Java Collections Framework

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial takes you on an extended tour of the Java Collections Framework. The tutorial starts with a few simple programming examples for beginners and experts alike, to get started with the Collections Framework quickly. The tutorial continues with a discussion of sets and maps, their properties, and how their mathematical definition differs from the `Set`, `Map`, and `Collection` definitions within the Collections Framework. A section on the history of Java Collections Framework clears up some of the confusion around the proliferation of set- and map-like classes. This tutorial includes a thorough presentation of all the interfaces and their implementation classes in the Collections Framework. The tutorial explores the algorithm support for the collections, as well as working with collections in a thread-safe and read-only manner. In addition, the tutorial includes a discussion of using a subset of the Collections Framework with JDK 1.1. The tutorial concludes with an introduction of JGL, a widely used algorithm and data structure library from ObjectSpace that predates the Java Collections Framework.

Concepts

At the end of this tutorial you will know the following:

* The mathematical meaning of set, map, and collection
* The six key interfaces of the Collections Framework

Objectives

By the end of this tutorial, you will know how to do the following:

* Use the concrete collection implementations
* Apply sorting and searching through collections
* Use read-only and thread-safe collections

Contact

jGuru has been dedicated to promoting the growth of the Java technology community through evangelism, education, and software since 1995. You can find out more about their activities, including their huge collection of FAQs at *jGuru.com* . To send feedback to jGuru about this tutorial, send mail to *producer@jguru.com* .

Course author: John Zukowski does strategic Java consulting for *JZ Ventures, Inc.* . His latest book is *"Java Collections"* (Apress, May 2001).

Section 2. Collections Framework

Introduction

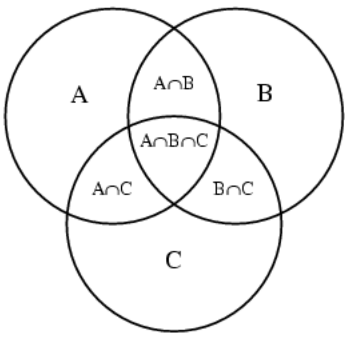
This tutorial takes you on an extended tour of the Collections Framework, first introduced with the Java 2 platform, Standard Edition, version 1.2. The Collections Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a *collection*. The framework provides a convenient API to many of the abstract data types familiar from computer science data structure curriculum: maps, sets, lists, trees, arrays, hashtables, and other collections. Because of their object-oriented design, the Java classes in the Collections Framework encapsulate both the data structures and the algorithms associated with these abstractions. The framework provides a standard programming interface to many of the most common abstractions, without burdening the programmer with too many procedures and interfaces. The operations supported by the collections framework nevertheless permit the programmer to easily define higher-level data abstractions, such as stacks, queues, and thread-safe collections.

One thing worth noting early on is that while the framework is included with the Java 2 platform, a subset form is available for use with Java 1.1 run-time environments. The framework subset is discussed in Working with the Collections Framework support in JDK 1.1 on page 33.

Before diving into the Collections Framework, it helps to understand some of the terminology and set theory involved when working with the framework.

Mathematical background

In common usage, a *collection* is the same as the intuitive, mathematical concept of a *set*. A set is just a group of unique items, meaning that the group contains no duplicates. The Collections Framework, in fact, includes a `Set` interface, and a number of concrete `Set` classes. But the formal notion of a set predates Java technology by a century, when the British mathematician George Boole defined it in formal logic. Most people learned some set theory in elementary school when introduced to "set intersection" and "set union" through the familiar Venn Diagrams:



Some real-world examples of sets include the following:

* The set of uppercase letters 'A' through 'Z'
* The set of non-negative integers {0, 1, 2 ...}

* The set of reserved Java programming language keywords {'import', 'class', 'public', 'protected'...}
* A set of people (friends, employees, clients, ...)
* The set of records returned by a database query
* The set of `Component` objects in a `Container`
* The set of all pairs
* The empty set {}

Sets have the following basic properties:

* They contains only one instance of each item
* They may be finite or infinite
* They can define abstract concepts

Sets are fundamental to logic, mathematics, and computer science, but also practical in everyday applications in business and systems. The idea of a "connection pool" is a set of open connections to a database server. Web servers have to manage sets of clients and connections. File descriptors provide another example of a set in the operating system.

A *map* is a special kind of set. It is a set of pairs, each pair representing a one-directional mapping from one element to another. Some examples of maps are:

* The map of IP addresses to domain names (DNS)
* A map from keys to database records
* A dictionary (words mapped to meanings)
* The conversion from base 2 to base 10

Like sets, the idea behind a map is much older than the Java programming language, older even than computer science. Sets and maps are important tools in mathematics and their properties are well understood. People also long recognized the usefulness of solving programming problems with sets and maps. A language called SETL (Set Language) invented in 1969 included sets as one of its only primitive data types (SETL also included garbage collection -- not widely accepted until Java technology was developed in the 1990s). Although sets and maps appear in many languages including C++, the Collections Framework is perhaps the best designed set and map package yet written for a popular language. (Users of C++ Standard Template Library (STL) and Smalltalk's collection hierarchy might argue that last point.)

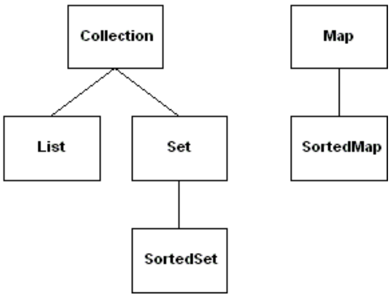
Also because they are sets, maps can be finite or infinite. An example of an infinite map is the conversion from base 2 to base 10. Unfortunately, the Collections Framework does not support infinite maps -- sometimes a mathematical function, formula, or algorithm is preferred. But when a problem can be solved with a finite map, the Collections Framework provides the Java programmer with a useful API.

Because the Collections Framework has formal definitions for the classes `Set`,
, and `Map`, you'll notice the lower case words *set*, *collection*, and *map* to distinguish the implementation from the concept.

Section 3. Collection interfaces and classes

Introduction

Now that you have some set theory under your belt, you should be able to understand the Collections Framework more easily. The Collections Framework is made up of a set of interfaces for working with groups of objects. The different interfaces describe the different types of groups. For the most part, once you understand the interfaces, you understand the framework. While you always need to create specific implementations of the interfaces, access to the actual collection should be restricted to the use of the interface methods, thus allowing you to change the underlying data structure, without altering the rest of your code. The following diagrams shows the framework interface hierarchy.



One might think that `Map` would extend `Collection`. In mathematics, a map is just a collection of pairs. In the Collections Framework, however, the interfaces `Map` and `Collection` are distinct with no lineage in the hierarchy. The reasons for this distinction have to do with the ways that `Set` and `Map` are used in the Java libraries. The typical application of a `Map` is to provide access to values stored by keys. The set of collection operations are all there, but you work with a key-value pair instead of an isolated element. `Map` is therefore designed to support the basic operations of *`get()`* and *`put()`*, which are not required by `Set`. Moreover, there are methods that return `Set` views of `Map` objects:

```
Set set = aMap.keySet();
```

When designing software with the Collections Framework, it is useful to remember the following hierarchical relationships of the four basic interfaces of the framework:

* The `Collection` interface is a group of objects, with duplicates allowed.
* The `Set` interface extends `Collection` but forbids duplicates.
* The `List` interface extends `Collection`, allows duplicates, and introduces positional indexing.
* The `Map` interface extends neither `Set` nor `Collection`.

Moving on to the framework implementations, the concrete collection classes follow a naming convention, combining the underlying data structure with the framework interface. The following table shows the six collection implementations introduced with the Java 2 framework, in addition to the four historical collection classes. For information on how the historical collection classes changed, like how `Hashtable` was reworked into the framework, see the Historical collection classes on page 25.

Interface	Implementation	Historical
`Set`	`HashSet`	

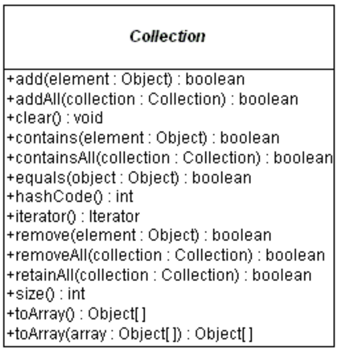
	TreeSet	
List	ArrayList	Vector
	LinkedList	Stack
Map	HashMap	Hashtable
	TreeMap	Properties

There are no implementations of the `Collection` interface. The historical collection classes are called such because they have been around since the 1.0 release of the Java class libraries.

If you are moving from the historical collection classes to the new framework classes, one of the primary differences is that all operations are unsynchronized with the new classes. While you can add synchronization to the new classes, you cannot remove it from the old.

Collection interface

The `Collection` interface is used to represent any group of objects, or elements. You use the interface when you wish to work with a group of elements in as general a manner as possible. Here is a list of the public methods of `Collection` in Unified Modeling Language (UML) notation.



The interface supports basic operations like adding and removing. When you try to remove an element, only a single instance of the element in the collection is removed, if present.

