
NNG


iGO
navigation

Introduction to Collections

What is a Collections Framework?

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

- Interfaces
- Implementations
- Algorithms

Benefits of the Java Collections Framwork

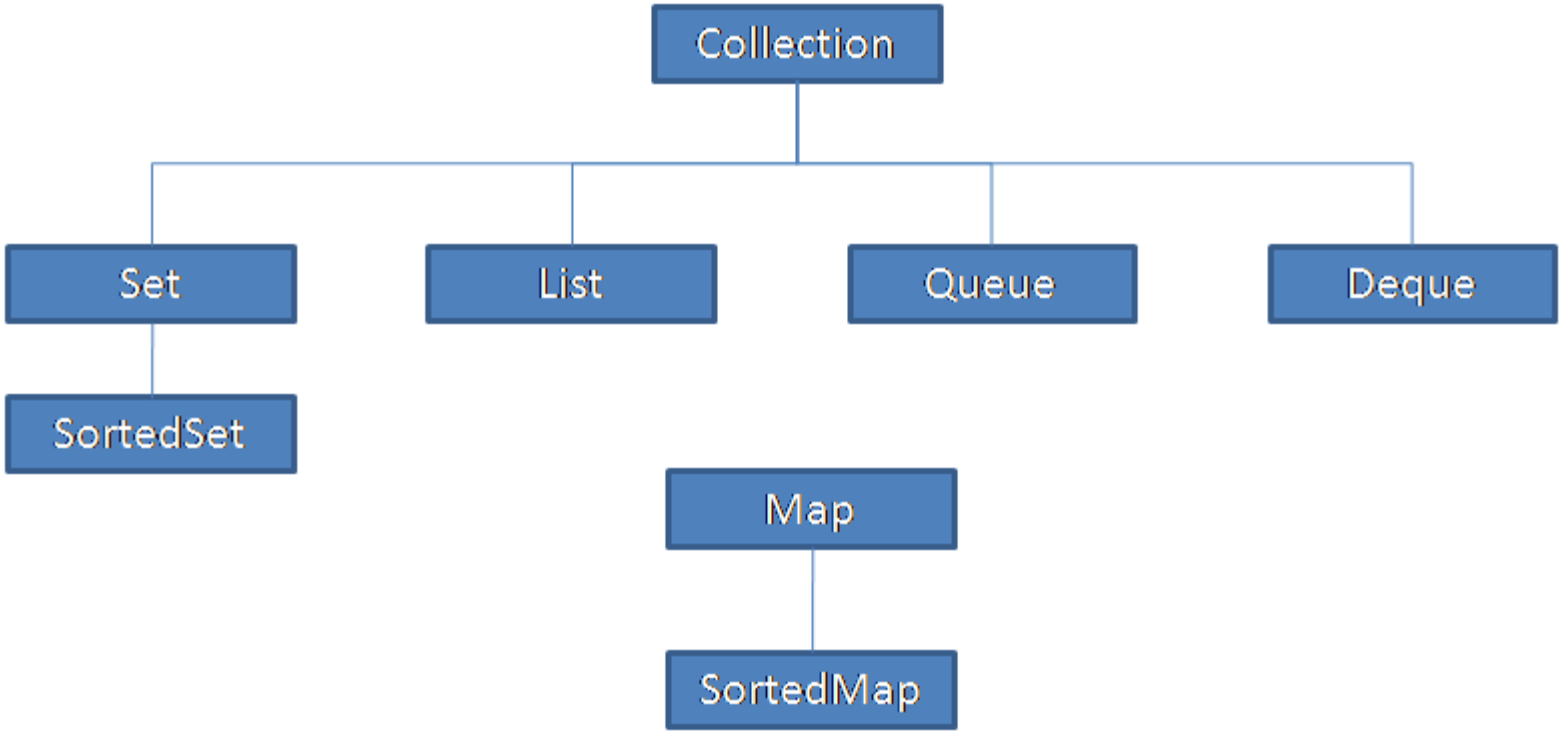
The Java Collections Framework provides the following benefits:

- **Reduces programming effort**: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality**: This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs**: The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

Benefits of the Java Collections Framwork 2.

- Reduces effort to learn and to use new APIs: Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- Reduces effort to design new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- Fosters software reuse: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Interfaces



Interfaces 2.

- 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.
- 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.
- 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 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.

Interfaces 3.

- Deque — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.
- 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.
- 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.
- 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.

Set

- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.
- The Java platform contains three general-purpose Set implementations: **HashSet, TreeSet,** and **LinkedHashSet**.
- **HashSet**, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.
- **TreeSet**, which stores its elements in a red-black tree, orders its elements based on their values
- **LinkedHashSet**, 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)

List

- A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, 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. The **listIterator** 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**, which is usually the better-performing implementation
- **LinkedList** which offers better performance under certain circumstances

Queue

- Queue is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface follows:
  - E element();
  - boolean offer(E e);
  - E peek();
  - E poll();
  - E remove();
- Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner.
- Queue implementations generally do not allow insertion of null elements (expect **LinkedList**).

Deque

- Deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points.
- Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner.
- Queue implementations generally do not allow insertion of null elements (expect **LinkedList**).

