Principles of Software Construction: The Design of the Collections API – Part 2

Josh Bloch            Charlie Garrod
Administrivia

• Homework 4b due today
• Have a great spring break
Key concepts from Tuesday...

• Finished discussing API design
  – One-page handout still available if you want one
• Started discussing design of collections framework
  – Implementations, interfaces, algorithms
Outline

I. The initial release of the collections API
II. Design of the first release
III. Evolution
IV. Code example
V. Critique
Collection interfaces
first release, 1998
General-purpose **implementations**

*first release, 1998*

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>HashSet, TreeSet</td>
</tr>
<tr>
<td>List</td>
<td>ArrayList, LinkedList</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap, TreeMap</td>
</tr>
</tbody>
</table>
Other implementations

first release, 1998

• Convenience implementations
  – Arrays.asList(Object[] a)
  – EMPTY_SET, EMPTY_LIST, EMPTY_MAP
  – singleton(Object o)
  – nCopies(Object o)

• Decorator implementations
  – Unmodifiable{Collection, Set, List, Map, SortedMap}
  – Synchronized{Collection, Set, List, Map, SortedMap}

• Special Purpose implementation – WeakHashMap
Reusable **algorithms**

*first release, 1998*

- static void `sort(List[])`;
- static int `binarySearch(List list, Object key)`;
- static object `min(List[])`;
- static object `max(List[])`;
- static void `fill(List list, Object o)`;
- static void `copy(List dest, List src)`;
- static void `reverse(List list)`;
- static void `shuffle(List list)`;
And that’s all there was to it!
OK, I told a little white lie:
Array utilities, *first release, 1998*

- static int `binarySearch(type[] a, type key)`
- static int `binarySearch(Object[] a, Object key, Comparator c)`
- static boolean `equals(type[] a, type[] a2)`
- static void `fill(type[] a, type val)`
- static void `fill(type[] a, int fromIndex, int toIndex, type val)`
- static void `sort(type[] a)`
- static void `sort(type[] a, int fromIndex, int toIndex)`
- static void `sort(type[] a, Comparator c)`
- static void `sort(type[] a, int fromIdx, int toIdx, Comparator c)`
Documentation matters

Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

Of course you need good JavaDoc

But it is not sufficient for a substantial API
A single place to go for documentation

The Collections Framework

The collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on six collection interfaces. It includes implementations of these interfaces, and algorithms to manipulate them.

Overview

- **Overview** - An overview of the Collections framework.

API Specification

- **API Reference** - An annotated outline of the classes and interfaces comprising the collections framework, with links into the JavaDoc.

API Enhancements

- **API Enhancements** - An annotated list of API changes between the Beta4 and FCS releases, with links into the JavaDoc.

Design FAQ

- **Design FAQ** - Answers to frequently asked questions concerning the design of the collections framework.

Tutorial

- **Tutorial** - A tutorial introduction to the collections framework with plenty of programming examples.
Overviews provide understanding
A place to go when first learning an API

Collections Framework Overview

Introduction

The 1.2 release of the Java platform includes a new collections framework. A collection is an object that represents a group of objects (such as the familiar Vector class). A collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.

The primary advantages of a collections framework are that it:

- Reduces programming effort by providing useful data structures and algorithms so you don't have to write them yourself.
- Increases performance by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.
- Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
- Reduces the effort required to learn APIs by eliminating the need to learn multiple ad hoc collection APIs.
- Reduces the effort required to design and implement APIs by eliminating the need to produce ad hoc collections APIs.
- Fosters software reuse by providing a standard interface for collections and algorithms to manipulate them.

The collections framework consists of:

- Collection Interfaces - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- General-purpose Implementations - Primary implementations of the collection interfaces.
- Legacy Implementations - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
- Wrapper Implementations - Add functionality, such as synchronization, to other implementations.
- Convenience Implementations - High-performance "mini-implementations" of the collection interfaces.
- Abstract Implementations - Partial implementations of the collection interfaces to facilitate custom implementations.
- Algorithms - Static methods that perform useful functions on collections, such as sorting a list.
- Infrastructure - Interfaces that provide essential support for the collection interfaces.
- Array Utilities - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.
Annotated outlines provide access
They’re awesome and underutilized

Annotated Outline of Collections Framework

The collections framework consists of:

- **Collection Interfaces** - The primary means by which collections are manipulated.
  - **Collection** - A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.
  - **Set** - The familiar set abstraction. No duplicate elements permitted. May or may not be ordered. Extends the Collection interface.
  - **List** - Ordered collection, also known as a sequence. Duplicates are generally permitted. Allows positional access. Extends the Collection interface.
  - **Map** - A mapping from keys to values. Each key can map to at most one value.
  - **SortedSet** - A set whose elements are automatically sorted, either in their natural ordering (see the Comparable interface), or by a Comparator object provided when a SortedSet instance is created. Extends the Set interface.
  - **SortedMap** - A map whose mappings are automatically sorted by key, either in the keys' natural ordering or by a comparator provided when a SortedMap instance is created. Extends the Map interface.