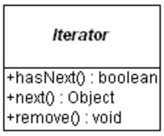
* `boolean add(Object element)`
* `boolean remove(Object element)`

The `Collection` interface also supports query operations:

* `int size()`
* `boolean isEmpty()`
* `boolean contains(Object element)`
* `Iterator iterator()`

Iterator interface

The *`iterator()`* method of the `Collection` interface returns an `Iterator`. An `Iterator` is similar to the `Enumeration` interface, which you may already be familiar with, and will be described in Enumeration interface on page 26. With the `Iterator` interface methods, you can traverse a collection from start to finish and safely remove elements from the underlying `Collection`:



The *`remove()`* method is optionally supported by the underlying collection. When called, and supported, the element returned by the last *`next()`* call is removed. To demonstrate, the following code shows the use of the `Iterator` interface for a general `Collection`:

```
Collection collection = ...;
 Iterator iterator = collection.iterator();
 while (iterator.hasNext()) {
   Object element = iterator.next();
   if (removalCheck(element)) {
     iterator.remove();
   }
 }
```

Group operations

Other operations the `Collection` interface supports are tasks done on groups of elements or the entire collection at once:

* `boolean containsAll(Collection collection)`
* `boolean addAll(Collection collection)`
* `void clear()`
* `void removeAll(Collection collection)`
* `void retainAll(Collection collection)`

The *`containsAll()`* method allows you to discover if the current collection contains all the elements of another collection, a *subset*. The remaining methods are optional, in that a specific collection might not support the altering of the collection. The *`addAll()`* method ensures all elements from another collection are added to the current collection, usually a *union*. The *`clear()`* method removes all elements from the current collection. The *`removeAll()`* method is like *`clear()`* but only removes a subset of elements. The *`retainAll()`* method is similar to the *`removeAll()`* method but does what might be perceived as the opposite: it removes from the current collection those elements not in the other collection, an *intersection*.

The remaining two interface methods, which convert a `Collection` to an array, will be discussed in Converting from new collections to historical collections on page 32.

AbstractCollection class

The `AbstractCollection` class provides the basis for the concrete collections framework classes. While you are free to implement all the methods of the `Collection` interface yourself, the `AbstractCollection` class provides implementations for all the methods, except for the `iterator()` and `size()` methods, which are implemented in the appropriate subclass. Optional methods like *`add()`* will throw an exception if the subclass doesn't override the behavior.

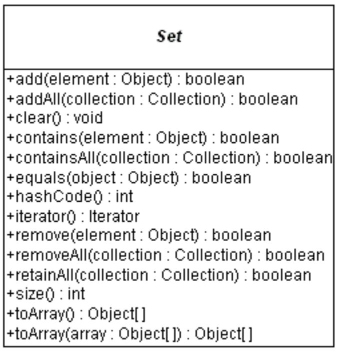
Collections Framework design concerns

In the creation of the Collections Framework, the Sun development team needed to provide flexible interfaces that manipulated groups of elements. To keep the design simple, instead of providing separate interfaces for optional capabilities, the interfaces define all the methods an implementation class may provide. However, *some* of the interface methods are optional. Because an interface implementation must provide implementations for all the interface methods, there needed to be a way for a caller to know if an optional method is not supported. The manner the framework development team chose to signal callers when an optional method is called was to throw an `UnsupportedOperationException`. If in the course of using a collection an `UnsupportedOperationException` is thrown, then the operation failed because it is not supported. To avoid having to place all collection operations within a `try-catch` block, the `UnsupportedOperationException` class is an extension of the `RuntimeException` class.

In addition to handling optional operations with a run-time exception, the iterators for the concrete collection implementations are *fail-fast* . That means that if you are using an `Iterator` to traverse a collection while the underlying collection is being modified by another thread, then the `Iterator` fails immediately by throwing a `ConcurrentModificationException` (another `RuntimeException`). That means the next time an `Iterator` method is called, and the underlying collection has been modified, the `ConcurrentModificationException` exception gets thrown.

Set interface

The `Set` interface extends the `Collection` interface and, by definition, forbids duplicates within the collection. All the original methods are present and no new methods are introduced. The concrete `Set` implementation classes rely on the *`equals()`* method of the object added to check for equality.



HashSet and TreeSet classes

The Collections Framework provides two general-purpose implementations of the `Set` interface: `HashSet` and `TreeSet`. More often than not, you will use a `HashSet` for storing your duplicate-free collection. For efficiency, objects added to a `HashSet` need to implement the `hashCode()` method in a manner that properly distributes the hash codes. While most system classes override the default *`hashCode()`* implementation in `Object`, when creating your own classes to add to a HashSet remember to override hashCode(). The TreeSet implementation is useful when you need to extract elements from a collection in a sorted manner. In order to work properly, elements added to a TreeSet must be sortable. The Collections Framework adds support for Comparable elements and will be covered in detail in "Comparable interface" in Sorting on page 17. For now, just assume a tree knows how to keep elements of the java.lang wrapper classes sorted. It is generally faster to add elements to a HashSet, then convert the collection to a TreeSet for sorted traversal.

To optimize HashSet space usage, you can tune the initial capacity and load factor. The TreeSet has no tuning options, as the tree is always balanced, ensuring log(n) performance for insertions, deletions, and queries.

Both HashSet and TreeSet implement the Cloneable interface.

Set usage example

To demonstrate the use of the concrete Set classes, the following program creates a HashSet and adds a group of names, including one name twice. The program then prints out the list of names in the set, demonstrating the duplicate name isn't present. Next, the program treats the set as a TreeSet and displays the list sorted.

```
import java.util.*;

 public class SetExample {
   public static void main(String args[]) {
     Set set = new HashSet();
     set.add("Bernadine");
     set.add("Elizabeth");
     set.add("Gene");
     set.add("Elizabeth");
     set.add("Clara");
     System.out.println(set);
     Set sortedSet = new TreeSet(set);
     System.out.println(sortedSet);
   }
 }
```

Running the program produces the following output. Notice that the duplicate entry is only present once, and the second list output is sorted alphabetically.

```
[Gene, Clara, Bernadine, Elizabeth]
 [Bernadine, Clara, Elizabeth, Gene]
```

AbstractSet class

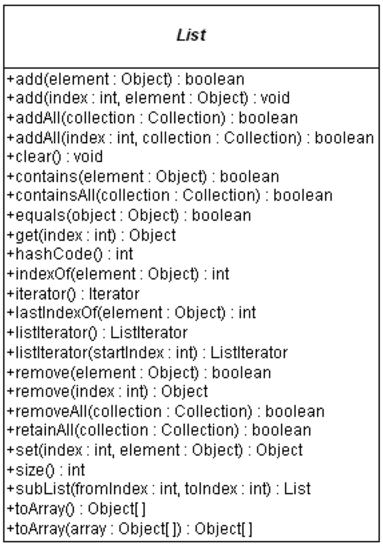
The AbstractSet class overrides the *equals()* and *hashCode()* methods to ensure two equal sets return the same hash code. Two sets are equal if they are the same size and contain the same elements. By definition, the hash code for a set is the sum of the hash codes for the elements of the set. Thus, no matter what the internal ordering of the sets, two equal sets will report the same hash code.

Exercises

* Exercise 1. How to use a HashSet for a sparse bit set on page 36
* Exercise 2. How to use a TreeSet to provide a sorted JList on page 38

List interface

The `List` interface extends the `Collection` interface to define an ordered collection, permitting duplicates. The interface adds position-oriented operations, as well as the ability to work with just a part of the list.



The position-oriented operations include the ability to insert an element or `Collection`, get an element, as well as remove or change an element. Searching for an element in a `List` can be started from the beginning or end and will report the position of the element, if found.

* `void add(int index, Object element)`
* `boolean addAll(int index, Collection collection)`
* `Object get(int index)`
* `int indexOf(Object element)`
* `int lastIndexOf(Object element)`
* `Object remove(int index)`
* `Object set(int index, Object element)`

The `List` interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position-friendly manner:

* `ListIterator listIterator()`
* `ListIterator listIterator(int startIndex)`
* `List subList(int fromIndex, int toIndex)`

In working with *`subList()`* , it is important to mention that the element at `fromIndex` is in the sublist, but the element at `toIndex` is not. This loosely maps to the following `for-loop` test cases:

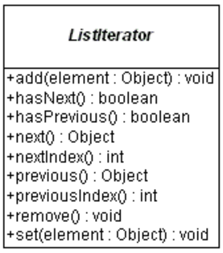
```
for (int i=fromIndex; i<toIndex; i++) {
```

```
   // process element at position i
 }
```

In addition, it should be mentioned that changes to the sublist (like `add()`, `remove()`, and `set()` calls) have an effect on the underlying `List`.

ListIterator interface

The `ListIterator` interface extends the `Iterator` interface to support bi-directional access, as well as adding or changing elements in the underlying collection.



The following source code demonstrates looping backwards through a list. Notice that the `ListIterator` is originally positioned beyond the end of the list (`list.size()`), as the index of the first element is 0.

```
List list = ...;
 ListIterator iterator = list.listIterator(list.size());
 while (iterator.hasPrevious()) {
   Object element = iterator.previous();
   // Process element
 }
```

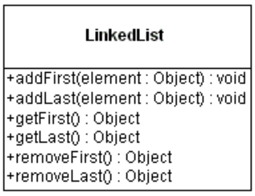
Normally, one doesn't use a `ListIterator` to alternate between going forward and backward in one iteration through the elements of a collection. While technically possible, calling `next()` immediately after `previous()` results in the same element being returned. The same thing happens when you reverse the order of the calls to next() and previous().

