Get started with the Java Collections Framework

By Dan Becker
JavaWorld | Nov 1, 1998

JDK 1.2 introduces a new framework for collections of objects, called the Java Collections Framework. "Oh no," you groan, "not another API, not another framework to learn!" But wait, before you turn away, hear me out: the Collections framework is worth your effort and will benefit your programming in many ways. Three big benefits come immediately to mind:

• It dramatically increases the readability of your collections by providing a standard set of interfaces to be used by many programmers in many applications.

• It makes your code more flexible by allowing you to pass and return interfaces instead of concrete classes, generalizing your code rather than locking it down.

• It offers many specific implementations of the interfaces, allowing you to choose the collection that is most fitting and offers the highest performance for your needs.

And that's just for starters.

Our tour of the framework will begin with an overview of the advantages it provides for storing sets of objects. As you'll soon discover, because your old workhorse friends HashMap and
**Vector** support the new API, your programs will be uniform and concise -- something you and the developers accessing your code will certainly cheer about.

After our preliminary discussion, we'll dig deeper into the details.

**The Java Collections advantage: An overview**

Before Collections made its most welcome debut, the standard methods for grouping Java objects were via the array, the **Vector**, and the **Hashtable**. All three of these collections have different methods and syntax for accessing members: arrays use the square bracket ([[]]) symbols, **Vector** uses the **elementAt** method, and **HashTable** uses **get** and **put** methods. These differences have long led programmers down the path to inconsistency in implementing their own collections -- some emulate the **Vector** access methods and some emulate the **Enumeration** interface.

To further complicate matters, most of the **Vector** methods are marked as final; that is, you cannot extend the **Vector** class to implement a similar sort of collection. We could create a collection class that looked like a **Vector** and acted like a **Vector**, but it couldn't be passed to a method that takes a **Vector** as a parameter.

Finally, none of the collections (array, **Vector** or **Hashtable**) implements a standard member access interface. As programmers developed algorithms (like sorts) to manipulate collections, a heated discourse erupted on what object to pass to the algorithm. Should you pass an array or a **Vector**? Should you implement both interfaces? Talk about duplication and confusion.

Thankfully, the Java Collections Framework remedies these problems and offers a number of advantages over using no
framework or using the **Vector** and **Hashtable**:

- **A usable set of collection interfaces** By implementing one of the basic interfaces -- **Collection**, **Set**, **List**, or **Map** -- you ensure your class conforms to a common API and becomes more regular and easily understood. So, whether you are implementing an SQL database, a color swatch matcher, or a remote chat application, if you implement the **Collection** interface, the operations on your collection of objects are well-known to your users. The standard interfaces also simplify the passing and returning of collections to and from class methods and allow the methods to work on a wider variety of collections.

- **A basic set of collection implementations** In addition to the trusty **Hashtable** and **Vector**, which have been updated to implement the **Collection** interfaces, new collection implementations have been added, including **HashSet** and **TreeSet**, **ArrayList** and **LinkedList**, and **HashMap** and **Map**. Using an existing, common implementation makes your code shorter and quicker to download. Also, using existing Core Java code core ensures that any improvements to the base code will also improve the performance of your code.

- **Other useful enhancements** Each collection now returns an **Iterator**, an improved type of **Enumeration** that allows element operations such as insertion and deletion. The **Iterator** is "fail-fast," which means you get an exception if the list you're iterating is changed by another user. Also, list-based collections such as **Vector** return a **ListIterator** that allow bi-directional iteration and updating. Several collections (**TreeSet** and **TreeMap**) implicitly support ordering. Use these classes to maintain a
sorted list with no effort. You can find the smallest and largest elements or perform a binary search to improve the performance of large lists. You can sort other collections by providing a collection-compare method (a Comparator object) or an object-compare method (the Comparable interface). Finally, a static class Collections provides unmodifiable (read-only) and synchronized versions of existing collections. The unmodifiable classes are helpful to prevent unwanted changes to a collection. The synchronized version of a collection is a necessity for multithreaded programs.

