anywhere in a list. ArrayList is simply an expandable array, and is preferred when you need to use random access to list elements.

The program in Listing 7 puts both ArrayList and LinkedList to the test. The `get` method returns the object at a particular position, while `indexOf` performs a linear search to find the index of an element, and returns `-1` if it's not present. The `subList` method is useful feature of the List interface. It returns a List that is a subset of the original, using an asymmetric range specification, similar to STL ranges, where the first index is inclusive and the second is exclusive. For example, the statements

```
LinkedList list = new LinkedList(array);  
List view = list.subList(1,3);
```

define `view` as containing the second and third elements of `list` (i.e., indexes 1 and 2), where `list` is itself a shallow copy of `array`. Since `view` is just a wrapper for a subset of `list`, any operations on `view` affect `list`. Appending Gregory's instance to `view`, therefore, inserts it after James, not Charles, the latter being outside of `view`.

Listing 7 also illustrates various algorithms found in the `java.util.Collections` class, such as `sort`, `binarySearch`, `max`, `min`, and `reverse`. These are similar in spirit to those in `java.util.Arrays`, but apply only to collections. The complete listing of algorithms in `java.util.Collections` is in Listing 8. Notice the methods containing the prefixes *unmodifiable* and *synchronized*. These methods act as wrappers for existing collections and restrict their use. If you call `unmodifiableCollection` on an existing collection, for example, the object returned would throw an `UnsupportedOperationException` if you try to call any of the Collection methods that modify a collection. The `synchronized` methods return objects that are thread-safe (i.e., the mutator methods in the wrapper are synchronized), which is the only way to use collections in multi-threaded situations since for efficiency reasons none of the Collection methods are synchronized. The wrappers act as a synchronized view into the original collection, and any changes you make in the wrapper are reflected in the backing collection. If want to iterate through a synchronized collection, you need to synchronize on the collection to ensure thread-safety, as in:

```
Collection coll = Collections.synchronizedCollection(aCollection);
```

```

    ...
synchronized(coll)
{
    Iterator iter = coll.iterator();
    while (iter.hasNext())
    {
        // use iter.next()...
    }
}

```

You may have noticed the glaring omission of some popular data structures, such as queue, stack and deque. Although they're not there explicitly, you can use `LinkedList` to achieve the desired effect. `LinkedList` includes the following methods, among others:

```

// Some LinkedList methods:
void addFirst(Object o); // push_front()
void addLast(Object o); // push_back()
Object getFirst(); // front()
Object getLast(); // back()
Object removeFirst(); // pop_front()
Object removeLast(); // pop_back()

```

For you STL folks, I have included the names of the equivalent C++ functions in the comments above. To implement a queue, all you have to do is use a `LinkedList` that supports `addFirst` and `removeLast`, as follows:

```

class Queue
{
    private LinkedList data;

    public Queue()
    {
        data = new LinkedList();
    }
    public void put(Object o)
    {
        data.addFirst(o);
    }
    public Object get()
        throws NoSuchElementException
    {
        return data.removeLast();
    }
    public int size()
    {
        return data.size();
    }
}

```

For a stack, just replace `removeLast` with `removeFirst`. For a deque, which allows insertions and removals at both ends, just use `LinkedList` as it is.

Iterators

Iterators, like Enumerations, are objects that facilitate traversing through an ordered sequence (see Listing 9). Unlike Enumerations, you can remove elements via an iterator. All collections support iterators, but the order of the traversal depends on the collection itself. For lists, you visit objects in the order they are stored. For sets (discussed below), objects appear in no particular order unless the set is ordered. The `ListIterator` interface, also in Listing 9, lets you update an object through an iterator, as shown in Listing 10. All the iterators returned by the Java 2 collections are *fail-fast*, which means that if the underlying structure changes during the lifetime of an iterator, other than through operations performed through the iterator itself, the iterator becomes invalid and any attempted use of it thereafter will result in a `ConcurrentModificationException`.