The `add()` operation requires a little bit of explanation also. Adding an element results in the new element being added immediately prior to the implicit cursor. Thus, calling `previous()` after adding an element would return the new element and calling `next()` would have no effect, returning what would have been the next element prior to the add operation.

ArrayList and LinkedList classes

There are two general-purpose `List` implementations in the Collections Framework: `ArrayList` and `LinkedList`. Which of the two `List` implementations you use depends on your specific needs. If you need to support random access, without inserting or removing elements from any place other than the end, then `ArrayList` offers the optimal collection. If, however, you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially, then `LinkedList` offers the better implementation.

Both `ArrayList` and `LinkedList` implement the `Cloneable` interface. In addition, `LinkedList` adds several methods for working with the elements at the ends of the list (only the new methods are shown in the following diagram):



By using these new methods, you can easily treat the `LinkedList` as a stack, queue, or other end-oriented data structure.

```
LinkedList queue = ...;
 queue.addFirst(element);
 Object object = queue.removeLast();

LinkedList stack = ...;
 stack.addFirst(element);
 Object object = stack.removeFirst();
```

The `Vector` and `Stack` classes are historical implementations of the `List` interface. They will be discussed in Vector and Stack classes on page 25.

List usage example

The following program demonstrates the use of the concrete `List` classes. The first part creates a `List` backed by an `ArrayList`. After filling the list, specific entries are retrieved. The `LinkedList` part of the example treats the `LinkedList` as a queue, adding things at the beginning of the queue and removing them from the end.

```
import java.util.*;

 public class ListExample {
   public static void main(String args[]) {
     List list = new ArrayList();
     list.add("Bernadine");
     list.add("Elizabeth");
     list.add("Gene");
     list.add("Elizabeth");
     list.add("Clara");
     System.out.println(list);
     System.out.println("2: " + list.get(2));
     System.out.println("0: " + list.get(0));
     LinkedList queue = new LinkedList();
     queue.addFirst("Bernadine");
     queue.addFirst("Elizabeth");
     queue.addFirst("Gene");
     queue.addFirst("Elizabeth");
     queue.addFirst("Clara");
     System.out.println(queue);
     queue.removeLast();
     queue.removeLast();
     System.out.println(queue);
   }
 }
```

Running the program produces the following output. Notice that unlike `Set`, `List` permits duplicates.

```
[Bernadine, Elizabeth, Gene, Elizabeth, Clara]
 2: Gene
 0: Bernadine
 [Clara, Elizabeth, Gene, Elizabeth, Bernadine]
 [Clara, Elizabeth, Gene]
```

AbstractList and AbstractSequentialList classes

There are two abstract `List` implementations classes: `AbstractList` and `AbstractSequentialList`. Like the `AbstractSet` class, they override the `equals()` and `hashCode()` methods to ensure two equal collections return the same hash code. Two sets are equal if they are the same size and contain the same elements in the same order. The `hashCode()` implementation is specified in the `List` interface definition and implemented here.

Besides the `equals()` and `hashCode()` implementations, `AbstractList` and `AbstractSequentialList` provide partial implementations of the remaining `List` methods. They make the creation of concrete list implementations easier, for random-access and sequential-access data sources, respectively. Which set of methods you need to define depends on the behavior you wish to support. The following table should help you remember which methods need to be implemented. One thing you'll *never* need to provide yourself is an implementation of the `Iterator iterator()` method.

	AbstractList	AbstractSequentialList
unmodifiable	Object get(int index) int size()	ListIterator listIterator(int index) - boolean hasNext() - Object next() - int nextIndex() - boolean hasPrevious() - Object previous() - int previousIndex() int size()
modifiable	Object get(int index) int size() Object set(int index, Object element)	ListIterator listIterator(int index) - boolean hasNext() - Object next() - int nextIndex() - boolean hasPrevious() - Object previous() - int previousIndex() int size() ListIterator - set(Object element)

variable-size and modifiable	`Object get(int index)`
	`int size()`
	`Object set(int index, Object element)`
	`add(int index, Object element)`
	`Object remove(int index)`
		`ListIterator listIterator(int index)`
		`- boolean hasNext()`
		`- Object next()`
		`- int nextIndex()`
		`- boolean hasPrevious()`
		`- Object previous()`
		`- int previousIndex()`
		`int size()`
		`ListIterator`
		`- set(Object element)`
		`ListIterator`
		`- add(Object element)`
		`- remove()`

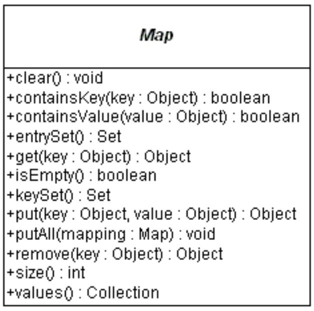
As the `Collection` interface documentation states, you should also provide two constructors, a no-argument one and one that accepts another `Collection`.

Exercise

* Exercise 3. How to use an ArrayList with a JComboBox on page 40

Map interface

The `Map` interface is not an extension of the `Collection` interface. Instead, the interface starts off its own interface hierarchy for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition.



The interface methods can be broken down into three sets of operations: altering, querying, and providing alternative views.

The alteration operations allow you to add and remove key-value pairs from the map. Both the key and value can be `null`. However, you should not add a `Map` to itself as a key or value.

* `Object put(Object key, Object value)`
* `Object remove(Object key)`
* `void putAll(Map mapping)`
* `void clear()`

The query operations allow you to check on the contents of the map:

* `Object get(Object key)`
* `boolean containsKey(Object key)`
* `boolean containsValue(Object value)`
* `int size()`
* `boolean isEmpty()`

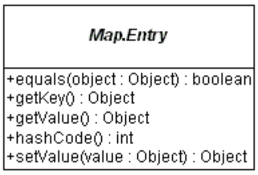
The last set of methods allow you to work with the group of keys or values as a collection.

* `public Set keySet()`
* `public Collection values()`
* `public Set entrySet()`

Because the collection of keys in a map must be unique, you get a `Set` back. Because the collection of values in a map may not be unique, you get a `Collection` back. The last method returns a `Set` of elements that implement the `Map.Entry` interface.

Map.Entry interface

The `entrySet()` method of `Map` returns a collection of objects that implement the `Map.Entry` interface. Each object in the collection is a specific key-value pair in the underlying `Map`.



Iterating through this collection, you can get the key or value, as well as change the value of each entry. However, the set of entries becomes invalid, causing the iterator behavior to be undefined, if the underlying `Map` is modified outside the `setValue()` method of the `Map.Entry` interface.

HashMap and TreeMap classes

The Collections Framework provides two general-purpose `Map` implementations: `HashMap` and `TreeMap`. As with all the concrete implementations, which implementation you use depends on your specific needs. For inserting, deleting, and locating elements in a `Map`, the `HashMap` offers the best alternative. If, however, you need to traverse the keys in a sorted order, then `TreeMap` is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a `HashMap`, then convert the map to a `TreeMap` for sorted key traversal. Using a `HashMap` requires that the class of key added have a well-defined `hashCode()` implementation. With the `TreeMap` implementation, elements added to the map must be sortable. We'll say more about this in Sorting on page 17.

To optimize `HashMap` space usage, you can tune the initial capacity and load factor. The `TreeMap` has no tuning options, as the tree is always balanced.

Both `HashMap` and `TreeMap` implement the `Cloneable` interface.

The `Hashtable` and `Properties` classes are historical implementations of the `Map` interface. They will be discussed in Dictionary, Hashtable, and Properties classes on page 26

.

Map usage example

The following program demonstrates the use of the concrete `Map` classes. The program generates a frequency count of words passed from the command line. A `HashMap` is initially used for data storage. Afterwards, the map is converted to a `TreeMap` to display the key list sorted.

```
import java.util.*;

 public class MapExample {
   public static void main(String args[]) {
     Map map = new HashMap();
     Integer ONE = new Integer(1);
     for (int i=0, n=args.length; i<n; i++) {
       String key = args[i];
       Integer frequency = (Integer)map.get(key);
       if (frequency == null) {
         frequency = ONE;
       } else {
         int value = frequency.intValue();
         frequency = new Integer(value + 1);
       }
       map.put(key, frequency);
     }
     System.out.println(map);
     Map sortedMap = new TreeMap(map);
     System.out.println(sortedMap);
   }
 }
```

Running the program with the text from *Article 3* of the Bill of Rights produces the following output. Notice how much more useful the sorted output looks.