The Java Collections Framework is part of Core Java and is contained in the java.util.collections package of JDK 1.2. The framework is also available as a package for JDK 1.1 (see Resources).

Let us now look more closely at these advantages by exercising the Java Collections Framework with some code of our own. The first advantage of the Java Collections Framework is a consistent and regular API. The API is codified in a basic set of interfaces, Collection, Set, List, or Map. The Collection interface contains basic collection operations such as adding, removing, and tests for membership (containment). Any implementation of a collection, whether it is one provided by the Java Collections Framework or one of your own creations, will support one of these interfaces. Because the Collections framework is regular and consistent, you will learn a large portion of the frameworks simply by learning these interfaces.

Both Set and List implement the Collection interface. The Set interface is identical to the Collection interface except for an additional method, toArray, which converts a Set to an
Object array. The List interface also implements the Collection interface, but provides many accessors that use an integer index into the list. For instance, get, remove, and set all take an integer that affects the indexed element in the list. The Map interface is not derived from collection, but provides an interface similar to the methods in java.util.Hashtable. Keys are used to put and get values. Each of these interfaces are described in following code examples.

The following code segment demonstrates how to perform many Collection operations on HashSet, a basic collection that implements the Set interface. A HashSet is simply a set that doesn't allow duplicate elements and doesn't order or position its elements. The code shows how you create a basic collection and add, remove, and test for elements. Because Vector now supports the Collection interface, you can also execute this code on a vector, which you can test by changing the HashSet declaration and constructor to a Vector.

```java
import java.util.collections.*;

class CollectionTest {
    // Statics
    public static void main(String[] args) {
        System.out.println("Collection Test");
        // Create a collection
        HashSet collection = new HashSet();
        // Adding
        String dog1 = "Max", dog2 = "Bailey", dog3 = "Harriet";
        collection.add(dog1);
        collection.add(dog2);
        collection.add(dog3);
        // Sizing
```
System.out.println("Collection created" +
", size=" + collection.size() +
", isEmpty=" + collection.isEmpty() );

// Containment
System.out.println("Collection contains " + dog3 +
": " + collection.contains( dog3 ) );

// Iteration. Iterator supports hasNext, next, remove
System.out.println("Collection iteration (unsorted):" );
Iterator iterator = collection.iterator();
while ( iterator.hasNext() )
    System.out.println("   "+ iterator.next() );

// Removing
collection.remove( dog1 );
collection.clear();
}
}

Let's now build on our basic knowledge of collections and look at other interfaces and implementations in the Java Collections Framework.

Good concrete implementations
We have exercised the Collection interface on a concrete collection, the HashSet. Let's now look at the complete set of concrete collection implementations provided in the Java Collections framework. (See the Resources section for a link to Sun's annotated outline of the Java Collections framework.)

<table>
<thead>
<tr>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Table</td>
</tr>
<tr>
<td>Resizable</td>
</tr>
<tr>
<td>Balanced Tree (Sorted)</td>
</tr>
<tr>
<td>Linked List</td>
</tr>
<tr>
<td>Interfaces</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>List</td>
</tr>
<tr>
<td>Map</td>
</tr>
</tbody>
</table>

Implementations marked with an asterix (*) make no sense or provide no compelling reason to implement. For instance, providing a List interface to a Hash Table makes no sense because there is no notion of order in a Hash Table. Similarly, there is no Map interface for a Linked List because a list has no notion of table lookup.

Let's now exercise the List interface by operating on concrete implementations that implement the List interface, the ArrayList, and the LinkedList. The code below is similar to the previous example, but it performs many List operations.

```java
import java.util.*;

public class ListTest {
    // Statics
    public static void main(String[] args) {
        System.out.println("List Test");
        // Create a collection
        ArrayList list = new ArrayList();
        // Adding
        String[] toys = {"Shoe", "Ball", "Frisbee"};
        list.addAll(Arrays.asList(toys));
        // Sizing
        System.out.println("List created" +
            ", size=" + list.size() +
            ", isEmpty=" + list.isEmpty());
        // Iteration using indexes.
        System.out.println("List iteration (unsorted):" );
        for (int i = 0; i < list.size(); i++)
```
As with the first example, it's simple to swap out one implementation for another. You can use a `LinkedList` instead of an `ArrayList` simply by changing the line with the `ArrayList` constructor. Similarly, you can use a `Vector`, which now supports the `List` interface.