Sets

The Set interface conforms to the usual notion of a mathematical set, which basically is just an unordered receptacle for elements, and provides operations for insertion, removal, and testing of membership. Since these are already part of the Collections interface, the Set interface is identical to Collection, with the added specification that duplicates are not allowed. The SortedSet interface adds methods useful for processing a set ordered by comparator (see Listing 11). The HashSet class implements Set, and TreeSet implements SortedSet. HashSets are more efficient, since they use hashing algorithms, but if order is important, TreeSets provide logarithmic-time algorithms since they store object in balanced tree structures. As you can see in Listing 12, any attempts to insert duplicates into a set are ignored. For TreeSets, `headSet` returns a view from the beginning up to but not including the specified element and `tailSet` comprises the rest of the sequence.

Maps

A map, like a hash table, is a dictionary, or associative array. A map stores pairs, treating the first element of the pair as a key, and the second as its associated value. Maps provide fast search times for keys, and duplicate keys are not allowed. Like Sets, there are HashMaps and TreeMap, the former being faster and unordered while instances of the latter are ordered. For some reason maps are in no way related to collections in Java 2; they have their own interface and implementation hierarchies (see Figures 3 and 4). A WeakHashMap stores keys as weak references, which means that if the only reference to the key object is the weak reference in the map, then that object can be garbage-collected, whereupon the entry is automatically removed from the map. (This basically give you caching for free).

The key methods of the Map interface are `put` and `get` (see Listing 13 for the complete specification). The `put` method stores a pair of objects in the map, and `get` retrieves the value for the given key. If you want to just see the keys, `keySet` returns them as a Set view, and the `values` method returns the values as a Collection. If you want to iterate through the pairs, you can call `entrySet`, which returns the pairs as a Set of instances of `Map.EntrySet`, which also appears in Listing 13. All the above is illustrated in the program in Listing 14. Note that calling `put` a second time with the same key replaces its associated value.

An Application

Let's finish by doing something useful. The program in Listing 15 implements a typical cross-reference lister, which prints each word from a text file with the line numbers where it appears. Applying Listing 15 to its own text produces the output:

```
a: 2, 28
Add: 45, 54
addLast: 55
already: 49
and: 45
args: 81, 84, 93, 100, 101
at: 28
BufferedReader: 21, 88, 102
(etc.)
```

This program uses a Map that associates a token with a list of line numbers. Since I want the tokens to appear in alphabetical order, I use a TreeMap with the default comparator. I want case to be ignored in comparisons, so I define a comparator, called `NoCase`, that calls on `String.compareToIgnoreCase` to do the job. Before inserting a new pair, I first check to see if it is already in the map. If not, I add it with the statement

```
map.put(token, new LinkedList());
```

Then I retrieve the list of lines with

```
LinkedList lines = (LinkedList) map.get(token);
```

I then check to see if the current line number has already been added to the list; if not, I append it. It would have been simpler to just use a `TreeSet`, but the constant re-balancing of the tree would degrade performance, and since the lines are entered in the correct order, a list suffices.

Summary

I hope you'll agree that Java has coming of age as far as data structures and algorithms are concerned. The Arrays and Collections classes have useful algorithms for arrays and collections, respectively, and the families of Collection and Map implementations provide virtually everything you need in the way of data structures. As a C++ programmer, I must admit that I miss function objects a little, but only a little. I must also confess that I find the Java 2 collections easier to use than STL. Although my C++ version of the cross-reference program was 20 lines shorter than the Java version, it took me longer to get it right. To be fair, the C++ collections tend to be much faster than Java's, but this is the story of these two languages in general (I realize not everyone agrees with me here!). If you want blinding speed, use C++. For typical applications it's hard to imagine a situation that the Java 2 collections won't nicely handle, and if you need more intricate data structures, you have a nice hierarchy to plug into and algorithms ready to use.

Listing 1 – File `VectorTest.java`: Illustrates the Vector Class

```
import java.util.*;

class VectorTest
{
    static void search(Vector v, Object o)
    {
        int idx = v.indexOf(o);
        System.out.print(o + " ");
        if (idx < 0)
            System.out.println("not found");
        else
            System.out.println("found in position " + idx);
    }
    public static void main(String[] args)
    {
        // Build Vector:
        Vector v = new Vector();
        v.addElement(new Integer(88));
        v.addElement(new Integer(17));
        v.addElement(new Integer(-10));
        v.addElement(new Integer(34));
        v.addElement(new Integer(27));
        v.addElement(new Integer(0));
        v.addElement(new Integer(-2));
        System.out.println(v);

        // Search:
        search(v, new Integer(0));
        search(v, new Integer(1));
    }
}

/* Output:
[88, 17, -10, 34, 27, 0, -2]
0 found in position 5
1 not found
*/
```