Unsorted:

```
{prescribed=1, a=1, time=2, any=1, no=1, shall=1, nor=1, peace=1,
owner=1, soldier=1, to=1, the=2, law=1, but=1, manner=1, without=1,
house=1, in=4, by=1, consent=1, war=1, quartered=1, be=2, of=3}
```

and sorted:

```
{a=1, any=1, be=2, but=1, by=1, consent=1, house=1, in=4, law=1,
manner=1, no=1, nor=1, of=3, owner=1, peace=1, prescribed=1,
quartered=1, shall=1, soldier=1, the=2, time=2, to=1, war=1,
without=1}
```

AbstractMap class

Similar to the other abstract collection implementations, the `AbstractMap` class overrides the `equals()` and `hashCode()` methods to ensure two equal maps return the same hash code. Two maps are equal if they are the same size, contain the same keys, and each key maps to the same value in both maps. By definition, the hash code for a map is the sum of the hash codes for the elements of the map, where each element is an implementation of the `Map.Entry` interface. Thus, no matter what the internal ordering of the maps, two equal maps will report the same hash code.

WeakHashMap class

A `WeakHashMap` is a special-purpose implementation of `Map` for storing only weak references to the keys. This allows for the key-value pairs of the map to be garbage collected when the key is no longer referenced outside of the `WeakHashMap`. Using `WeakHashMap` is beneficial for maintaining registry-like data structures, where the usefulness of an entry vanishes when its key is no longer reachable by any thread.

The Java 2 SDK, Standard Edition, version 1.3 adds a constructor to `WeakHashMap` that accepts a `Map`. With version 1.2 of the Java 2 platform, the available constructors permit only overriding the default load factor and initial capacity setting, not initializing the map from another map (as recommended by the `Map` interface documentation).

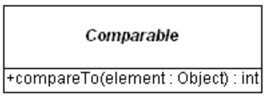
Sorting

There have been many changes to the core Java libraries to add support for sorting with the addition of the Collections Framework to the Java 2 SDK, version 1.2. Classes like `String` and `Integer` now implement the `Comparable` interface to provide a natural sorting order. For those classes without a natural order, or when you desire a different order than the natural order, you can implement the `Comparator` interface to define your own.

To take advantage of the sorting capabilities, the Collections Framework provides two interfaces that use it: `SortedSet` and `SortedMap`.

Comparable interface

The `Comparable` interface, in the `java.lang` package, is for when a class has a natural ordering. Given a collection of objects of the same type, the interface allows you to order the collection into that natural ordering.



The `compareTo()` method compares the current instance with an element passed in as an argument. If the current instance comes before the argument in the ordering, a negative value is returned. If the current instance comes after, then a positive value is returned. Otherwise, zero is returned. It is not a requirement that a zero return value signifies equality of elements. A zero return value just signifies that two objects are ordered at the same position.

There are fourteen classes in the Java 2 SDK, version 1.2 that implement the `Comparable` interface. The following table shows their natural ordering. While some classes share the same natural ordering, you can sort only classes that are *mutually comparable*.

Class	Ordering

BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short	Numerical
Character	Numerical by Unicode value
CollationKey	Locale-sensitive string ordering
Date	Chronological
File	Numerical by Unicode value of characters in fully-qualified, system-specific pathname
ObjectStreamField	Numerical by Unicode value of characters in name
String	Numerical by Unicode value of characters in string

The documentation for the `compareTo()` method of `String` defines the ordering lexicographically. This means the comparison is of the numerical values of the characters in the text, which is not necessarily alphabetically in all languages. For locale-specific ordering, use `Collator` with `CollationKey`.

The following demonstrates the use of `Collator` with `CollationKey` to do a locale-specific sorting:

```
import java.text.*;
 import java.util.*;

 public class CollatorTest {
   public static void main(String args[]) {
     Collator collator = Collator.getInstance();
     CollationKey key1 = collator.getCollationKey("Tom");
     CollationKey key2 = collator.getCollationKey("tom");
     CollationKey key3 = collator.getCollationKey("thom");
     CollationKey key4 = collator.getCollationKey("Thom");
     CollationKey key5 = collator.getCollationKey("Thomas");

     Set set = new TreeSet();
     set.add(key1);
     set.add(key2);
     set.add(key3);
     set.add(key4);
     set.add(key5);
     printCollection(set);
   }
   static private void printCollection(
       Collection collection) {
     boolean first = true;
     Iterator iterator = collection.iterator();
     System.out.print("[");
     while (iterator.hasNext()) {
       if (first) {
         first = false;
       } else {
         System.out.print(", ");
```

```
       }
       CollationKey key = (CollationKey)iterator.next();
       System.out.print(key.getSourceString());
     }
     System.out.println("]");
   }
 }
```

Running the program produces the following output:

```
[thom, Thom, Thomas, tom, Tom]
```

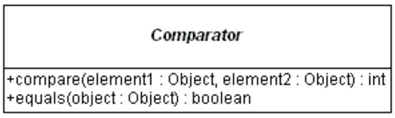
If the names were stored directly, without using `Collator`, then the lowercase names would appear apart from the uppercase names:

```
[Thom, Thomas, Tom, thom, tom]
```

Making your own class `Comparable` is just a matter of implementing the `compareTo()` method. It usually involves relying on the natural ordering of several data members. Your own classes should also override `equals()` and `hashCode()` to ensure two equal objects return the same hash code.

Comparator interface

When a class wasn't designed to implement `java.lang.Comparable`, you can provide your own `java.util.Comparator`. Providing your own `Comparator` also works if you don't like the default `Comparable` behavior.



The return values of the `compare()` method of `Comparator` are similar to the `compareTo()` method of `Comparable`. In this case, if the first element comes before the second element in the ordering, a negative value is returned. If the first element comes after, then a positive value is returned. Otherwise, zero is returned. Similar to `Comparable`, a zero return value does not signify equality of elements. A zero return value just signifies that two objects are ordered at the same position. It's up to the user of the `Comparator` to determine how to deal with it. If two unequal elements compare to zero, you should first be sure that is what you want and second document the behavior.

To demonstrate, you may find it easier to write a new `Comparator` that ignores case, instead of using `Collator` to do a locale-specific, case-insensitive comparison. The following is one such implementation:

```
class CaseInsensitiveComparator implements Comparator {
   public int compare(Object element1, Object element2) {
     String lowerE1 = ((String)element1).toLowerCase();
     String lowerE2 = ((String)element2).toLowerCase();
     return lowerE1.compareTo(lowerE2);
   }
 }
```

Because every class subclasses `Object` at some point, it is not a requirement that you implement the `equals()` method. In fact, in most cases you won't. Do keep in mind the `equals()` method checks for equality of `Comparator` implementations, not the objects being compared.

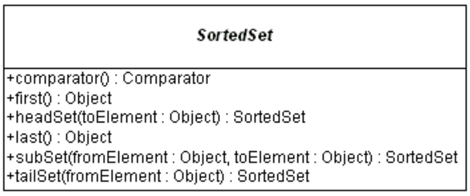
The `Collections` class has one predefined `Comparator` available for reuse. Calling `Collections.reverseOrder()` returns a `Comparator` that sorts objects that implement the `Comparable` interface in reverse order.

Exercise

* Exercise 4. How to use a map to count words on page 41

SortedSet interface

The Collections Framework provides a special `Set` interface for maintaining elements in a sorted order: `SortedSet`.



The interface provides access methods to the ends of the set as well as to subsets of the set. When working with subsets of the list, changes to the subset are reflected in the source set. In addition, changes to the source set are reflected in the subset. This works because subsets are identified by elements at the end points, not indices. In addition, if the `fromElement` is part of the source set, it is part of the subset. However, if the `toElement` is part of the source set, it is not part of the subset. If you would like a particular to-element to be in the subset, you must find the next element. In the case of a `String`, the next element is the same string with a null character appended (`string+"\0"`).;

The elements added to a `SortedSet` must either implement `Comparable` or you must provide a `Comparator` to the constructor of its implementation class: `TreeSet`. (You can implement the interface yourself. But the Collections Framework only provides one such concrete implementation class.)

To demonstrate, the following example uses the reverse order `Comparator` available from the `Collections` class:

```
Comparator comparator = Collections.reverseOrder();
 Set reverseSet = new TreeSet(comparator);
 reverseSet.add("Bernadine");
 reverseSet.add("Elizabeth");
 reverseSet.add("Gene");
 reverseSet.add("Elizabeth");
 reverseSet.add("Clara");
 System.out.println(reverseSet);
```

Running the program produces the following output:

```
[Gene, Elizabeth, Clara, Bernadine]
```

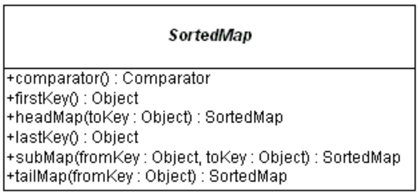
Because sets must hold unique items, if comparing two elements when adding an element results in a zero return value (from either the `compareTo()` method of `Comparable` or the `compare()` method of `Comparator`), then the new element is not added. If the elements are equal, then that is okay. However, if they are not, then you should modify the comparison method such that the comparison is compatible with `equals()`.

Using the prior `CaseInsensitiveComparator` to demonstrate this problem, the following creates a set with three elements: thom, Thomas, and Tom, not five elements as might be expected.

```
Comparator comparator = new CaseInsensitiveComparator();
 Set set = new TreeSet(comparator);
 set.add("Tom");
 set.add("tom");
 set.add("thom");
 set.add("Thom");
 set.add("Thomas");
```

SortedMap interface

The Collections Framework provides a special `Map` interface for maintaining keys in a sorted order: `SortedMap`.