When deciding between these two implementations, you should consider whether the list is volatile (grows and shrinks often) and whether access is random or ordered. My own tests have shown that the `ArrayList` generally outperforms the `LinkedList` and the new `Vector`.

Notice how we add elements to the list: we use the `addAll` method and the static method `Arrays.asList`. This static method is one of the most useful utility methods in the Collections framework because it allows any array to be viewed as a `List`. Now an array may be used anywhere a `Collection` is needed.

Notice that I iterate through the list via an indexed accessor, `get`, and the `ListIterator` class. In addition to reverse iteration, the
ListIterator class allows you to add, remove, and set any element in the list at the point addressed by the ListIterator. This approach is quite useful for filtering or updating a list on an element-by-element basis.

The last basic interface in the Java Collections Framework is the Map. This interface is implemented with two new concrete implementations, the TreeMap and the HashMap. The TreeMap is a balanced tree implementation that sorts elements by the key. Let's illustrate the use of the Map interface with a simple example that shows how to add, query, and clear a collection. This example, which uses the HashMap class, is not much different from how we used the Hashtable prior to the debut of the Collections framework. Now, with the update of Hashtable to support the Map interface, you can swap out the line that instantiates the HashMap and replace it with an instantiation of the Hashtable.

```java
import com.sun.java.util.collections.*;

class HashMapTest {
    // Statics
    public static void main( String [] args ) {
        System.out.println( "Collection HashMap Test" );
        HashMap collection1 = new HashMap();
        // Test the Collection interface
        System.out.println( "Collection 1 created, size=" +
        collection1.size() +
        ", isEmpty=" + collection1.isEmpty() );
        // Adding
        collection1.put( new String( "Harriet" ), new String( "Bone" ) );
    }
}
```
collection1.put( new String( "Bailey" ), new String( "Big Chair" ) );
collection1.put( new String( "Max" ), new String( "Tennis Ball" ) );
System.out.println( "Collection 1 populated, size=" +
collection1.size() +
    ", isEmpty=" + collection1.isEmpty() );
    // Test Containment/Access
    String key = new String( "Harriet" );
    if ( collection1.containsKey( key ) )
        System.out.println( "Collection 1 access, key=" + key + ",
value=" +
            (String) collection1.get( key ) );
    // Test iteration of keys and values
    Set keys = collection1.keySet();
    System.out.println( "Collection 1 iteration (unsorted),
collection contains keys:" );
    Iterator iterator = keys.iterator();
    while ( iterator.hasNext() )
        System.out.println( "   " + iterator.next() );
    collection1.clear();
    System.out.println( "Collection 1 cleared, size=" +
collection1.size() +
    ", isEmpty=" + collection1.isEmpty() );
}

We've covered most of the interfaces and implementations in the Java Collections framework, and we're ready to check out some of the additional capabilities Collections offers us.
Other capabilities
Many of the additional features such as sorting and synchronization are encapsulated in the `Collections` and `Arrays` classes. These classes, which will appear throughout the following discussion, provide static methods for acting on collections.

### Sorting a collection

We'll begin by exploring sorting. Two of the concrete implementations in the Java Collections Framework provide easy means to maintain a sorted collection: `TreeSet` and `TreeMap`. In fact, these two classes implement the `SortedSet` and `SortedMap` interfaces, which are similar to their unsorted counterparts except that they provide methods to access first and last elements and portions of the sorted collections.

