CS2 Advanced Programming in Java note 6

Collections in Java

Organising a library

This note provides a summary of the Java libraries for manipulating collections of objects. Having read it, you should be able to explain the different kinds of Java collection, and their various concrete implementations.

To supplement this note, you need to read your course textbook: *JPL* §16; *TIJ* §9; *UJ* §14; and look at the online Java lesson on collections.

Two Books

Budd, *Classic Data Structures in Java*, Addison Wesley.

Good for learning how standard datastructures are implemented. Well organised, properly object-oriented, and clearly explains the use of Java.

Ince, *From Data Structures to Patterns*, Macmillan.

Concentrates on design patterns and how to use classes, rather than the details of datastructure implementation.

6.1 Old Java

The first release of Java provided a few classes in java.util specifically to support operations on collections of objects. Obviously *arrays* are built in to the language, but there is also a class Vector that provides arrays whose size can vary dynamically. These are implemented using standard arrays, but when the array becomes overfull it is copied across to a new, larger, one. An Enumeration provides one-by-one access to the elements of a vector. Other classes are a last-in-first-out Stack of objects, and a Hashtable which maps keys to values.

6.2 New Java

One of the significant features in the rebranded "Java 2" (*i.e.* releases 1.2 onwards) was a larger library of collection classes. As well as providing efficient implementations of useful datastructures, the classes are interesting for their use of inheritance and other OO techniques for arranging an extensible library.

The library makes a clear distinction between *abstract datatypes* (ADT's) that require a particular suite of operations, and concrete *datastructures* that implement them, perhaps in more than one way. An ADT is given as an interface, with classes for the datastructures. For example, the Set interface has one implementation based on trees (TreeSet) and another using a hash table (HashSet). Between interfaces and concrete classes, there are a range of abstract classes, making it simpler to extend the library with custom-built collections.

This is more than just type-checking: the documentation for these interfaces declares *contracts* which any implementation should honour, but which are not checked by a compiler. For example, that a Set should not contain duplicates.

Programs using the collection library can take advantage of this separation between *specification* and *implementation*. For most code it is enough to state the interface that some variable or method argument must support; this makes that code *polymorphic*. Only when actually creating an object is it necessary to pick a particular implementing class; and the choice then depends on balancing space and time efficiency.

The old classes Vector, Stack and Hashtable still have positions in the new collections hierarchy, to provide backwards compatibility. They fit a little awkwardly, being a blurred mix of specification and implementation.

6.3 At the top: interfaces

Collection is a top-level interface in java.util for describing any *collection* of objects: it specifies methods like add, remove and contains for single objects; and containsAll, addAll, removeAll, and retainAll to combine one collection with another. Collection makes no specification about whether elements might repeat, or what order they are in. Collections are known as *multisets* or *bag*s in other languages.

Oddly, Java has no direct implementation of Collection. Every concrete collection is an instance of one of its two subinterfaces Set and List. A *set* is a collection where objects appear at most once, and order is unimportant; a *list* is a collection where order matters, and objects may be repeated. The Set interface doesn't actually add any methods: it is simply a statement of intent. List adds methods to manipulate elements at particular places in the list.

As well as operations on a whole collection, it can be useful to work through elements one at a time with an Iterator. This interface provides hasNext and next methods, and every collection has an iterator method that returns an iterator over itself. A List is more structured, and can offer a ListIterator which adds operations to work backwards as well as forwards. When the underlying collection can be modified, these iterators include methods to do so at the current point.

Sometimes it makes sense to keep a collection sorted, for convenience or efficiency. This is specified by the SortedSet interface, an child of Set. There are two general ways to indicate how sorting should happen. Where a class has some natural ordering, like Date, it should implement the java.lang.Comparable interfaces with a compareTo method. When the ordering is in some way unusual, a one-off class implementing java.util.Comparator is required, with a class method to compare two objects. A SortedSet must then ensure, for example, that any iterator works through elements in the given order.

The remaining interface in the library is Map. A *map* describes an association between *keys* and *values*, where every key matches at most one value, and keys are not repeated. The primary operation is to get the value for a given key; there are other methods to put in a new key-value *entry*, to remove a key, and to test whether a map contains some key or value. Any map has three *collection views*: the set of keys, the collection of values, and the set of entries. Maps too can be sorted: the SortedMap interface specifies that the keys will be a SortedSet.