Listing 2 – File VectorTest2.java: Traverses a Vector with an Enumeration

```
import java.util.*;

class VectorTest2
{
    static void printVector(Vector v)
    {
        System.out.println("Vector = {");
        Enumeration e = v.elements();
        for (int idx = 0; e.hasMoreElements(); ++idx)
            System.out.println("\t" + idx + ": "
                + e.nextElement());
        System.out.println("}");
    }
    static void search(Vector v, Object o)
    {
        System.out.print(o);
        if (!v.contains(o))
            System.out.print(" not");
        System.out.println(" found");
    }
    public static void main(String[] args)
    {
        // Build Vector:
        Vector v = new Vector();
        v.addElement(new Integer(88));
        v.addElement(new Integer(17));
        v.addElement(new Integer(-10));
        v.addElement(new Integer(34));
        v.addElement(new Integer(27));
        v.addElement(new Integer(0));
        v.addElement(new Integer(-2));
        printVector(v);

        // Search:
        search(v, new Integer(0));
        search(v, new Integer(1));
    }
}

/* Output:
Vector = {
    0: 88
    1: 17
    2: -10
    3: 34
    4: 27
    5: 0
    6: -2
}
0 found
1 not found
*/
```

Listing 3 – File BitSetTest.java: Illustrates the BitSet Container Class

```
import java.util.*;

class BitSetTest
{
    static void printBits(BitSet b)
    {
        System.out.println(
            "size: " + b.size() +
```

```

        ", length: " + b.length() +
        ", bits: " + b);
    }
    public static void main (String[] args)
    {
        BitSet b1 = new BitSet();
        printBits(b1);
        b1.set(1);
        printBits(b1);
        b1.set(3);
        printBits(b1);
        b1.set(5);
        printBits(b1);

        BitSet b2 = new BitSet();
        printBits(b2);
        for (int i = 0; i < 4; ++i)
            b2.set(i);
        printBits(b2);
        b1.and(b2);
        printBits(b1);
    }
}

/* Output:
size: 64, length: 0, bits: {}
size: 64, length: 2, bits: {1}
size: 64, length: 4, bits: {1, 3}
size: 64, length: 6, bits: {1, 3, 5}
size: 64, length: 0, bits: {}
size: 64, length: 4, bits: {0, 1, 2, 3}
size: 64, length: 4, bits: {1, 3}
*/

```

Listing 4 – File HashtableTest.java: Maps States to their Capitals

```

import java.util.*;

class HashtableTest
{
    public static void main(String[] args)
    {
        Hashtable h = new Hashtable();
        h.put("Alabama", "Montgomery");
        h.put("Tennessee", "Nashville");
        h.put("Georgia", "Savannah");
        // The following value replaces "Savannah":
        h.put("Georgia", "Atlanta");
        System.out.println(h);
        test(h);
        iterate(h);
    }

    static void test(Hashtable h)
    {
        if (h.containsKey("Alabama"))
            System.out.println("Alabama is a key");
        if (h.contains("Montgomery"))
            System.out.println("Montgomery is a value");
        System.out.println("m[Georgia] = " +
            h.get("Georgia"));
        System.out.println("m[Michigan] = " +
            h.get("Michigan"));
    }
}

```

```

static void iterate(Hashtable h)
{
    Enumeration e = h.keys();
    System.out.println("Keys:");
    while (e.hasMoreElements())
    {
        System.out.print("\t");
        System.out.println(e.nextElement());
    }

    System.out.println("Values:");
    e = h.elements();
    while (e.hasMoreElements())
    {
        System.out.print("\t");
        System.out.println(e.nextElement());
    }
}

/* Output:
{Tennessee=Nashville, Georgia=Atlanta, Alabama=Montgomery}
Alabama is a key
Montgomery is a value
m[Georgia] = Atlanta
m[Michigan] = null
Keys:
    Tennessee
    Georgia
    Alabama
Values:
    Nashville
    Atlanta
    Montgomery
*/