The interface provides access methods to the ends of the map as well as to subsets of the map. Working with a `SortedMap` is just like a `SortedSet`, except the sort is done on the map keys. The implementation class provided by the Collections Framework is a `TreeMap`.

Because maps can only have one value for every key, if comparing two keys when adding a key-value pair results in a zero return value (from either the `compareTo()` method of `Comparable` or the `compare()` method of `Comparator`), then the value for the original key is replaced with the new value. If the elements are equal, then that is okay. However, if they are not, then you should modify the comparison method such that the comparison is compatible with `equals()`.

Section 4. Special collection implementations

Introduction

To keep the Collections Framework simple, added functionality is provided by wrapper implementations (also known as the Decorator design pattern -- see the *Design Patterns* book for more information on patterns). These wrappers delegate the collections part to the underlying implementation class, but they add functionality on top of the collection. These wrappers are all provided through the `Collections` class. The `Collections` class also provides support for creating special-case collections.

Read-only collections

After you've added all the necessary elements to a collection, it may be convenient to treat that collection as read-only, to prevent the accidental modification of the collection. To provide this capability, the `Collections` class provides six factory methods, one for each of `Collection`, `List`, `Map`, `Set`, `SortedMap`, and `SortedSet`.

* `Collection unmodifiableCollection(Collection collection)`
* `List unmodifiableList(List list)`
* `Map unmodifiableMap(Map map)`
* `Set unmodifiableSet(Set set)`
* `SortedMap unmodifiableSortedMap(SortedMap map)`
* `SortedSet unmodifiableSortedSet(SortedSet set)`

Once you've filled the collection, replace the original reference with the read-only reference. If you don't replace the original reference, then the collection is not read-only, as you can still use the original reference to modify the collection. The following program demonstrates the proper way to make a collection read-only. In addition, it shows what happens when you try to modify a read-only collection.

```
import java.util.*;

 public class ReadOnlyExample {
   public static void main(String args[]) {
     Set set = new HashSet();
     set.add("Bernadine");
     set.add("Elizabeth");
     set.add("Gene");
     set.add("Elizabeth");
     set = Collections.unmodifiableSet(set);
     set.add("Clara");
   }
 }
```

When the program is run and the last `add()` operation is attempted on the read-only set, an `UnsupportedOperationException` is thrown.

Thread-safe collections

The key difference between the historical collection classes and the new implementations within the Collections Framework is the new classes are *not* thread-safe. The designers took this approach to allow you to use synchronization only when you need it, making everything work much faster. If, however, you are using a collection in a multi-threaded environment, where multiple threads can modify the collection simultaneously, the modifications need to be synchronized. The `Collections` class provides for the ability to wrap existing collections into synchronized ones with another set of six methods:

* `Collection synchronizedCollection(Collection collection)`
* `List synchronizedList(List list)`
* `Map synchronizedMap(Map map)`
* `Set synchronizedSet(Set set)`
* `SortedMap synchronizedSortedMap(SortedMap map)`
* `SortedSet synchronizedSortedSet(SortedSet set)`

Synchronize the collection immediately after creating it. You also must not retain a reference to the original collection, or else you can access the collection unsynchronized. The simplest way to make sure you don't retain a reference is never to create one:

```
Set set = Collection.synchronizedSet(new HashSet());
```

Making a collection unmodifiable also makes a collection thread-safe, as the collection can't be modified. This avoids the synchronization overhead.

Singleton collections

The `Collections` class provides for the ability to create single element sets fairly easily. Instead of having to create the `Set` and fill it in in separate steps, you can do it all at once. The resulting `Set` is immutable.

```
Set set = Collection.singleton("Hello");
```

The Java 2 SDK, Standard Edition, version 1.3 adds the ability to create singleton lists and maps, too:

* `List singletonList(Object element)`
* `Map singletonMap(Object key, Object value)`

Multiple copy collections

If you need an immutable list with multiple copies of the same element, the `nCopies(int n, Object element)` method of the `Collections` class returns just such the `List`:

```
List fullOfNullList = Collection.nCopies(10, null);
```

By itself, that doesn't seem too useful. However, you can then make the list modifiable by passing it along to another list:

```
List anotherList = new ArrayList(fullOfNullList);
```

This now creates a 10-element `ArrayList`, where each element is `null`. You can now modify each element at will, as it becomes appropriate.

Empty collections

The `Collections` class also provides constants for empty collections:

* `List EMPTY_LIST`
* `Set EMPTY_SET`

The Java 2 SDK, Standard Edition, version 1.3 has a predefined empty map constant:

* `Map EMPTY_MAP`

Section 5. Historical collection classes

Introduction

While this tutorial is about the new Collections Framework of the Java 2 SDK, there are times when you still need to use some of the original collections capabilities. This section reviews some of the capabilities of working with arrays, vectors, hashtables, enumerations, and other historical capabilities.

Arrays

One learns about arrays fairly early on when learning the Java programming language. Arrays are defined to be fixed-size collections of the same datatype. They are the only collection that supports storing primitive datatypes. Everything else, including arrays, can store objects. When creating an array, you specify both the number and type of object you wish to store. And, over the life of the array, it can neither grow nor store a different type (unless it extends the first type).

To find out the size of an array, you ask its single public instance variable, `length`, as in `array.length`.

To access a specific element, either for setting or getting, you place the integer argument within square brackets (`[int]`), either before *or* after the array reference variable. The integer index is zero-based, and accessing beyond either end of the array will throw an `ArrayIndexOutOfBoundsException` at run time. If, however, you use a `long` variable to access an array index, you'll get a compiler-time error.

Arrays are full-fledged subclasses of `java.lang.Object`. They can be used with the various Java constructs except for an object:

```
Object obj = new int[5];
 if (obj instanceof int[]) {
   // true
 }
 if (obj.getClass().isArray()) {
   // true
 }
```

When created, arrays are automatically initialized, either to `false` for a `boolean` array , `null` for an `Object` array, or the numerical equivalent of 0 for everything else.

To make a copy of an array, perhaps to make it larger, you use the `arraycopy()` method of `System`. You need to preallocate the space in the destination array.

```
System.arraycopy(Object sourceArray, int
         sourceStartPosition, Object destinationArray, int
         destinationStartPosition, int length)
```

Vector and Stack classes

A `Vector` is a historical collection class that acts like a growable array, but can store heterogeneous data elements. With the Java 2 SDK, version 2, the `Vector` class has been retrofitted into the Collections Framework hierarchy to implement the `List` interface. However, if you are using the new framework, you should use `ArrayList`, instead.

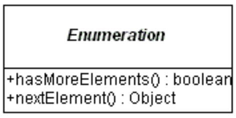
When transitioning from `Vector` to `ArrayList`, one key difference is that the arguments have been reversed to positionally change an element's value:

* From original `Vector` class `void setElementAt(Object element, int index)`
* From `List` interface `void set(int index, Object element)`

The `Stack` class extends `Vector` to implement a standard last-in-first-out (LIFO) stack, with `push()` and `pop()` methods. Be careful, though. Because the `Stack` class extends the `Vector` class, you can still access or modify a `Stack` with the inherited `Vector` methods.

Enumeration interface

The `Enumeration` interface allows you to iterate through all the elements of a collection. In the Collections Framework, this interface has been superceded by the `Iterator` interface. However, not all libraries support the newer interface, so you may find yourself using `Enumeration` from time to time.



Iterating through an `Enumeration` is similar to iterating through an `Iterator`, though some people like the method names better with `Iterator`. However, there is no removal support with `Enumeration`.

```
Enumeration enum = ...;
 while (enum.hasNextElement()) {
   Object element = iterator.nextElement();
   // process element
 }
```

Dictionary, Hashtable, and Properties classes

The `Dictionary` class is completely full of abstract methods. In other words, it should have been an interface. It forms the basis for key-value pair collections in the historical collection classes, with its replacement being `Map` in the new framework. `Hashtable` and `Properties` are the two specific implementations of `Dictionary` available.

The `Hashtable` implementation is a generic dictionary that permits storing any object as its key or value (besides `null`). With the Java 2 SDK, version 1.2, the class has been reworked into the Collections Framework to implement the `Map` interface. So you can use the original `Hashtable` methods or the newer `Map` methods. If you need a synchronized `Map`, using `Hashtable` is slightly faster than using a synchronized `HashMap`.

The `Properties` implementation is a specialized `Hashtable` for working with text strings. While you have to cast values retrieved from a `Hashtable` to your desired class, the `Properties` class allows you to get text values without casting. The class also supports loading and saving property settings from an input stream or to an output stream. The most commonly used set of properties is the system properties list, retrieved by `System.getProperties()`.

BitSet class

A `BitSet` represents an alternate representation of a set. Given a finite number of n objects, you can associate a unique integer with each object. Then each possible subset of the objects corresponds to an n-bit vector, with each bit "on" or "off" depending on whether the object is in the subset. For small values of n, a bit vector might be an extremely compact representation. However, for large values of n, an actual bit vector might be inefficient, when most of the bits are off.

