Java Tutorial Collections

# Introduction to Collections

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

* Interfaces: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* Implementations: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

# Interfaces

As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees — a Map is not a true Collection.

The following list describes the core collection interfaces:

* Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List. Also see [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section.
* Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also [The Set Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) section.
* List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List. Also see [The List Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html) section.
* Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queueimplementation must specify its ordering properties. Also see [The Queue Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html) section.

* Deque （双向队列）— a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see [The Deque Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html) section.

* Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map. Also see [The Map Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html) section.

The last two core collection interfaces are merely sorted versions of Set and Map:

* SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see [The SortedSet Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) section.
* SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see [The SortedMap Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-map.html) section.

## The Collection Interface

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

List<String> list = new ArrayList<String>(c);

Or — if you are using JDK 7 or later — you can use the diamond operator:

List<String> list = new ArrayList<>(c);

In JDK 8 and later, the Collection interface also exposes methods Stream<E> stream() and Stream<E> parallelStream(), for obtaining sequential or parallel streams from the underlying collection. (See the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html) for more information about using streams.)

### Traversing Collections（解析集合）

There are three ways to traverse collections: (1) using aggregate operations (2) with the for-each construct and (3) by using Iterators.

#### Aggregate Operations

In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

myShapesCollection.stream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

#### for-each Construct

for (Object o : collection)

System.out.println(o);

#### Iterators

An [Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

public interface Iterator<E> {

boolean hasNext();

E next();

void remove(); //optional

}

The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration. The remove method removes the last element that was returned by next from the underlying Collection. The remove method may be called only once per call to next and throws an exception if this rule is violated.

Use Iterator instead of the for-each construct when you need to:

* Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
* Iterate over multiple collections in parallel.

The following method shows you how to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements.

static void filter(Collection<?> c) {

for (Iterator<?> it = c.iterator(); it.hasNext(); )

if (!cond(it.next()))

it.remove();

}

#### Collection Interface Bulk Operations

* containsAll — returns true if the target Collection contains all of the elements in the specified Collection.
* addAll — adds all of the elements in the specified Collection to the target Collection.
* removeAll — removes from the target Collection all of its elements that are also contained in the specified Collection.
* retainAll — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
* clear — removes all elements from the Collection.

#### Collection Interface Array Operations

For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c.

Object[] a = c.toArray();

Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

String[] a = c.toArray(new String[0]);

## The Set Interface

The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet. [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html), which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. [TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html), which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. [LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html), which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Or, if using JDK 8 or later, you could easily collect into a Set using aggregate operations:

c.stream()

.collect(Collectors.toSet()); // no duplicates

Here's a slightly longer example that accumulates a Collection of names into a TreeSet:

Set<String> set = people.stream()

.map(Person::getName)

.collect(Collectors.toCollection(TreeSet::new));

### Set Interface Basic Operations

Using JDK 8 Aggregate Operations:

import java.util.\*;

import java.util.stream.\*;

public class FindDups {

public static void main(String[] args) {

Set<String> distinctWords = Arrays.asList(args).stream()

.collect(Collectors.toSet());

System.out.println(distinctWords.size()+

" distinct words: " +

distinctWords);

}

}

Using the for-each Construct:

import java.util.\*;

public class FindDups {

public static void main(String[] args) {

Set<String> s = new HashSet<String>();

for (String a : args)

s.add(a);

System.out.println(s.size() + " distinct words: " + s);

}

}

### Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

* s1.containsAll(s2) — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
* s1.addAll(s2) — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
* s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
* s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

### The List Interface

the List interface includes operations for the following:

* Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
* Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
* Iteration — extends Iterator semantics to take advantage of the list's sequential nature. ThelistIterator methods provide this behavior.
* Range-view — The sublist method performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose List implementations. [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html), which is usually the better-performing implementation, and [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) which offers better performance under certain circumstances.

#### Collection Operations

#### Positional Access and Search Operations

#### Iterators

The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces.

Here's the standard idiom for iterating backward through a list.

for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {

Type t = it.previous();

...

}

An initial call to previous would return the element whose index was index-1. Calls to next and previous can be intermixed, but you have to be a bit careful. The first call to previous returns the same element as the last call to next. Similarly, the first call to next after a sequence of calls to previous returns the same element as the last call to previous.