```

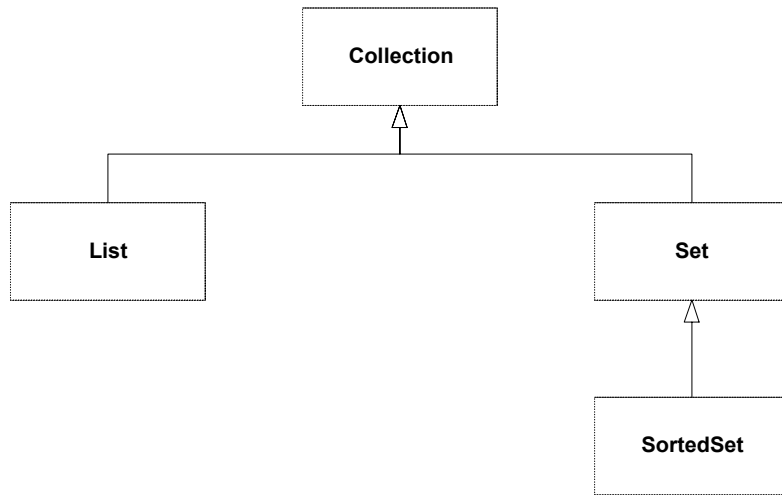
Listing 5 – The Collection Interface

```

Interface Collection
{
    // All mutators are optional!
    boolean    add(Object o);
    boolean    addAll(Collection c);
    void       clear();
    boolean    contains(Object o);
    boolean    containsAll(Collection c);
    boolean    equals(Object o);
    int        hashCode();
    boolean    isEmpty();
    Iterator   iterator();
    boolean    remove(Object o);
    boolean    removeAll(Collection c);
    boolean    retainAll(Collection c);
    int        size();
    Object[]   toArray();
    Object[]   toArray(Object[] a);
}

```

Figure 1 – The Collection Interface Hierarchy

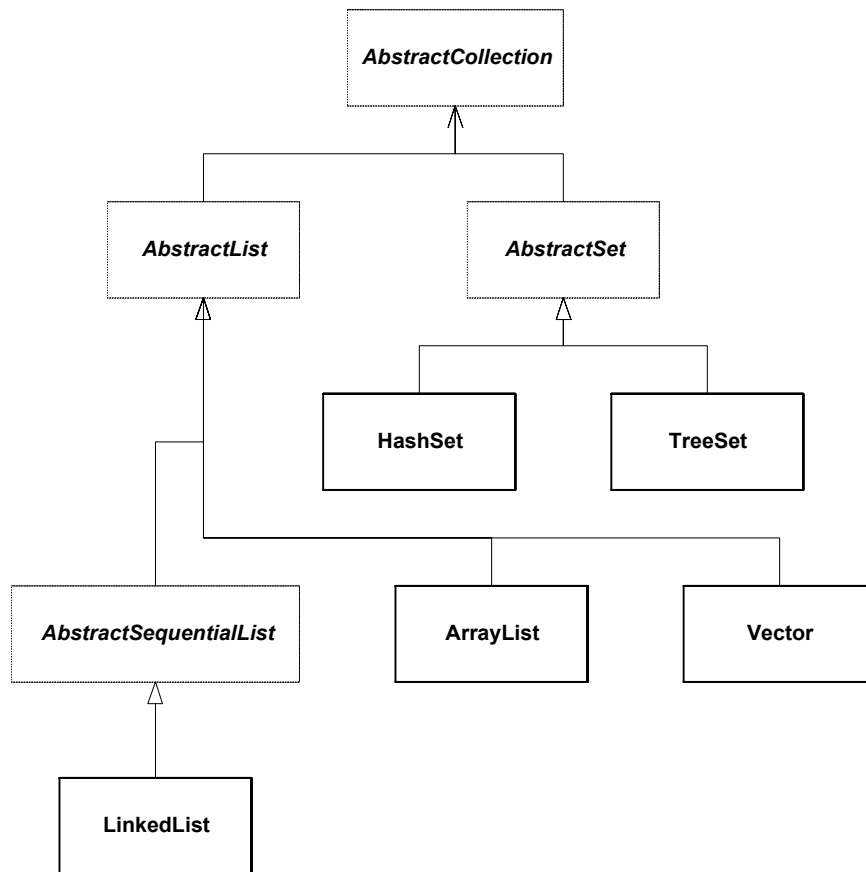


Listing 6 – The List Interface

Interface List extends Collection

```
{  
    void          add(int index, Object element)  
    Object        get(int index)  
    int           indexOf(Object o)  
    int           lastIndexOf(Object o)  
    ListIterator  listIterator()  
    ListIterator  listIterator(int index)  
    Object        remove(int index)  
    Object        set(int index, Object element)  
    int           size()  
    List          subList(int fromIndex, int toIndex)  
}
```