Type of Operation	First Elemenent	Last Elemenent (End)
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

Map

- A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value
- The Map interface includes methods for:
  - basic operations (such as **put**, **get**, **remove**, **containsKey**, **containsValue**, **size**, and **empty**)
  - bulk operations (such as **putAll** and **clear**)
  - collection views (such as **keySet**, **entrySet**, and **values**)
- The Java platform contains three general-purpose Map implementations: **HashMap**, **TreeMap**, and **LinkedHashMap**. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet, as described in The Set Interface section.

Object Ordering

- **Comparable** implementations provide a natural ordering for a class, which allows objects of that class to be sorted automatically:
  - public int compareTo(T o);
- **Comparator** to use different logic than natural ordering, or compare objects without natural ordering:
  - int compare(T o1, T o2);

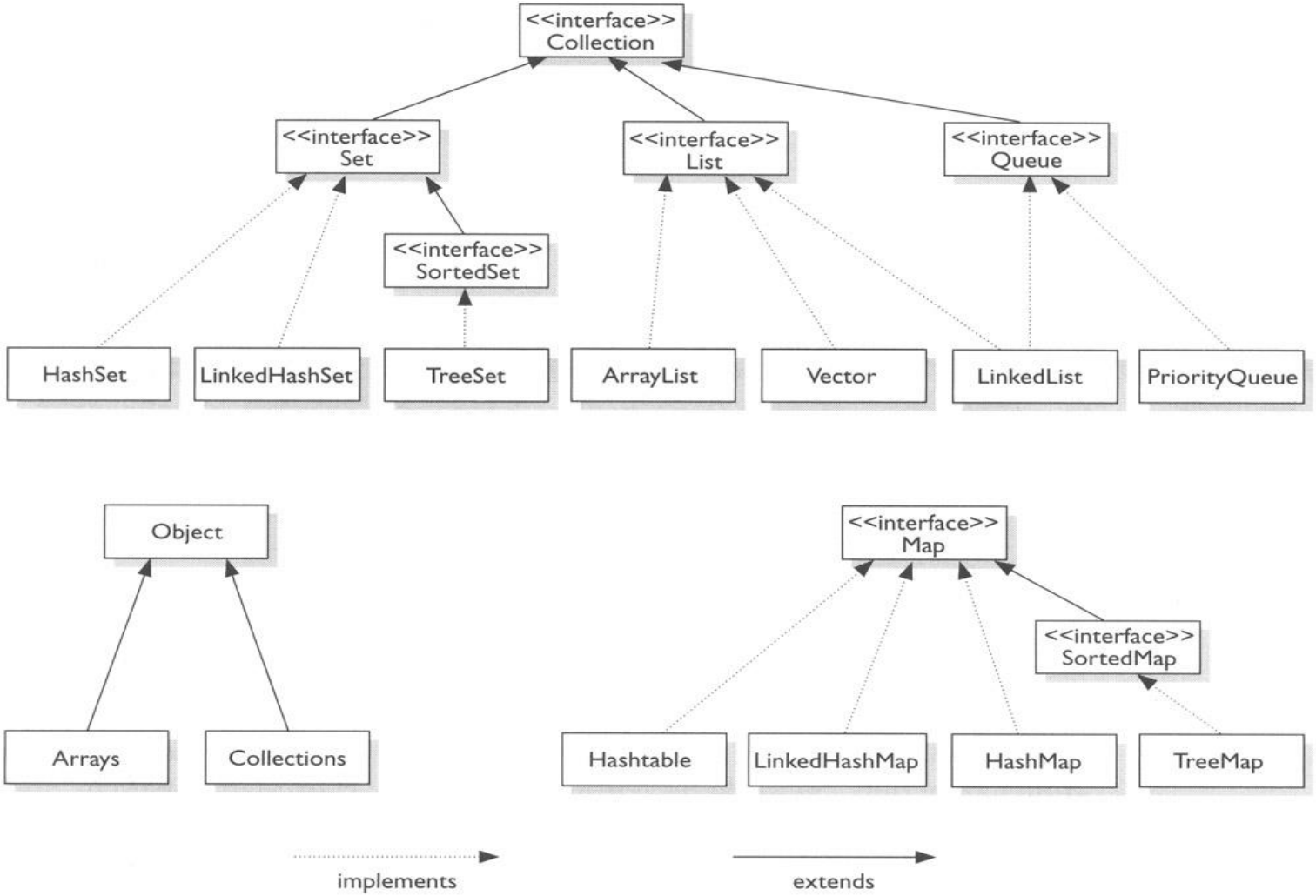
SortedSet

- A SortedSet is a Set 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

SortedMap

- SortedMap is a Map that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of the SortedMap creation.
- In addition to the normal Map operations, the SortedMap interface provides operations for the following:
  - **Range view** — allows arbitrary range operations on the sorted map
  - **Endpoints** — returns the first or last element in the sorted map
  - **Comparator access** — returns the Comparator, if any, used to sort the map

Overview


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
LinkedHashMap
HashMap
TreeMap
implements
extends

Streams (Java 8)

- Streams are designed to work with Java **lambda expressions**.
- Methods (common):
  - filter()
  - collect()
  - min(), max(),
  - reduce()

References

- https://docs.oracle.com/javase/tutorial/collections/intro/index.html
- http://web.archive.org/web/20110626160836/http://www.coderfriendly.com/wp-content/uploads/2009/05/java_collections_v2.pdf

Questions?