There is no replacement to `BitSet` in the new framework.

Exercise

* Exercise 1. How to use a HashSet for a sparse bit set on page 36

Section 6. Algorithm support

Introduction

The `Collections` and `Arrays` classes, available as part of the Collections Framework, provide support for various algorithms with the collection classes, both new and old. The different operations, starting with sorting and searching, are described next.

Sorting arrays

While the `TreeSet` and `TreeMap` classes offer sorted version of sets and maps, there is no sorted `List` collection implementation. Also, prior to the collections framework, there was no built-in support for sorting arrays. As part of the framework, you get both support for sorting a `List`, as well as support for sorting arrays of anything, including primitives. With any kind of sorting, all items must be comparable to each other (*mutually comparable*). If they are not, a `ClassCastException` will be thrown.

Sorting of a `List` is done with one of two `sort()` methods in the `Collections` class. If the element type implements `Comparable` then you would use the `sort(List list)` version. Otherwise, you would need to provide a `Comparator` and use `sort(List list, Comparator comparator)`. Both versions are destructive to the `List` and guarantee `O(n log 2 n)` performance (or better), including when sorting a `LinkedList`, using a merge sort variation.

Sorting of arrays is done with one of eighteen different methods. There are two methods for sorting each of the seven primitive types (except `boolean`), one for sorting the whole array and one for sorting part of the array. The remaining four are for sorting object arrays `Object[ ]`).

To sort primitive arrays, simply call `Arrays.sort()` with your array as the argument and let the compiler determine which of the following methods to pick:

```
*   void sort(byte array[ ])
*   void sort(byte array[ ], int fromIndex, int toIndex)
*   void sort(short array[ ])
*   void sort(short array[ ], int fromIndex, int toIndex)
*   void sort(int array[ ])
*   void sort(int array[ ], int fromIndex, int toIndex)
*   void sort(long array[ ])
*   void sort(long array[ ], int fromIndex, int toIndex)
*   void sort(float array[ ])
*   void sort(float array[ ], int fromIndex, int toIndex)
*   void sort(double array[ ])
*   void sort(double array[ ], int fromIndex, int toIndex)
*   void sort(char array[ ])
*   void sort(char array[ ], int fromIndex, int toIndex)
```

The sorting of object arrays is a little more involved, as the compiler doesn't check everything for you. If the object in the array implements `Comparable`, then you can just sort the array directly, in whole or in part. Otherwise, you must provide a `Comparator` to do the sorting for you. You can also provide a `Comparator` implementation if you don't like the default ordering.

* void sort(Object array[ ])
* void sort(Object array[ ], int fromIndex, int toIndex)
* void sort(Object array[ ], Comparator comparator)
* void sort(Object array[ ], int fromIndex, int toIndex, Comparator comparator)

Searching

Besides sorting, the `Collections` and `Arrays` classes provide mechanisms to search a `List` or array, as well as to find the minimum and maximum values within a `Collection`.

While you can use the `contains()` method of `List` to find if an element is part of the list, it assumes an unsorted list. If you've previously sorted the `List`, using `Collections.sort()`, then you can do a much quicker binary search using one of the two overridden `binarySearch()` methods. If the objects in the `List` implement `Comparable`, then you don't need to provide a `Comparator`. Otherwise, you must provide a `Comparator`. In addition, if you sorted with a `Comparator`, you must use the same `Comparator` when binary searching.

* public static int binarySearch(List list, Object key)
* public static int binarySearch(List list, Object key, Comparator comparator)

If the `List` to search subclasses the `AbstractSequentialList` class (like `LinkedList`), then a sequential search is actually done.

Array searching works the same way. After using one of the `Arrays.sort()` methods, you can take the resulting array and search for an element. There are seven overridden varieties of `binarySearch()` to search for a primitive (all but `boolean`), and two to search an `Object` array, both with and without a `Comparator`.

If the original `List` or array is unsorted, the result is non-deterministic.

Besides searching for specific elements within a `List`, you can search for extreme elements within any `Collection`: the minimum and maximum. If you know your collection is already sorted, just get the first or last element. However, for unsorted collections, you can use one of the `min()` or `max()` methods of `Collections`. If the object in the collection doesn't implement `Comparable`, then you must provide a `Comparator`.

* Object max(Collection collection)
* Object max(Collection collection, Comparator comparator)
* Object min(Collection collection)
* Object min(Collection collection, Comparator comparator)

Checking equality

While the `MessageDigest` class always provided an `isEqual()` method to compare two `byte` arrays, it never felt right to use it to compare `byte` arrays unless they were from message digests. Now, with the help of the `Arrays` class, you can check for equality of any array of primitive or object type. Two arrays are equal if they contain the same elements in the same order. Checking for equality with arrays of objects relies on the `equals()` method of each object to check for equality.

```
byte array1[] = ...;
 byte array2[] = ...;
 if (Arrays.equals(array1, array2) {
   // They're equal
 }
```

Manipulating elements

The `Collections` and `Arrays` classes offer several ways of manipulating the elements within a `List` or array. There are no additional ways to manipulate the other key framework interfaces (`Set` and `Map`).

With a `List`, the `Collections` class lets you replace every element with a single element, copy an entire list to another, reverse all the elements, or shuffle them around. When copying from one list to another, if the destination list is larger, the remaining elements are untouched.

* `void fill(List list, Object element)`
* `void copy(List source, List destination)`
* `void reverse(List list)`
* `void shuffle(List list)`
* `void shuffle(List list, Random random)`

The `Arrays` class allows you to replace an entire array or part of an array with one element via eighteen overridden versions of the `fill()` method. All the methods are of the form `fill(array, element)` or `fill(array, fromIndex, toIndex, element)`.

Big-O notation

Performance of sorting and searching operations with collections of size *n* is measured using Big-O notation. The notation describes the complexity of the algorithm in relation to the maximum time in which an algorithm operates, for large values of *n*. For instance, if you iterate through an entire collection to find an element, the Big-O notation is referred to as `O(n)`, meaning that as *n* increases, time to find an element in a collection of size *n* increases linearly. This demonstrates that Big-O notation assumes worst case performance. It is always possible that performance is quicker.

The following table shows the Big-O values for different operations, with 65,536 as the value for *n*. In addition, the operation count shows that if you are going to perform multiple search operations on a collection, it is faster to do a quick sort on the collection, prior to searching, versus doing a linear search each time. (And, one should avoid bubble sorting, unless *n* is really small!)

Description	**Big-O**	**# Operations**	**Example**

Constant	O(1)	1	Hash table lookup (ideal)
Logarithmic	O(log base 2 of n)	16	Binary search on sorted collection
Linear	O(n)	65,536	Linear search
Linear-logarithmic	O(n x log base 2 of n)	1,048,576	Quick sort
Quadratic	O(n x n)	4,294,967,296	Bubble sort

Legend: n = 65536

Section 7. Usage issues

Introduction

The Collections Framework was designed such that the new framework classes and the historical data structure classes can interoperate. While it is good if you can have all your new code use the new framework, sometimes you can't. The framework provides much support for intermixing the two sets of collections. In addition, you can develop with a subset of the capabilities with JDK 1.1.

Converting from historical collections to new collections

There are convenience methods for converting from many of the original collection classes and interfaces to the newer framework classes and interfaces. They serve as bridges when you need a new collection but have a historical collection. You can go from an array or `Vector` to a `List`, a `Hashtable` to a `Map`, or an `Enumeration` to any `Collection`.

For going from any array to a `List`, you use the `asList(Object array[])` method of the `Arrays` class. Changes to the `List` pass through to the array, but you cannot adjust the size of the array.

```
String names[] = {"Bernadine",
   "Elizabeth", "Gene", "Clara"};
 List list = Arrays.asList(names);
```

Because the original `Vector` and `Hashtable` classes have been retrofitted into the new framework, as a `List` and `Map` respectively, there is no work to treat either of these historical collections as part of the new framework. Treating a `Vector` as a `List` automatically carries to its subclass `Stack`. Treating a `Hashtable` as a `Map` automatically carries to its subclass `Properties`.

Moving an `Enumeration` to something in the new framework requires a little more work, as nothing takes an `Enumeration` in its constructor. So, to convert an `Enumeration`, you create some implementation class in the new framework and add each element of the enumeration.

```
Enumeration enumeration = ...;
 Collection collection = new LinkedList();
 while (e.hasMoreElements()) {
   collection.add(e.nextElement());
 }
 // Operate on collection
```

Converting from new collections to historical collections

In addition to supporting the use of the old collection classes within the new Collections Framework, there is also support for using the new framework and still using libraries that only support the original collections. You can easily convert from `Collection` to array, `Vector`, or `Enumeration`, as well as from `Map` to `Hashtable`.

There are two ways to go from `Collection` to array, depending upon the type of array you need. The simplest way involves going to an `Object` array. In addition, you can also convert the collection into any other array of objects. However, you cannot directly convert the collection into an array of primitives, as collections must hold objects.