Figure 2 – Implementations for the Collection Interfaces



Listing 7 – File ListTest.java: Illustrates ArrayList and LinkedList

```
import java.util.*;

class ListTest
{
    public static void test(List x)
    {
        // Do a search:
        Object p = x.get(2);
        if (x.contains(p))
            System.out.println("Found " + p);

        // Do another:
        int index = x.indexOf(p);
        if (index != -1)
            System.out.println("Found " + p +
                " in position "
                + index);

        // Do a binary search:
        Collections.sort(x);
        System.out.println(x);
        index = Collections.binarySearch(x, p);
        if (index >= 0)
            System.out.println("Found " +
                p + " in position "
                + index);
    }
}
```

```

    // Misc:
    System.out.println("max == " +
        Collections.max(x));
    System.out.println("min == " +
        Collections.min(x));
    Comparator desc = Collections.reverseOrder();
    System.out.println("max (desc) == " +
        Collections.max(x, desc));
    System.out.println("min (desc) == " +
        Collections.min(x, desc));
    Collections.shuffle(x);
    iterate(x);
}

public static void iterate(Collection c)
{
    System.out.println("Iterating...");
    Iterator i = c.iterator();
    while (i.hasNext())
        System.out.println(i.next());
}

public static void main(String[] args)
{
    // Build ArrayList:
    ArrayList array = new ArrayList();
    array.add(new Person("Horatio", 1835,12,6));
    array.add(new Person("Charles",1897,3,11));
    array.add(new Person("Albert",1901,1,20));
    System.out.println(array);
    test(array);
    array.add(new Person("James", 1976, 8, 13));
    System.out.println(array);
    System.out.println();

    // Build LinkedList:
    LinkedList list = new LinkedList(array);
    List view = list.subList(1,3);
    view.add(new Person("Gregory", 1582, 10, 15));
    Collections.reverse(view);
    System.out.println(list);
    test(list);
}
}

/* Output:
[Horatio,12/6/1835], {Charles,3/11/1897}, {Albert,1/20/1901}
Found {Albert,1/20/1901}
Found {Albert,1/20/1901} in position 2
[Albert,1/20/1901], {Charles,3/11/1897}, {Horatio,12/6/1835}
Found {Albert,1/20/1901} in position 0
max == {Horatio,12/6/1835}
min == {Albert,1/20/1901}
max (desc) == {Albert,1/20/1901}
min (desc) == {Horatio,12/6/1835}
Iterating...
{Horatio,12/6/1835}
{Albert,1/20/1901}
{Charles,3/11/1897}
[Horatio,12/6/1835], {Albert,1/20/1901}, {Charles,3/11/1897}, {James,8/13/1976}

[Horatio,12/6/1835], {Gregory,10/15/1582}, {Charles,3/11/1897}, {Albert,1/20/1901},
{James,8/13/1976}
Found {Charles,3/11/1897}

```

```

Found {Charles,3/11/1897} in position 2
[{Albert,1/20/1901}, {Charles,3/11/1897}, {Gregory,10/15/1582}, {Horatio,12/6/1835},
{James,8/13/1976}]
Found {Charles,3/11/1897} in position 1
max == {James,8/13/1976}
min == {Albert,1/20/1901}
max (desc) == {Albert,1/20/1901}
min (desc) == {James,8/13/1976}
Iterating...
{James,8/13/1976}
{Gregory,10/15/1582}
{Charles,3/11/1897}
{Horatio,12/6/1835}
{Albert,1/20/1901}
*/