#### Range-View Operation

 For example, the following idiom removes a range of elements from a List.

list.subList(fromIndex, toIndex).clear();

Similar idioms can be constructed to search for an element in a range.

int i = list.subList(fromIndex, toIndex).indexOf(o);

int j = list.subList(fromIndex, toIndex).lastIndexOf(o);

Note that the preceding idioms return the index of the found element in the subList, not the index in the backing List.

### The Queue Interface

Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner.

The remove and poll methods differ in their behavior only when the queue is empty. Under these circumstances, remove throws NoSuchElementException, while poll returns null.

### The Deque Interface

A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points.

Predefined classes like [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html) and [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) implement the Deque interface.

### The Map Interface

The Java platform contains three general-purpose Map implementations: [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html), and[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html).

The remainder of this page discusses the Map interface in detail. But first, here are some more examples of collecting to Maps using JDK 8 aggregate operations. Modeling real-world objects is a common task in object-oriented programming, so it is reasonable to think that some programs might, for example, group employees by department:

// Group employees by department

Map<Department, List<Employee>> byDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));

Again, these are but a few examples of how to use the new JDK 8 APIs. For in-depth coverage of lambda expressions and aggregate operations see the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).

#### Map Interface Basic Operations

#### Map Interface Bulk Operations

#### Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

* keySet — the Set of keys contained in the Map.
* values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
* entrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

#### Fancy Uses of Collection Views: Map Algebra

#### Multimaps

A *multimap* is like a Map but it can map each key to multiple values.

// Read words from file and put into a simulated multimap

Map<String, List<String>> m = new HashMap<String, List<String>>();

#### Object Ordering

A List l may be sorted as follows.

Collections.sort(l);

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String and Date both implement the [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) interface. Comparable implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically.

##### Writing Your Own Comparable Types

The Comparable interface consists of the following method.

public interface Comparable<T> {

public int compareTo(T o);

}

The compareTo method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

##### Comparators

Like the Comparable interface, the Comparator interface consists of a single method.

public interface Comparator<T> {

int compare(T o1, T o2);

}

The compare method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second.

#### The SortedSet Interface

A [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html) is a [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a Comparator provided at SortedSet creation time. In addition to the normal Set operations, the SortedSet interface provides operations for the following:

* Range view — allows arbitrary range operations on the sorted set
* Endpoints — returns the first or last element in the sorted set
* Comparator access — returns the Comparator, if any, used to sort the set

##### Set Operations

The operations that SortedSet inherits from Set behave identically on sorted sets and normal sets with two exceptions:

* The Iterator returned by the iterator operation traverses the sorted set in order.
* The array returned by toArray contains the sorted set's elements in order.

##### Standard Constructors

##### Range-view Operations

int count = dictionary.subSet("doorbell", "pickle").size();

In like manner, the following one-liner removes all the elements beginning with the letter f.

dictionary.subSet("f", "g").clear();

##### Endpoint Operations

Object predecessor = ss.headSet(o).last();

##### Comparator Accessor

#### The SortedMap Interface

The SortedMap interface provides operations for normal Map operations and for the following:

* Range view — performs arbitrary range operations on the sorted map
* Endpoints — returns the first or the last key in the sorted map
* Comparator access — returns the Comparator, if any, used to sort the map

##### Map Operations

The operations SortedMap inherits from Map behave identically on sorted maps and normal maps with two exceptions:

* The Iterator returned by the iterator operation on any of the sorted map's Collection views traverse the collections in order.
* The arrays returned by the Collection views' toArray operations contain the keys, values, or entries in order.

#### Summary of Interfaces

The Java Collections Framework hierarchy consists of two distinct interface trees:

* The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections.
* The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
* The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
* The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
* The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.
* The second tree starts with the Map interface, which maps keys and values similar to a Hashtable.
* Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.

These interfaces allow collections to be manipulated independently of the details of their representation.