To go from a collection to an `Object[]`, you use the `toArray()` method of `Collection`:

```
Collection collection = ...;
 Object array[] = collection.toArray();
```

The `toArray()` method is overridden to accept an array for placing the elements of the collection: `toArray(Object array[])`. The datatype of the argument determines the type of array used to store the collection and returned by the method. If the array isn't large enough, a new array of the appropriate type will be created.

```
Collection collection = ...;
 int size = collection.size();
 Integer array[] = collection.toArray(new Integer[size]);
```

To go from `Collection` to `Vector`, the `Vector` class now includes a constructor that accepts a `Collection`. As with all these conversions, if the element in the original conversion is mutable, then no matter from where it is retrieved and modified, it's changed everywhere.

```
Dimension dims[] = {new Dimension (0,0),
   new Dimension (0,0)};
 List list = Arrays.asList(dims);
 Vector v = new Vector(list);
 Dimension d = (Dimension)v.get(1);
 d.width = 12;
```

Going from `Collection` to `Enumeration` is much easier than going from `Enumeration` to `Collection`. The `Collections` class includes a static method to do the conversion for you:

```
Collection collection = ...;
 Enumeration enum = Collections.enumeration(collection);
```

The conversion from `Map` to `Hashtable` is similar to the conversion from `Collection` to `Vector`: just pass the new framework class to the constructor. After the conversion, changing the value for the key in one does not alter the value for the key in the other.

```
Map map = ...;
 Hashtable hashtable = new Hashtable(map);
```

Working with the Collections Framework support in JDK 1.1

If you are still using JDK 1.1, you can start taking advantage of the Collections Framework today. Sun Microsystems provides a *subset of the collections API* for use with JDK 1.1. The interfaces and classes of the framework have been moved from the `java.lang` and `java.util` package to the non-core `com.sun.java.util.collections` package. This is not a complete set of classes changed to support the framework, but only copies of those introduced. Basically, that means that none of the system classes are sortable by default; you must provide your own `Comparator`.

The following table lists the classes available in the Collections Framework release for JDK 1.1. In some cases, there will be two different implementations of the same class, like with `Vector`, as the 1.2 framework version implements `List` and the core 1.1 version doesn't.

AbstractCollection	AbstractList
AbstractMap	AbstractSequentialList
AbstractSet	ArrayList
Arrays	Collection
Collections	Comparable
Comparator	ConcurrentModificationException
HashMap	HashSet
Hashtable	Iterator
LinkedList	List
ListIterator	Map
NoSuchElementException	Random
Set	SortedMap
SortedSet	TreeMap
TreeSet	UnsupportedOperationException
Vector	

Section 8. Alternative collections

Introduction

Because the Collections Framework was not available prior to the introduction of the Java 2 platform, several alternative collection libraries became available. Two such libraries are Doug Lea's Collections Package and ObjectSpace's JGL.

Doug Lea's collections package

The *collections package* from Doug Lea (author of *"Concurrent Programming in Java"*) was first available in October 1995 and last updated in April 1997. It probably offered the first publicly available collections library. While no longer supported, the library shows the complexity added to the class hierarchy when you try to provide updateable and immutable collections, without optional methods in interfaces or wrapper implementations. While a good alternative at the time, its use is no longer recommended. (Doug also helped author some of the Collections Framework.)

ObjectSpace's JGL

In addition to Doug Lea's collections library, the *Generic Collection Library for Java* (JGL) from ObjectSpace was an early collection library available for the Java platform. (If you are curious of how the library name maps to the acronym, it doesn't. The name of the first version of the library infringed on Sun's Java trademark. ObjectSpace changed the name, but the original acronym stuck.) Following the design patterns of the Standard Template Library (STL) for C++, the library provides algorithmic support, in addition to a data structure library. While the JGL is a good alternative collection framework, it didn't meet the *design goals* of the Collections Framework team: "The main design goal was to produce an API that was reasonably small, both in size and, more importantly, in *conceptual weight*." With that in mind, the team came up with the Collections Framework.

While not adopted by Sun Microsystems, the JGL has been included with many IDE tools. Due to its early availability, the JGL is available to well over 100,000 developers.

For a comparison of JGL versus the Collections Framework, see *The battle of the container frameworks: which should you use?* article in JavaWorld.

Section 9. Exercises

About the exercises

These exercises are designed to provide help according to your needs. For example, you might simply complete the exercise given the information and the task list in the exercise body; you might want a few hints; or you may want a step-by-step guide to successfully complete a particular exercise. You can use as much or as little help as you need per exercise. Moreover, because complete solutions are also provided, you can skip a few exercises and still be able to complete future exercises requiring the skipped ones.

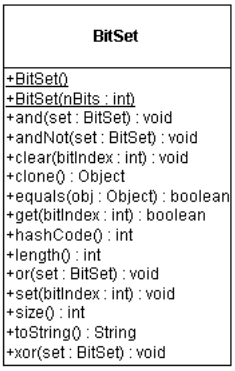
Each exercise has a list of any prerequisite exercises, a list of skeleton code for you to start with, links to necessary API pages, and a text description of the exercise goal. In addition, there is help for each task and a solutions page with links to files that comprise a solution to the exercise.

Exercise 1. How to use a HashSet for a sparse bit set

A *sparse* bitset is a large collection of `boolean` values where many of the values are off (or `false`). For maintaining these sparsely populated sets, the `BitSet` class can be very inefficient. Because the majority of the bits will be off, space will be occupied to store nothing. For working with these sparse bitsets, you can create an alternate representation, backed instead by a hashtable, or `HashMap`. Only those positions where a value is set are then stored in the mapping.

To create a sparse bitset, subclass `BitSet` and override the necessary methods (everything). The skeleton code should help you get started, so you can focus on the set-oriented routines.

The following UML diagram shows you the *`BitSet`* operations:



For more information on the `BitSet` class, see BitSet class on page 27 .

Skeleton Code

* *SparseBitSet.java*
* *Tester.java*

Task 1: Either start with the *skeleton code* or create a `SparseBitSet` class. The skeleton provides a no-argument constructor only. Because the bitmap will be sparse, you shouldn't provide a constructor that will preallocate any space, as `BitMap` does. Besides a constructor, the skeleton defines the `clear()`, `clone()`, `equals()`, `get()`, `hashCode()`, `set()`, `size()`, and `toSting()` method.

In the skeleton, the `getBitSet()` method returns the internal `Set` used to store the bits. You should use this method as you complete the other methods in the subclass. The actual `HashSet` used to store the bit values is created for you in the constructor.

Help for task 1: Shift click to save the file to your working directory.

Task 2: Working alphabetically, the first method to complete is the *`and(BitSet set)`* method. This method performs a logical AND of the two bit sets. Only those bits in both sets are in the resulting set. Complete the `and()` method to combine the internal `Set` with that of the argument.

Help for task 2: The *`retainAll()`* method of *`Set()`* retains only the elements in this set that are contained in the other set.

Task 3: The next method to complete is the *`andNot(BitSet set)`* method. Instead of keeping bits present in both, the `andNot()` operation will remove bits from the current set that are also in the set passed as an argument. This is sometimes called a logical NAND operation.

Help for task 3: The *`removeAll()`* method of *`Set()`* removes the elements in this set that are contained in the other set.

Task 4: Because the `clear()`, `clone()`, `equals()`, `get()`, and `hashCode()` methods are defined in the skeleton code, the next method to complete is the *`length()`* method. The `length()` method returns the *logical* size of the `BitSet`, which is defined to be the position of the highest set bit, plus one. Thus, if bit 127 was *set*, the length would be 128 as the bit counting starts at zero.

Help for task 4: The *`max()`* method of *`Collections()`* reports the highest value in a collection. Make sure you check for an empty set, as an empty set reports zero, not one.

Task 5: The last easy method to complete is the *`or(BitSet set)`* method. This method performs a logical OR operation of the two bit sets. Every bit set of either set is in the resulting set.

Help for task 5: The *`addAll()`* method of *`Set()`* combines the elements of two sets.

Task 6: With the `set()`, `size()`, and `toString()` methods already defined for you, you're left to complete the *`xor(BitSet set)`* method. This method performs a logical exclusive or (XOR) operation. Only those bits on in one of the sets will be on in the resulting set.

Unlike the other operations, the solution is not just a single method call of `Set`.

Help for task 6: You need to find out what elements are in each set that are not in the other set without altering the original sets. Once you have these two sets, combine them to create the resulting set.

Task 7: Compile your program and run the `Tester` program to see what happens. The `Tester` program creates a couple of sets and performs all the operations.

Help for task 7: Check your output to make sure the various set operations are correct.

Exercise 1. How to use a HashSet for a Sparse Bit Set: Solution

The following Java source files represent a solution to this exercise.

* *Solution/SparseBitSet.java*
* *Solution/Tester.java*

Exercise 2. How to use a TreeSet to provide a sorted JList

By default, the *`JList`* component displays its element list unsorted. With the help of the *`TreeSet`* , you can make it sorted by providing your own implementation of the *`ListModel`* interface for storing the data.

This exercise has you create just such an implementation.

If you aren't familiar with the Swing component set, don't worry. The `Tester` program includes all the necessary code to create the user interface. You are only responsible for finishing up the data model implementation and adding some action behind some of the buttons in the user interface.