```

Listing 8 – The Algorithms in java.util.Collections

```

class Collections
{
    static int binarySearch(List list, Object key);
    static int binarySearch(List list, Object key, Comparator c);
    static void copy(List dest, List src);
    static Enumeration enumeration(Collection c);
    static void fill(List list, Object o);
    static Object max(Collection coll);
    static Object max(Collection coll, Comparator comp);
    static Object min(Collection coll);
    static Object min(Collection coll, Comparator comp);
    static List nCopies(int n, Object o);
    static void reverse(List l);
    static Comparator reverseOrder();
    static void shuffle(List list);
    static void shuffle(List list, Random rnd);
    static Set singleton(Object o);
    static void sort(List list);
    static void sort(List list, Comparator c);
    static Collection synchronizedCollection(Collection c);
    static List synchronizedList(List list);
    static Map synchronizedMap(Map m);
    static Set synchronizedSet(Set s);
    static SortedMap synchronizedSortedMap(SortedMap m);
    static SortedSet synchronizedSortedSet(SortedSet s);
    static Collection unmodifiableCollection(Collection c);
    static List unmodifiableList(List list);
    static Map unmodifiableMap(Map m);
    static Set unmodifiableSet(Set s);
    static SortedMap unmodifiableSortedMap(SortedMap m);
    static SortedSet unmodifiableSortedSet(SortedSet s);
}

```

Listing 9 – The Iterator and List Iterator Interfaces

```

Interface Iterator
{
    boolean    hasNext();
    Object     next();
    void       remove();
}

Interface ListIterator
{
    void       add(Object o);
    boolean    hasNext();
    boolean    hasPrevious();
}

```

```

    Object    next();
    int       nextIndex();
    Object    previous();
    int       previousIndex();
    void      remove();
    void      set(Object o);
}

```

Listing 10 – File Modify.java: Modifies a List element via a ListIterator

```

import java.util.*;

class Modify
{
    public static void main(String[] args)
    {
        // Build Array:
        ArrayList a = new ArrayList();
        a.add(new Integer(1));
        a.add(new Integer(2));
        a.add(new Integer(3));
        System.out.println(a);

        // Modify via iterator:
        ListIterator p =
            (ListIterator) a.listIterator();
        while (p.hasNext())
        {
            Integer i = (Integer) p.next();
            p.set(new Integer(i.intValue()
                               + 1));
        }
        System.out.println(a);
    }
}

/* Output:
[1, 2, 3]
[2, 3, 4]
*/

```

Listing 11 – The SortedSet Interface

```

Interface SortedSet extends Set
{
    Comparator comparator();
    Object first();
    SortedSet headSet(Object toElement);
    Object last();
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet tailSet(Object fromElement);
}

```

Listing 12 – File SetTest.java: Illustrates Sets

```

import java.util.*;

class SetTest
{
    public static void test(Set x)
    {
        // Do a search:
        Person p = new Person("Albert",1901,1,20);
        if (x.contains(p))
            System.out.println("Found " + p);
    }
}

```

```

        // Misc:
        System.out.println("max == " +
            Collections.max(x));
        System.out.println("min == " +
            Collections.min(x));
        iterate(x);
    }

    public static void iterate(Collection c)
    {
        System.out.println("Iterating...");
        Iterator i = c.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }

    public static void main(String[] args)
    {
        // Build HashSet:
        HashSet h = new HashSet();
        h.add(new Person("Horatio", 1835,12,6));
        h.add(new Person("Charles",1897,3,11));
        h.add(new Person("Albert",1901,1,20));
        // The following is ignored:
        h.add(new Person("Albert",1901,1,20));
        System.out.println(h);
        test(h);
        System.out.println();

        // Build TreeSet:
        TreeSet t =
            new TreeSet(Collections.reverseOrder());
        t.addAll(h);
        System.out.println(t);
        test(t);

        // Extra TreeSet stuff:
        boolean ctest =
            t.comparator()
                .equals(Collections.reverseOrder());
        System.out.println("Comparator test: " + ctest);
        System.out.println("first = " + t.first());
        System.out.println("last = " + t.last());
        Person p = new Person("Charles",1897,3,11);
        System.out.println("headSet: " + t.headSet(p));
        System.out.println("tailSet: " + t.tailSet(p));
    }
}

/* Output:
[[{Charles,3/11/1897}, {Albert,1/20/1901}, {Horatio,12/6/1835}]
Found {Albert,1/20/1901}
max == {Horatio,12/6/1835}
min == {Albert,1/20/1901}
Iterating...
{Charles,3/11/1897}
{Albert,1/20/1901}
{Horatio,12/6/1835}

[[{Horatio,12/6/1835}, {Charles,3/11/1897}, {Albert,1/20/1901}]
Found {Albert,1/20/1901}
max == {Horatio,12/6/1835}
min == {Albert,1/20/1901}
Iterating...