## Lesson: Implementations

This lesson describes the following kinds of implementations:

* **General-purpose implementations** are the most commonly used implementations, designed for everyday use. They are summarized in the table titled General-purpose-implementations.
* **Special-purpose implementations** are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
* **Concurrent implementations** are designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the java.util.concurrent package.
* **Wrapper implementations** are used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
* **Convenience implementations** are mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
* **Abstract implementations** are skeletal implementations that facilitate the construction of custom implementations — described later in the [Custom Collection Implementations](https://docs.oracle.com/javase/tutorial/collections/custom-implementations/index.html) section. An advanced topic, it's not particularly difficult, but relatively few people will need to do it.
* The general-purpose implementations are summarized in the following table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **General-purpose Implementations** | | | | | |
| **Interfaces** | **Hash table Implementations** | **Resizable array Implementations** | **Tree Implementations** | **Linked list Implementations** | **Hash table + Linked list Implementations** |
| Set | HashSet |  | TreeSet |  | LinkedHashSet |
| List |  | ArrayList |  | LinkedList |  |
| Queue |  |  |  |  |  |
| Deque |  | ArrayDeque |  | LinkedList |  |
| Map | HashMap |  | TreeMap |  | LinkedHashMap |

### Set Implementations

The Set implementations are grouped into general-purpose and special-purpose implementations.

#### General-Purpose Set Implementations

There are three general-purpose [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) implementations — [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html), [TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html), and [LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html).

HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees. If you need to use the operations in the SortedSet interface, or if value-ordered iteration is required, use TreeSet; otherwise, use HashSet. It's a fair bet that you'll end up using HashSet most of the time.

LinkedHashSet is in some sense intermediate between HashSet and TreeSet. Implemented as a hash table with a linked list running through it, it provides *insertion-ordered* iteration (least recently inserted to most recently) and runs nearly as fast as HashSet. The LinkedHashSet implementation spares its clients from the unspecified, generally chaotic ordering provided by HashSet without incurring the increased cost associated with TreeSet.

#### Special-Purpose Set Implementations

There are two special-purpose Set implementations — [EnumSet](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html) and [CopyOnWriteArraySet](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html).

The EnumSet class provides a static factory that makes it easy.

for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))

System.out.println(d);

CopyOnWriteArraySet is a Set implementation backed up by a copy-on-write array.

### List Implementations

#### General-Purpose List Implementations

There are two general-purpose [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) implementations — [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) and [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html). Most of the time, you'll probably use ArrayList, which offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the List, and it can take advantage of System.arraycopy when it has to move multiple elements at the same time. Think of ArrayList as Vector without the synchronization overhead.

There are two general-purpose [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) implementations — [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) and [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html). Most of the time, you'll probably use ArrayList, which offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the List, and it can take advantage of System.arraycopy when it has to move multiple elements at the same time. Think of ArrayList as Vector without the synchronization overhead.

#### Special-Purpose List Implementations

[CopyOnWriteArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) is a List implementation backed up by a copy-on-write array. This implementation is similar in nature to CopyOnWriteArraySet. No synchronization is necessary, even during iteration, and iterators are guaranteed never to throw ConcurrentModificationException. This implementation is well suited to maintaining event-handler lists, in which change is infrequent, and traversal is frequent and potentially time-consuming.

### Map Implementations

Map implementations are grouped into general-purpose, special-purpose, and concurrent implementations.

#### General-Purpose Map Implementations

If you need SortedMap operations or key-ordered Collection-view iteration, use TreeMap; if you want maximum speed and don't care about iteration order, use HashMap; if you want near-HashMap performance and insertion-order iteration, use LinkedHashMap.

#### Special-Purpose Map Implementations

There are three special-purpose Map implementations — [EnumMap](https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html), [WeakHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html) and [IdentityHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/IdentityHashMap.html). EnumMap, which is internally implemented as an array, is a high-performance Map implementation for use with enum keys. This implementation combines the richness and safety of the Map interface with a speed approaching that of an array. If you want to map an enum to a value, you should always use an EnumMap in preference to an array.

WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage-collected when its key is no longer referenced outside of the WeakHashMap. This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

IdentityHashMap is an identity-based Map implementation based on a hash table. This class is useful for topology-preserving object graph transformations, such as serialization or deep-copying. To perform such transformations, you need to maintain an identity-based "node table" that keeps track of which objects have already been seen. Identity-based maps are also used to maintain object-to-meta-information mappings in dynamic debuggers and similar systems. Finally, identity-based maps are useful in thwarting "spoof attacks" that are a result of intentionally perverse equals methods because IdentityHashMap never invokes the equals method on its keys. An added benefit of this implementation is that it is fast.

#### Concurrent Map Implementations

The [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) package contains the [ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html) interface, which extends Map with atomic putIfAbsent, remove, and replace methods, and the [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html) implementation of that interface.

ConcurrentHashMap is a highly concurrent, high-performance implementation backed up by a hash table.

### Queue Implementations

The Queue implementations are grouped into general-purpose and concurrent implementations.

#### General-Purpose Queue Implementations

The [PriorityQueue](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html) class is a priority queue based on the *heap* data structure. This queue orders elements according to the order specified at construction time, which can be the elements' natural ordering or the ordering imposed by an explicit Comparator.

#### Concurrent Queue Implementations

The java.util.concurrent package contains a set of synchronized Queue interfaces and classes. [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) extends Queue with operations that wait for the queue to become nonempty when retrieving an element and for space to become available in the queue when storing an element. This interface is implemented by the following classes:

* [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html) — an optionally bounded FIFO blocking queue backed by linked nodes
* [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html) — a bounded FIFO blocking queue backed by an array
* [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html) — an unbounded blocking priority queue backed by a heap
* [DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html) — a time-based scheduling queue backed by a heap
* [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html) — a simple rendezvous mechanism that uses the BlockingQueue interface

In JDK 7, [TransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html) is a specialized BlockingQueue in which code that adds an element to the queue has the option of waiting (blocking) for code in another thread to retrieve the element. TransferQueue has a single implementation:

* [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html) — an unbounded TransferQueue based on linked nodes

### Deque Implementations

#### General-Purpose Deque Implementations

The general-purpose implementations include LinkedList and ArrayDeque classes. The Deque interface supports insertion, removal and retrieval of elements at both ends. The [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html) class is the resizable array implementation of the Deque interface, whereas the [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) class is the list implementation.

The LinkedList implementation is more flexible than the ArrayDeque implementation. LinkedList implements all optional list operations. null elements are allowed in the LinkedList implementation but not in the ArrayDeque implementation.

In terms of efficiency, ArrayDeque is more efficient than the LinkedList for add and remove operation at both ends. The best operation in a LinkedList implementation is removing the current element during the iteration. LinkedList implementations are not ideal structures to iterate.

The LinkedList implementation consumes more memory than the ArrayDeque implementation. For the ArrayDeque instance traversal use any of the following:

##### foreach

The foreach is fast and can be used for all kinds of lists.

ArrayDeque<String> aDeque = new ArrayDeque<String>();

. . .

for (String str : aDeque) {

System.out.println(str);

}

##### Iterator

The Iterator can be used for the forward traversal on all kinds of lists for all kinds of data.

ArrayDeque<String> aDeque = new ArrayDeque<String>();

. . .

for (Iterator<String> iter = aDeque.iterator(); iter.hasNext(); ) {

System.out.println(iter.next());

}

The ArrayDeque class is used in this tutorial to implement the Deque interface. The complete code of the example used in this tutorial is available in[ArrayDequeSample](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/ArrayDequeSample.java). Both the LinkedList and ArrayDeque classes do not support concurrent access by multiple threads.

#### Concurrent Deque Implementations

The [LinkedBlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html) class is the concurrent implementation of the Deque interface. If the deque is empty then methods such as takeFirst and takeLast wait until the element becomes available, and then retrieves and removes the same element.

### Wrapper Implementations

Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers. For design pattern fans, this is an example of the *decorator* pattern. Although it may seem a bit exotic, it's really pretty straightforward.

#### Synchronization Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html),[Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html), [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html), [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html), [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html), and [SortedMap](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html) — has one static factory method.

public static <T> Collection<T> synchronizedCollection(Collection<T> c);

public static <T> Set<T> synchronizedSet(Set<T> s);

public static <T> List<T> synchronizedList(List<T> list);

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);

public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);

Each of these methods returns a synchronized (thread-safe) Collection backed up by the specified collection. To guarantee serial access, all access to the backing collection must be accomplished through the returned collection. The easy way to guarantee this is not to keep a reference to the backing collection. Create the synchronized collection with the following trick.