- **General-Purpose Implementations** - The primary implementations of the collection interfaces.
  - **HashSet** - Hash table implementation of the Set interface. The best all-around implementation of the Set interface.
  - **TreeSet** - Red-black tree implementation of the SortedSet interface.
  - **ArrayList** - Resizable-array implementation of the List interface. (Essentially an unsynchronized Vector.) The best all-around implementation of the List interface.
  - **LinkedList** - Doubly-linked list implementation of the List interface. May provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list. Useful for queues and double-ended queues (deques).
  - **HashMap** - Hash table implementation of the Map interface. (Essentially an unsynchronized Hashtable that supports null keys and values.) The best all-around implementation of the Map interface.
  - **TreeMap** - Red-black tree implementation of the SortedMap interface.

- **Wrapper Implementations** - Functionality-enhancing implementations for use with other implementations. Accessed solely through static factory methods.
  - **Collections.unmodifiableInterface** - Return an unmodifiable view of a specified collection that throws an UnsupportedOperation Exception if the user attempts to modify it.
A design rationale saves you hassle and provides a testament to history

Java Collections API Design FAQ

This document answers frequently asked questions concerning the design of the Java collections framework. It is derived from the large volume of traffic on the collections-comments alias. It serves as a design rationale for the collections framework.

Core Interfaces - General Questions

1. Why don’t you support immutability directly in the core collection interfaces so that you can do away with optional operations (and UnsupportedOperation Exception)?
2. Won’t programmers have to surround any code that calls optional operations with a try-catch clause in case they throw an UnsupportedOperation Exception?
3. Why isn’t there a core interface for "bags" (AKA multisets)?
4. Why don’t you provide for "gating functions" that facilitate the implementation of type-safe collections?
5. Why didn’t you use "Beans-style names" for consistency?

Collection Interface

1. Why doesn’t Collection extend Cloneable and Serializable?
2. Why don’t you provide an "apply" method in Collection to apply a given method ("upcall") to all the elements of the Collection?
3. Why didn’t you provide a "Predicate" interface, and related methods (e.g., a method to find the first element in the Collection satisfying the predicate)?
4. Why don’t you provide a form of the addAll method that takes an Enumeration (or an Iterator)?
5. Why don’t the concrete implementations in the JDK have Enumeration (or Iterator) constructors?
Outline

I. The initial release of the collections API
II. Design of the first release
III. Evolution
IV. Code example
V. Critique
A wonderful source of use cases

“Good artists copy, great artists steal.” – Pablo Picasso
You must maintain an *issues list*

- Centralizes all open and closed design issues
- List pros and cons for each possible decision
- Essential for efficient progress
- Forms the basis of a design rationale
The first draft of API was not so nice

- Map was called Table
- No HashMap, only Hashtable
- No algorithms (Collections, Arrays)
- Contained some unbelievable garbage
This interface must be implemented by Collections and Tables that are views on some backing collection. (It is necessary to implement this interface only if the backing collection is not encapsulated by this Collection or Table; that is, if the backing collection might conceivably be accessed in some way other than through this Collection or Table.) This allows users to detect potential aliasing between collections. If a user attempts to modify one collection object while iterating over another, and they are in fact views on the same backing object, the iteration may behave erratically. However, these problems can be prevented by recognizing the situation, and "defensively copying" the Collection over which iteration is to take place, prior to the iteration.

public interface Alias {
    /**
     * Returns the identityHashCode of the object "ultimately backing" this collection, or zero if the backing object is undefined or unknown.
     * The purpose of this method is to allow the programmer to determine when the possibility of aliasing exists between two collections (in other words, modifying one collection could affect the other). This is critical if the programmer wants to iterate over one collection and modify another; if the two collections are aliases, the effects of the iteration are undefined, and it could loop forever. To avoid this behavior, the careful programmer must "defensively copy" the collection prior to iterating over it whenever the possibility of aliasing exists.
     * @return the identityHashCode of the object "ultimately backing" this collection, or zero if the backing object is undefined or unknown.
     * @since JDK1.2
     */
    int backingObjectId();
}

* If this collection is a view on another Object that implements Alias, this method must return the backingObjectId of the backing Object. (To avoid the cost of recursive calls to this method, the backingObjectId may be cached at creation time).
* For all collections backed by a particular "external data source" (a SQL database, for example), this method must return the same value. The identityHashCode of a "proxy" Object created just for this purpose will do nicely, as will a pseudo-random integer permanently associated with the external data source.
* For any collection backed by multiple Objects (a "concatenation view" of two Lists, for instance), this method must return zero. Similarly, for any view collection for which it cannot be determined what Object backs the collection, this method must return zero. It is always safe for a collection to return zero as its backingObjectId, but doing so when it is not necessary will lead to inefficiency.
* The possibility of aliasing between two collections exists iff any of the following conditions are true: (case sensitive)
  * The two collections are the same Object.
  * Either collection implements Alias and has a backingObjectId that is the identityHashCode of the other collection.
  * Either collection implements Alias and has a backingObjectId of zero.
  * Both collections implement Alias and they have equal backingObjectId's.
* @see java.lang.System#identityHashCode
* @since JDK1.2