```

```

{Horatio,12/6/1835}
{Charles,3/11/1897}
{Albert,1/20/1901}
Comparator test: true
first = {Horatio,12/6/1835}
last = {Albert,1/20/1901}
headSet: [{Horatio,12/6/1835}]
tailSet: [{Charles,3/11/1897}, {Albert,1/20/1901}]
*/

```

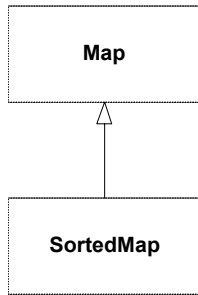
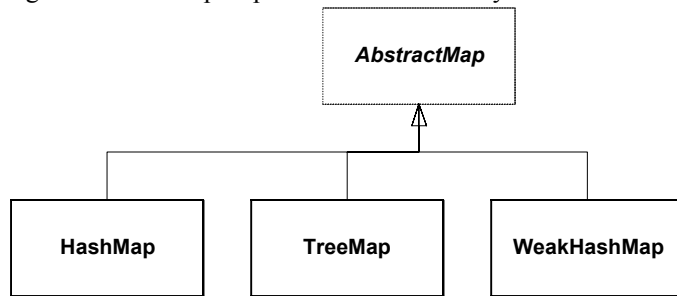


Figure 3 – The Map Interface Hierarchy

Figure 4 – The Map Implementation Hierarchy



Listing 13 – The Map Interfaces

```

Interface Map
{
    void clear();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Set entrySet();
    boolean equals(Object o);
    Object get(Object key);
    int hashCode();
    boolean isEmpty();
    Set keySet();
    Object put(Object key, Object value);
    void putAll(Map t);
    Object remove(Object key);
    int size();
    Collection values();

    static Interface Map.Entry
    {
        boolean equals(Object o);
        Object getKey();
        Object getValue();
        int hashCode();
        Object setValue(Object value);
    }
}

```

```

}

Interface SortedMap extends Map
{
    Comparator    comparator();
    Object         firstKey();
    SortedMap     headMap(Object toKey);
    Object         lastKey();
    SortedMap     subMap(Object fromKey, Object toKey);
    SortedMap     tailMap(Object fromKey);
}

```

Listing 14 – File MapTest.java: Illustrates Maps

```

import java.util.*;

class MapTest
{
    public static void main(String[] args)
    {
        HashMap h = new HashMap();
        h.put("Alabama", "Montgomery");
        h.put("Tennessee", "Nashville");
        h.put("Georgia", "Savannah");
        // The following value replaces "Savannah":
        h.put("Georgia", "Atlanta");
        System.out.println(h);
        test(h);
        System.out.println();

        TreeMap t =
            new TreeMap(Collections.reverseOrder());
        t.putAll(h);
        System.out.println(t);
        test(t);

        // Extra TreeMap Stuff:
        boolean ctest =
            t.comparator()
                .equals(Collections.reverseOrder());
        System.out.println("Comparator test: " + ctest);
        System.out.println("firstKey = " +
            t.firstKey());
        System.out.println("lastKey = " +
            t.lastKey());
        System.out.println("headMap: " +
            t.headMap("Georgia"));
        System.out.println("tailMap: " +
            t.tailMap("Georgia"));
    }

    public static void test(Map m)
    {
        if (m.containsKey("Alabama"))
            System.out.println("Alabama is a key");
        if (m.containsValue("Montgomery"))
            System.out.println("Montgomery is a value");
        System.out.println("m[Georgia] = " +
            m.get("Georgia"));
        System.out.println("m[Michigan] = " +
            m.get("Michigan"));
        System.out.println("Keys: " + m.keySet());
        System.out.println("Values: " + m.values());
        iterate(m);
    }
}