List<Type> list = Collections.synchronizedList(new ArrayList<Type>());

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a [Vector](https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html).

In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation. The following is the idiom to iterate over a wrapper-synchronized collection.

Collection<Type> c = Collections.synchronizedCollection(myCollection);

synchronized(c) {

for (Type e : c)

foo(e);

}

If an explicit iterator is used, the iterator method must be called from within the synchronized block. Failure to follow this advice may result in nondeterministic behavior. The idiom for iterating over a Collection view of a synchronized Map is similar. It is imperative that the user synchronize on the synchronized Map when iterating over any of its Collection views rather than synchronizing on the Collection view itself, as shown in the following example.

Map<KeyType, ValType> m = Collections.synchronizedMap(new HashMap<KeyType, ValType>());

...

Set<KeyType> s = m.keySet();

...

// Synchronizing on m, not s!

synchronized(m) {

while (KeyType k : s)

foo(k);

}

One minor downside of using wrapper implementations is that you do not have the ability to execute any noninterface operations of a wrapped implementation. So, for instance, in the preceding List example, you cannot call ArrayList's [ensureCapacity](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ensureCapacity-int-) operation on the wrapped ArrayList.

#### Unmodifiable Wrappers

Unlike synchronization wrappers, which add functionality to the wrapped collection, the unmodifiable wrappers take functionality away. In particular, they take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an UnsupportedOperationException. Unmodifiable wrappers have two main uses, as follows:

* To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.
* To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, clients can look but not modify, while you maintain full access.

Like synchronization wrappers, each of the six core Collection interfaces has one static factory method.

public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);

public static <T> Set<T> unmodifiableSet(Set<? extends T> s);

public static <T> List<T> unmodifiableList(List<? extends T> list);

public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);

public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);

public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);

#### Checked Interface Wrappers

The Collections.checked interface wrappers are provided for use with generic collections. These implementations return a dynamically type-safe view of the specified collection, which throws a ClassCastException if a client attempts to add an element of the wrong type. The generics mechanism in the language provides compile-time (static) type-checking, but it is possible to defeat this mechanism. Dynamically type-safe views eliminate this possibility entirely.

### Convenience Implementations

#### List View of an Array(数组的List视图)

The [Arrays.asList](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-) method returns a List view of its array argument.

If you need a fixed-size List, it's more efficient than any general-purpose List implementation. This is the idiom.

List<String> list = Arrays.asList(new String[size]);

#### Immutable Multiple-Copy List

Occasionally you'll need an immutable List consisting of multiple copies of the same element. The [Collections.nCopies](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#nCopies-int-T-) method returns such a list.

for example, suppose you want an ArrayList initially consisting of 1,000 null elements. The following incantation does the trick.

List<Type> list = new ArrayList<Type>(Collections.nCopies(1000, (Type)null);

#### Immutable Singleton Set

Sometimes you'll need an immutable *singleton* Set, which consists of a single, specified element. The [Collections.singleton](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singleton-T-) method returns such a Set. One use of this implementation is to remove all occurrences of a specified element from a Collection.

c.removeAll(Collections.singleton(e));

#### Empty Set, List, and Map Constants

The [Collections](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html) class provides methods to return the empty Set, List, and Map — [emptySet](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#emptySet--), [emptyList](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#emptyList--), and [emptyMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#emptyMap--). The main use of these constants is as input to methods that take a Collection of values when you don't want to provide any values at all, as in this example.

tourist.declarePurchases(Collections.emptySet());

### Summary of Implementations

The Java Collections Framework provides several general-purpose implementations of the core interfaces:

* For the Set interface, HashSet is the most commonly used implementation.
* For the List interface, ArrayList is the most commonly used implementation.
* For the Map interface, HashMap is the most commonly used implementation.
* For the Queue interface, LinkedList is the most commonly used implementation.
* For the Deque interface, ArrayDeque is the most commonly used implementation.