Skeleton Code

* *SortedListModel.java*
* *Tester.java*

Task 1: Either start with the *skeleton code* or create a `SortedListModel` class. The class extends *`AbstractListModel`* .

Help for task 1: Shift click to save the file to your working directory.

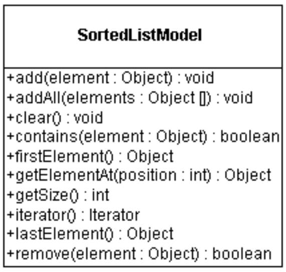
Task 2: Create an instance variable of type *`SortedSet`* . Then, in the constructor create an instance of type *`TreeSet`* and assign it to the variable.

Task 3: At a minimum, the *`AbstractListModel`* class requires the *`getSize()`* and *`getElementAt()`* methods of the *`ListModel`* interface to be defined. Complete the stubs such that they get the size and element from the set saved in the prior step.

Help for task 3: Use the `size()` method of *`Set`* to complete `getSize()`.

Either iterate through the set to the appropriate position, or convert the set to an array using the `toArray()` method of *`Set`* to complete `getElementAt()`.

Task 4: Besides implementing the methods of *ListModel* , the SortedListModel class provides several methods to access and alter the data model. Many of the methods are already completed. The following UML diagram shows the complete set of operations.



Help for task 4: If you are using the skeleton code, there is no task to perform here.

Task 5: Two methods in the SortedListModel skeleton are left to complete: firstElement() and lastElement(). These require the use of methods specific to the *SortedSet* interface to complete.

Help for task 5: Use the first() method of *SortedSet* to find the first element.

Use the last() method of *SortedSet* to find the last element.

Task 6: In the Tester skeleton, the printAction() method needs to be completed. As the name may imply, its purpose is to display a list of the elements in the JList. Use an *Iterator* to display the elements in its data model. The data model is stored in the model variable, which is of type SortedListModel.

Help for task 6: Use the iterator() method of SortedListModel to get an Iterator.

Task 7: Compile your program and run the Tester program to see what happens. You can provide several values as command line arguments to initialize the contents. Try out several buttons on the user interface to make sure the SortedListModel works.

Help for task 7:

```
java Tester One Two Three Four Five Six Seven Eight
    Nine Ten
```

Make sure the elements in the JList are sorted.

Exercise 2. How to use a TreeSet to provide a sorted JList: Solution

The following Java source files represent a solution to this exercise.

* *Solution/SortedListModel.java*
* *Solution/Tester.java*

Exercise 3. How to use an ArrayList with a JComboBox

If you've ever looked at the data model class for the `JComboBox` component of the JFC/Swing component set, you may have noticed that the data model is backed by a `Vector` . If, however, you don't need the synchronized access of a `Vector` (thus increasing performance) or you prefer the new Collections Framework, you can create your own implementation of the `ComboBoxModel` or `MutableComboBoxModel` interface for storing the data in a `List` or more specifically in a `ArrayList` .

This exercise has you create just such an implementation.

If you aren't familiar with the Swing component set, don't worry. The `Tester` program includes all the necessary code to create the user interface. You are only responsible for finishing up the data model implementation.

Skeleton Code

* *ArrayListComboBoxModel.java*
* *Tester.java*

Task 1: Either start with the *skeleton code* or create an `ArrayListComboBoxModel` class. The class extends `AbstractListModel` and implements `MutableComboBoxModel` .

Help for task 1: Shift click to save the file to your working directory.

Task 2: Create a variable of type `List`. Refer the `List` argument in the constructor to your variable.

Task 3: The `AbstractListModel` class leaves the `getSize()` and `getElementAt()` methods of the `ListModel` interface to be defined. Complete the stubs such that they get the size and element from the list saved in the prior step.

Help for task 3: Use the `size()` method of `List` to complete `getSize()`.

Use the `get(int position)` method of `List` to complete `getElementAt()`.

Task 4: By stating that the `ArrayListComboBoxModel` class implements the `MutableComboBoxModel` interface, you are saying you'll provide implementations for the methods of both the `MutableComboBoxModel` and `ComboBoxModel` interfaces, as the former extends the latter. The `getSelectedItem()` and `setSelectedItem()` methods of the `ComboBoxModel` interface are already defined for you.

Task 5: The `MutableComboBoxModel` interface, defines four methods: `addElement(Object element)` , `insertElementAt(Object element, int position)` , `removeElement(Object element)` , and `removeElementAt(int position)` . Complete the stubs such that they alter the list saved in a prior step.

Help for task 5: Use the `add(Object element)` method of `List` to insert an element at the end.

Use the `add(Object element, int position)` method of `List` to insert an element at a designated position.

Use the `remove(Object element)` method of *`List`* to remove the first instance of an element.

Use the `remove(int position)` method of *`List`* to remove an element at a designated position.

Task 6: Compile your program and run the `Tester` program to see what happens. Provide several names as command line arguments. The `Tester` program tests your new data model class by adding and removing elements from the model.

Help for task 6:

```
java Tester Jim Joe Mary
```

Check your output to make sure that Jim, Joe, and Mary are added to the names you provided.

Exercise 3. How to use an ArrayList with a JComboBox: Solution

The following Java source files represent a solution to this exercise.

* *Solution/ArrayListComboBoxModel.java*
* *Solution/Tester.java*

Exercise 4. How to use a map to count words

This program enhances the program from Map interface on page 14 to read from a URL, instead of just counting words from the command line.

If you aren't familiar with the Swing component set, don't worry. The `Tester` program includes all the necessary code to create the user interface. You are only responsible for counting the words and formatting the output. Even the source code to read from the URL is provided.

Skeleton Code

* *CaseInsensitiveComparator.java*
* *Tester.java*
* *WordCount.java*

Task 1: Either start with the *skeleton code* or create a `WordCount` class.

Help for task 1: Shift click to save the file to your working directory.

If you don't start from the skeleton code, you'll have to read the URL yourself and parse the contents with a *`StringTokenizer`* .

Task 2: Create an instance variable of type *Map* and assign it to a new *HashMap* . In the getMap() method return the map created. In the clear() method, empty out the defined map.

Help for task 2: Use the *clear()* method of Map to empty it out.

Task 3: Complete the addWords() method to count each word returned by the *StringTokenizer* . The program already separates each line in the URL into individual words.

Help for task 3: The value for the key (word) is the current frequency. If the word is not found, then it is not in the map yet and should be added with a value of one. Otherwise, add one to the existing frequency.

Refer back to the map usage example in Map interface on page 14 .

Feel free to try a different set of delimiters with the StringTokenizer.

Task 4: The Tester program has a JTextArea to display the results of the counting. The program displays the String returned by the private convertMap() method in the JTextArea. It is your job to format the output nicely, as the *toString()* of *AbstractMap* displays everything on one line. Start off with the *skeleton code* for CaseInsensitiveComparator so you can sort the output in a case-insensitive manner. The implementation will be identical to the comparator interface described in Sorting on page 17 .

Help for task 4: Shift click to save the file to your working directory.

Either implement *compare()* yourself, or copy it from the course notes.

Task 5: Now that you have a case-insensitive *Comparator* , use it to create a *TreeMap* full of the original map contents. That way, the output can be displayed sorted.

Help for task 5: Getting the original *Map* sorted with the new Comparator is a two-step process. In the TreeMap constructor, specify the Comparator. Then, put all the original map entries in the TreeMap with the *putAll()* method.

Task 6: After getting an *Iterator* of all the keys, display one key-value pair per line, using the predefined PrintWriter to format the output. It is backed by a StringBuffer and will be automatically returned.

Help for task 6: First get the entry set with the *entrySet()* method.

Then, get its Iterator. Each element is a *Map.Entry*

Task 7: Compile your program and run the Tester program to see what happens. You specify the URL to read in the JTextField. When you press Enter, the URL is read and the words are added to the map. Once done reading, the JTextArea is updated. If you want to clear out the map, press the Clear button.

Exercise 4. How to use a map to count words: Solution

The following Java source files represent a solution to this exercise.

* *Solution/CaseInsensitiveComparator.java*
* *Solution/Tester.java*
* *Solution/WordCount.java*

Section 10. Wrapup

In summary

The Collections Framework provides a well-designed set of interfaces, implementations, and algorithms for representing and manipulating groups of elements. Understanding all the capabilities of this framework reduces the effort required to design and implement a comparable set of APIs, as was necessary prior to their introduction. Now that you have completed this tutorial, you can effectively manage groups of data elements.

Further reading and references

The following resources should help in your usage and understanding of the Collections Framework:

* *Java language essentials* tutorial on developerWorks
* *"Porting C++ to Java"* on developerWorks, a step-by-step approach to porting C++ to Java effectively
* *"How to Build Data Structures in Java"* , a JDC article from prior to the existence of the Collections Framework
* *"Design Patterns"* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (*The Gang of Four*)
* *Collections Framework Support for JDK 1.1*
* *Doug Lea's Collections Package*
* *Generic Collection Library for Java* , JGL from ObjectSpace
* *"The battle of the container frameworks: which should you use?"* , JavaWorld article from January 1999
* *Sun's Collections Framework home page*

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the author John Zukowski (*jaz@jguru.com*)

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.