```

```

public static void iterate(Map m)
{
    System.out.println("Iterating...");
    Set s = m.entrySet();
    Iterator i = s.iterator();
    while (i.hasNext())
    {
        Map.Entry e = (Map.Entry) i.next();
        System.out.println(e);
    }
}

/* Output:
{Tennessee=Nashville, Georgia=Atlanta, Alabama=Montgomery}
Alabama is a key
Montgomery is a value
m[Georgia] = Atlanta
m[Michigan] = null
Keys: [Tennessee, Georgia, Alabama]
Values: [Nashville, Atlanta, Montgomery]
Iterating...
Tennessee=Nashville
Georgia=Atlanta
Alabama=Montgomery

{Tennessee=Nashville, Georgia=Atlanta, Alabama=Montgomery}
Alabama is a key
Montgomery is a value
m[Georgia] = Atlanta
m[Michigan] = null
Keys: [Tennessee, Georgia, Alabama]
Values: [Nashville, Atlanta, Montgomery]
Iterating...
Tennessee=Nashville
Georgia=Atlanta
Alabama=Montgomery
Comparator test: true
firstKey = Tennessee
lastKey = Alabama
headMap: {Tennessee=Nashville}
tailMap: {Georgia=Atlanta, Alabama=Montgomery}
*/

```

Listing 15 – File Xref.java: A Cross-Reference Lister

```

// Generates a token-to-line cross-reference listing
import java.util.*;
import java.io.*;

class Xref
{
    // Comparator to ignore case:
    static class NoCase implements Comparator
    {
        public int compare(Object o1, Object o2)
        {
            String s1 = (String) o1;
            String s2 = (String) o2;
            return s1.compareToIgnoreCase(s2);
        }
    }
}

```

```

}

// This method does the work:
static void process(BufferedReader r)
    throws IOException
{
    TreeMap map = new TreeMap(new NoCase());
    String line;
    int lineno = 0;

    // Build map, reading a line at a time:
    while ((line = r.readLine()) != null)
    {
        ++lineno;

        // Read each token:
        String delim =
            "`~!@#$$%^&*()-_+=+\\|[{]};:'\"<.>/?"
            + "0123456789";
        StringTokenizer tokens =
            new StringTokenizer(line, delim);
        while (tokens.hasMoreTokens())
        {
            String token = tokens.nextToken();

            if (!map.containsKey(token))
            {
                // Add token and empty list to map:
                map.put(token, new LinkedList());
            }

            // See if this line is in there already:
            LinkedList lines = (LinkedList) map.get(token);
            if (lines.isEmpty() ||
                ((Integer)lines.getLast()).intValue() != lineno)
            {
                // Add line number to list:
                lines.addLast(new Integer(lineno));
            }
        }
    }

    // Output:
    Iterator p = map.entrySet().iterator();
    while (p.hasNext())
    {
        Map.Entry e = (Map.Entry) p.next();
        System.out.print(e.getKey() + ": ");
        LinkedList lines = (LinkedList) e.getValue();
        Iterator lineIter = lines.iterator();
        while (lineIter.hasNext())
        {
            Integer i = (Integer) lineIter.next();
            if (i != (Integer) lines.getFirst())
            {
                System.out.print(", ");
            }
            System.out.print(i);
        }
        System.out.println();
    }
}

```

```

public static void main(String[] args)
    throws IOException
{
    if (args.length == 0)
    {
        // Process standard input:
        Reader r = new InputStreamReader(System.in);
        process(new BufferedReader(r));
    }
    else
    {
        // Process each specified file:
        for (int i = 0; i < args.length; ++i)
        {
            if (i > 0)
            {
                System.out.println();
            }
            System.out.println("*** File: " +
                args[i] + " ***");
            FileReader f = new FileReader(args[i]);
            process(new BufferedReader(f));
        }
    }
}

```

¹ See my interview with Bjarne Stroustrup, "C++: The Making of a Standard", CUJ, October 1996, also available at <http://www.freshsources.com> (select "C++" under Articles).

² Don't get me started. The term STL is both misused and obsolete.

³ Read about JGL at <http://www.objectspace.com/jgl/prodJGL.asp>.

⁴ The array algorithms in `java.util.Arrays` were discussed in the March 2000 issue of this column.