



Collections Framework Overview

Introduction

The Java 2 platform includes a new *collections framework*. A *collection* is an object that represents a group of objects (such as the familiar [Vector](#) class). A collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.

The primary advantages of a collections framework are that it:

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

The collections framework consists of:

- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.
- **Legacy Implementations** - The collection classes from earlier releases, `Vector` and `Hashtable`, have been retrofitted to implement the collection interfaces.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.
- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.

- **Infrastructure** - Interfaces that provide essential support for the collection interfaces.
 - **Array Utilities** - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.
-

Collection Interfaces

There are six *collection interfaces*. The most basic interface is `Collection`. Three interfaces extend `Collection`: `Set`, `List`, and `SortedSet`. The other two collection interfaces, `Map` and `SortedMap`, do not extend `Collection`, as they represent mappings rather than true collections. However, these interfaces contain *collection-view* operations, which allow them to be manipulated as collections.

All of the modification methods in the collection interfaces are labeled *optional*. Some implementations may not perform one or more of these operations, throwing a runtime exception (`UnsupportedOperationException`) if they are attempted. Implementations must specify in their documentation which optional operations they support. Several terms are introduced to aid in this specification:

- Collections that do not support any modification operations (such as `add`, `remove` and `clear`) are referred to as *unmodifiable*. Collections that are not unmodifiable are referred to *modifiable*.
- Collections that additionally guarantee that no change in the `Collection` object will ever be visible are referred to as *immutable*. Collections that are not immutable are referred to as *mutable*.
- Lists that guarantee that their size remains constant even though the elements may change are referred to as *fixed-size*. Lists that are not fixed-size are referred to as *variable-size*.
- Lists that support fast (generally constant time) indexed element access are known as *random access* lists. Lists that do not support fast indexed element access are known as *sequential access* lists. The [RandomAccess](#) marker interface is provided to allow lists to advertise the fact that they support random access. This allows generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

Some implementations may restrict what elements (or in the case of `Maps`, keys and values) may be stored. Possible restrictions include requiring elements to:

- Be of a particular type.
- Be non-null.
- Obey some arbitrary predicate.

Attempting to add an element that violates an implementation's restrictions results in a runtime exception, typically a `ClassCastException`, an `IllegalArgumentException` or a `NullPointerException`. Attempting to remove or test for the presence of an element that

violates an implementation's restrictions may result in an exception, though some "restricted collections" may permit this usage.

Collection Implementations

Class that implement the collection interfaces typically have names of the form *<Implementation-style><Interface>*. The general purpose implementations are summarized in the table below:

Implementations						
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

The general-purpose implementations support all of the *optional operations* in the collection interfaces, and have no restrictions on the elements they may contain. They are unsynchronized, but the `Collections` class contains static factories called [synchronization wrappers](#) that may be used to add synchronization to any unsynchronized collection. All of the new implementations have *fail-fast iterators*, which detect illegal concurrent modification, and fail quickly and cleanly (rather than behaving erratically).

The `AbstractCollection`, `AbstractSet`, `AbstractList`, `AbstractSequentialList` and `AbstractMap` classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them. The API documentation for these classes describes precisely how each method is implemented so the implementer knows which methods should be overridden, given the performance of the "basic operations" of a specific implementation.

Design Goals

The main design goal was to produce an API that was reasonably small, both in size, and, more importantly, in "conceptual weight." It was critical that the new functionality not seem alien to current Java programmers; it had to augment current facilities, rather than replacing them. At the same time, the new API had to be powerful enough to provide all the advantages described above.

To keep the number of core interfaces small, the interfaces do not attempt to capture such subtle distinctions as mutability, modifiability, resizableability. Instead, certain calls in the core interfaces are *optional*, allowing implementations to throw an

`UnsupportedOperationException` to indicate that they do not support a specified optional operation. Of course, collection implementers must clearly document which optional operations are supported by an implementation.

To keep the number of methods in each core interface small, an interface contains a method only if either:

1. It is a truly *fundamental operation*: a basic operations in terms of which others could be reasonably defined,
2. There is a compelling performance reason why an important implementation would want to override it.

It was critical that all reasonable representations of collections interoperate well. This included arrays, which cannot be made to implement the `Collection` interface directly without changing the language. Thus, the framework includes methods to allow collections to be dumped into arrays, arrays to be viewed as collections, and maps to be viewed as collections.

[Copyright © 1995-99 Sun Microsystems, Inc.](#) All Rights Reserved.



Java Software

Please send comments to: collections-comments@java.sun.com