6.4 At the bottom: concrete classes

The library provides various implementations of the ADT's described above. In general the algorithms chosen are not particularly elaborate, but they are more efficient than the very simplest approach would be.

A TreeSet implements sorted sets using red-black binary trees, a balanced structure equivalent to 2-3 trees that guarantees $O(\log n)$ access to elements. A HashSet implements general sets using a hash function to index an array of short lists. This can be very fast, with constant-time access, provided everything is well-tuned: the hash function behaves properly and the load factor is good.

An ArrayList implements lists using an expandable array, like the old Vector class. This is quick to read from, but adding an element anywhere except the end is slow; and occasionally adding a single object means copying the whole array into a larger one. A LinkedList implements a doubly-linked list of objects: it may take longer to get to elements in the middle, but adding new ones is always fast. This is the right class to use for a *queue*, for example.

Maps are like sets, with a TreeMap implementation when keys can be sorted, and HashMap using a hash-indexed array. Trees guarantee log-time operations; hashes can be constant-time, but with no certainty.

6.5 In the middle: abstract classes

The collection interfaces are generous, specifying lots of methods, some with actions or parameters that vary only slightly. This makes them convenient to use, but increases the task for writers of new kinds of collection. To make this easier, the library has abstract classes containing skeleton implementations: AbstractCollection, AbstractMap, AbstractSet, AbstractList, and AbstractSequentialList.

For example, the Collection interface specifies a toArray method to fill an array with the elements of a collection. This can always be defined in terms of size and iterator, and AbstractCollection does just that. Indeed, a subclass need only implement iterator, size, and add to have a fully working collection: all other methods will then call these, thanks to inheritance and dynamic binding. Naturally an implementation may override some for efficiency: the clear method to remove every element of a collection can be implemented using iterate, but most representations improve on this.

There are two kinds of new collection class one might create: alternative implementations of set or list, perhaps with better algorithms; and other sorts of collection, like a bag or an ordered list. The abstract classes can help with both, but it may not be clear what is the best minimum set of methods. This is how AbstractList and AbstractSequentialList arise: one expects easy random access, as with ArrayList, and the other is based on efficient iterators, like LinkedList. Each then uses these to implement all the operations a list needs.

6.6 Useful things and a problem

The library includes two classes with static methods only, Arrays and Collections. They provide utility functions like Arrays.sort to quicksort any array, and also some wrappers that take a collection and return another with same underlying data but different properties. For example, the wrapper Collections.unmodifiableList returns a view on a list that can access values but not change them. Similarly, as none of the new collection classes are thread-safe, there are synchronization wrappers to make them so.

All of these classes deal only with collections of Objects. This works, because anything can be upcast, but it is not ideal. Suppose we have a class Widget, and build a Set of them. On taking one out, we know it to be a widget, but before using any Widget methods it must be explicitly downcast from Object. This has a runtime cost, as the JVM does some tests *even though they can never fail.* Worse, if there was a mistake and the set did contain a non-widget, the right time to detect this would be when it is added, but automatic upcasting hides it.

Information has been lost here; putting it back is known in OO as *generic* programming, and there is a project "GJ" to do this for Java. GJ has *parametric* types like List<Widget> which ensures that only widgets go in, and asserts that only widgets come out. As well as checking code for correct usage, GJ *retrofits* the collection classes with parametric types to reflect their behaviour. Generics are scheduled to appear in Java 1.5, with a compiler available from Sun now — see http://www.research.avayalabs.com/user/wadler/pizza/gj/.

Exercises

1. Write a method **static void** reverse(**int**[] a) to reverse the contents of an array in place. Write a program to test it on several arrays of different sizes. Swap your tests with someone else.
2. Write a method inThree that given three sets a, b and c, will return a new set containing just the objects in all three. Write tests, swap with someone else.
3. The directory /home/cs2/will contains the complete works of William Shakespeare. Write a program to find out which words appear exactly five times (ignore the glossary and all README files, as he didn't write those).

Ian Stark