There are two basic techniques for maintaining a sorted collection. The first uses one of the sorted collection classes and provides the `collection` with an object that implements a comparison via the `Comparator` interface. For example, going back to our first code example, we can sort our collection by creating a `StringComparator` and adding it to the end of the code, as shown here:

```java
// This class sorts two String objects.
class StringComparator implements Comparator {
    public int compare( Object object1, Object object2 ) {
        return ((String) object1).compareTo( (String) object2 );
    }
}
```

Next, we need to change the collection from a `HashSet` (unsorted) to a `HashMap` (sorted with our `StringComparator` by using the following constructor:

```java
TreeSet collection = new TreeSet( new StringComparator() );
```
Rerun the example and you should see that the iteration is performed in sorted order. Because the collection is ordered, you should now be able to find the min and the max elements using the static class `Collections`.

The second technique is to implement natural ordering of a class by making the class implement the `Comparable` interface. This technique adds a single `compareTo` method to a class, which then returns 0 for equal objects, less than 0 if the first parameter is less than the second, or greater than 0 if the first parameter is greater than the second. In Java 1.2, the `String` class (but not `StringBuffer`) implements the `Comparable` interface. Any comparable object can be placed in a sorted collection, and the collection order is maintained automatically by the collection.

You can also sort `Lists` by handing them to the `Collections` class. One static `sort` method takes a single `List` parameter that specifies a naturally ordered class (one that implements the `Comparable` interface). A second static `sort` method takes a `Comparator` object for other classes that do not implement the `Comparable` interface.

Unmodifiable collections

The `Collections` class provides many static factory methods (like `Collection.unmodifiableCollection` and `Collection.unmodifiableSet`) for providing unmodifiable or immutable collections. In fact, there is one method for each of the basic collection interfaces. These methods are extremely useful to ensure that no one modifies your collection. For instance, if you want to allow others to see your list but not change it, you may implement a method that returns an unmodifiable view of your collection. Here's an example:

```java
List getUnmodifiableView() {
```
return Collections.unmodifiableList( this );
}

This code will throw an UnsupportedOperationException, one of the RuntimeExceptions, if someone tries to add or remove an element from the list. Unfortunately, the unmodifiable views of a collection are of the same type as the original collection, which hinders compile-time type checking. Although you may pass an unmodifiable list to a method, by virtue of its type, the compiler has no way of ensuring the collection is unchanged by the method. The unmodifiable collection is checked at runtime for changes, but this is not quite as strong as compile-time checking and does not aid the compiler in code optimization. Perhaps it's time for Java to emulate C++'s const and add another modifier signifying immutability of any method, class, or object.

Synchronized collections
Finally, note that none of the concrete methods mentioned thus far support multithreaded access in the manner that the Vectors and Hashtable did. In other words, none of the methods on the concrete implementations are synchronized and none of the implementations are thread-safe. You must support thread safety yourself.

This may seem like a major omission, but in actuality it's not really a big problem. The Collections class provides a synchronized version of each of the collection implementations. You ensure thread safety by using the synchronized version of a collection and synchronizing on the returned object. For example, we can ensure thread safety on a List by using the following construct:
List dogs = synchronized List( new ArrayList() );
synchronized( list ) {
    Iterator iterator = list.iterator(); // Must be in synchronized block
    while ( iterator.hasNext() )
        nonAtomicOperation( iterator.next() );
}

Many programmers are already using these types of synchronized blocks around Vectors and Hashables, so the new considerations aren't too significant.

Conclusion
That concludes our survey of the new Java Collections Framework. We covered a lot of territory, so let's briefly review. We began with a look at the interfaces, which are used through the API and are useful for any new collections we may create. These interfaces are intuitive and provide a common way for all Java programmers to access collections. By implementing a common interface in the Collections framework, you make it easy for other programmers to access your collection, you reduce the time it takes for others to learn your class, and you make your class more useful.

We also examined the basic concrete implementations provided with the framework. You can use these basic collections to implement any generic collection of Java objects. Like the existing Vector and Hashtable, these new collection implementations will cover a majority of your needs as a developer.

Dan Becker works in the Network Computing Software Division of IBM Corporation in Austin, TX