Automatic alias detection
A horrible idea that died on the vine
I received a *lot* of feedback

- Initially from a small circle of colleagues
  - Some *very* good advice
  - Some not so good
- Then from the public at large: beta releases
  - Hundreds of messages
  - Many API flaws were fixed in this stage
  - I put up with a lot of flaming
Review from a **very** senior engineer

<table>
<thead>
<tr>
<th>API</th>
<th>vote</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>yes</td>
<td>But remove binarySearch* and toList</td>
</tr>
<tr>
<td>BasicCollection</td>
<td>no</td>
<td>I don't expect lots of collection classes</td>
</tr>
<tr>
<td>BasicList</td>
<td>no</td>
<td>see List below</td>
</tr>
<tr>
<td>Collection</td>
<td>yes</td>
<td>But cut toArray</td>
</tr>
<tr>
<td>Comparator</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>DoublyLinkedList</td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td>HashSet</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td>List</td>
<td>no</td>
<td>I'd like to say yes, but it's just way bigger than I was expecting</td>
</tr>
<tr>
<td>RemovalEnumeration</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>yes</td>
<td>BUT IT NEEDS A DIFFERENT NAME</td>
</tr>
<tr>
<td>TreeSet</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

I'm generally not keen on the toArray methods because they add complexity.

Similarly, I don't think that the table Entry subclass or the various views mechanisms carry their weight.
## III. Evolution of Java collections

<table>
<thead>
<tr>
<th>Release, Year</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 1.0, 1996</td>
<td>Java Released: Vector, Hashtable, Enumeration</td>
</tr>
<tr>
<td>JDK 1.1, 1996</td>
<td>(No API changes)</td>
</tr>
<tr>
<td>J2SE 1.2, 1998</td>
<td>Collections framework added</td>
</tr>
<tr>
<td>J2SE 1.3, 2000</td>
<td>(No API changes)</td>
</tr>
<tr>
<td>J2SE 1.4, 2002</td>
<td>LinkedHash{Map,Set}, IdentityHashSet, 6 new algorithms</td>
</tr>
<tr>
<td>J2SE 5.0, 2004</td>
<td>Generics, for-each, enums: generified everything, Iterable Queue, Enum{Set,Map}, concurrent collections</td>
</tr>
<tr>
<td>Java 6, 2006</td>
<td>Deque, Navigable{Set,Map}, newSetFromMap, asLifoQueue</td>
</tr>
<tr>
<td>Java 7, 2011</td>
<td>No API changes. Improved sorts &amp; defensive hashing</td>
</tr>
<tr>
<td>Java 8, 2014</td>
<td>Lambdas (+ streams and internal iterators)</td>
</tr>
</tbody>
</table>
IV. Example – How to find anagrams

• Alphabetize the characters in each word
  – cat → act, dog → dgo, mouse → emosu
  – Resulting string is called *alphagram*

• Anagrams share the same alphagram!
  – stop → opst, post → opst, tops → opst, opts → opst

• So go through word list making “multimap” from alphagram to word!
How to find anagrams in Java (1)

```java
public static void main(String[] args) throws IOException {
    // Read words from file and put into a simulated multimap
    Map<String, List<String>> groups = new HashMap<>();
    try (Scanner s = new Scanner(new File(args[0]))) {
        while (s.hasNext()) {
            String word = s.next();
            String alpha = alphabetize(word);
            List<String> group = groups.get(alpha);
            if (group == null)
                groups.put(alpha, group = new ArrayList<>());
            group.add(word);
        }
    }
}
```
How to find anagrams in Java (2)

```java
// Print all anagram groups above size threshold
int minGroupSize = Integer.parseInt(args[1]);
for (List<String> group : groups.values())
    if (group.size() >= minGroupSize)
        System.out.println(group.size() + ": " + group);
}

// Returns the alphagram for a string
private static String alphabetize(String s) {
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
```
Demo – Anagrams
Two slides in Java vs. a chapter in STL

Java’s verbosity is somewhat exaggerated
V. Critique

Some things I wish I’d done differently

• Algorithms should return collection, not void or boolean
  – Turns ugly multiliners into nice one-liners

private static String alphabetize(String s) {
    return new String(Arrays.sort(s.toCharArray()));
}

• Collection should have get(), remove()
  – Queue and Deque eventually did this

• Sorted{Set,Map} should have proper navigation
  – Navigable{Set,Map} are warts
Conclusion

• It takes a lot of work to make something that appears obvious
  – Coherent, unified vision
  – Willingness to listen to others
  – Flexibility to accept change
  – Tenacity to resist change
  – Good documentation!

• It’s worth the effort!
  – A solid foundation can last two decades