# Collections Framework Overview

## Introduction

The Java platform includes a *collections framework*. A *collection* is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

The primary advantages of a collections framework are that it:

- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be 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 requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which 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`, were retrofitted to implement the collection interfaces.
- **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**. Implementations designed for highly concurrent use.
- **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 primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

---

## Collection Interfaces

The *collection interfaces* are divided into two groups. The most basic interface, `java.util.Collection`, has the following descendants:

- `java.util.Set`
- `java.util.SortedSet`
- `java.util.NavigableSet`
- `java.util.Queue`
- `java.util.concurrent.BlockingQueue`
- `java.util.concurrent.TransferQueue`
- `java.util.Deque`
- `java.util.concurrent.BlockingDeque`

The other collection interfaces are based on `java.util.Map` and are not true collections. However, these interfaces contain *collection-view* operations, which enable them to be manipulated as collections. `Map` has the following offspring:

- `java.util.SortedMap`
- `java.util.NavigableMap`
- `java.util.concurrent.ConcurrentMap`
- `java.util.concurrent.ConcurrentNavigableMap`

Many of the modification methods in the collection interfaces are labeled *optional*. Implementations are permitted to not perform one or more of these operations, throwing a runtime exception (`UnsupportedOperationException`) if they are attempted. The documentation for each implementation must specify which optional operations are supported. Several terms are introduced to aid in this specification:

- Collections that do not support modification operations (such as `add`, `remove` and `clear`) are referred to as *unmodifiable*. Collections that are not unmodifiable are *modifiable.*

- Collections that additionally guarantee that no change in the `Collection` object will be visible are referred to as *immutable*. Collections that are not immutable are *mutable*.
- Lists that guarantee that their size remains constant even though the elements can 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 enables lists to advertise the fact that they support random access. This enables generic algorithms to change their behavior to provide good performance when applied to either random or sequential access lists.

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

- Be of a particular type.
- Be not 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 can result in an exception. Some restricted collections permit this usage.

---

## Collection Implementations

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

| Interface | Hash Table | Resizable Array | Balanced Tree | Linked List | Hash Table + Linked List |
|---|---|---|---|---|---|
| `Set` | `HashSet` | | `TreeSet` | | `LinkedHashSet` |
| `List` | | `ArrayList` | | `LinkedList` | |
| `Deque` | | `ArrayDeque` | | `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 can be used to add synchronization to many unsynchronized collections. All of the new implementations have *fail-fast iterators*, which detect invalid concurrent modification, and fail quickly and cleanly (rather than behaving erratically).

The `AbstractCollection`, `AbstractSet`, `AbstractList`, `AbstractSequentialList` and `AbstractMap` classes provide basic 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 must be overridden, given the performance of the basic operations of a specific implementation.

---

## Concurrent Collections

Applications that use collections from more than one thread must be carefully programmed. In general, this is known as *concurrent programming*. The Java platform includes extensive support for concurrent programming. See Java Concurrency Utilities for details.

Collections are so frequently used that various concurrent friendly interfaces and implementations of collections are included in the APIs. These types go beyond the synchronization wrappers discussed previously to provide features that are frequently needed in concurrent programming.

These concurrent-aware interfaces are available:

- `BlockingQueue`
- `TransferQueue`
- `BlockingDeque`
- `ConcurrentMap`
- `ConcurrentNavigableMap`

The following concurrent-aware implementation classes are available. See the API documentation for the correct usage of these implementations.

- `LinkedBlockingQueue`

- `ArrayBlockingQueue`
- `PriorityBlockingQueue`
- `DelayQueue`
- `SynchronousQueue`
- `LinkedBlockingDeque`
- `LinkedTransferQueue`
- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`
- `ConcurrentSkipListSet`
- `ConcurrentHashMap`
- `ConcurrentSkipListMap`

---

## Design Goals

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

To keep the number of core interfaces small, the interfaces do not attempt to capture such subtle distinctions as mutability, modifiability, and resizability. Instead, certain calls in the core interfaces are *optional*, enabling implementations to throw an `UnsupportedOperationException` to indicate that they do not support a specified optional operation. 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:

- It is a truly *fundamental operation*: a basic operations in terms of which others could be reasonably defined,
- 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 enable collections to be moved into arrays, arrays to be viewed as collections, and maps to be viewed as collections.