Java Collections Framework Tutorial

Java Collections are one of the core frameworks of Java language. We use Collections almost in every application, this tutorial will explain **Java Collections Framework** in detail.

Java Collection Framework

What is Java Collections Framework?

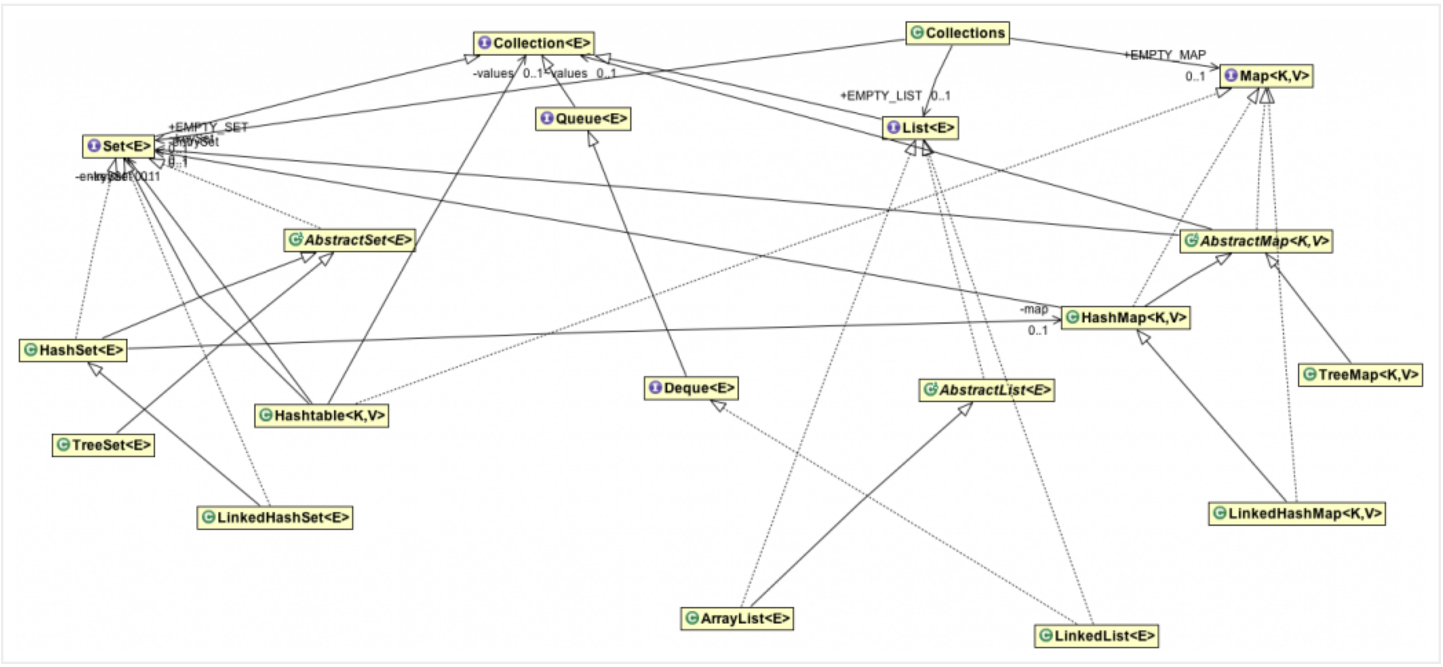
Collections are like containers that groups multiple items in a single unit. For example; a jar of chocolates, list of names etc. Collections are used almost in every programming language and when Java arrived, it also came with few Collection classes; **Vector**, **Stack**, **Hashtable**, **Array**. Java 1.2 provided **Collections Framework** that is architecture to represent and manipulate Collections in a standard way. Java Collections Framework consists of following parts:

- **Interfaces**: Java Collections Framework interfaces provides the abstract data type to represent collection. `java.util.Collection` is the root interface of Collections Framework. It is on the top of Collections framework hierarchy. It contains some important methods such as `size()`, `iterator()`, `add()`, `remove()`, `clear()` that every Collection class must implement.Some other important interfaces are `java.util.List`, `java.util.Set`, `java.util.Queue` and `java.util.Map`. Map is the only interface that doesn't inherits from Collection interface but it's part of Collections framework. All the collections framework interfaces are present in `java.util` package.
- **Implementation Classes**: Java provides core implementation classes for collections. We can use them to create different types of collections in our program. Some important collection classes are `ArrayList`, `LinkedList`, `HashMap`, `TreeMap`, `HashSet`, `TreeSet`.These classes solve most of our programming needs but if we need some special collection class, we can extend them to create our custom collection class.
  Java 1.5 came up with thread-safe collection classes that allowed to modify Collections while

iterating over it, some of them are `CopyOnWriteArrayList`, `ConcurrentHashMap`, `CopyOnWriteArraySet`. These classes are in java.util.concurrent package. All the collection classes are present in `java.util` and `java.util.concurrent` package.

- **Algorithms**: Algorithms are useful methods to provide some common functionalities, for example searching, sorting and shuffling.

Below class diagram shows Collections Framework hierarchy. For simplicity I have included only commonly used interfaces and classes.



Benefits of Java Collections Framework

Java Collections framework have following benefits:

- **Reduced Development Effort** – It comes with almost all common types of collections and useful methods to iterate and manipulate the data. So we can concentrate more on business logic rather than designing our collection APIs.
- **Increased Quality** – Using core collection classes that are well tested increases our program quality rather than using any home developed data structure.
- **Reusability and Interoperability**
- **Reduce effort** – to learn any new API if we use core collection API classes.

Java Collections Interfaces

Core collection interfaces are the foundation of the Java Collections Framework. Note that all the core collection interfaces are generic; for example `public interface Collection<E>`. The <E> syntax is for Generics and when we declare Collection, we should use it to specify the type of Object it can contain. It helps in reducing run-time errors by type-checking the Objects at compile-time.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. If an unsupported operation is invoked, a collection implementation throws an `UnsupportedOperationException`.

Collection Interface

This is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface.

The interface has methods to tell you how many elements are in the collection (`size`, `isEmpty`), to check whether a given object is in the collection (`contains`), to add and remove an element from the collection (`add`, `remove`), and to provide an iterator over the collection (`iterator`).

Collection interface also provides bulk operations methods that work on entire collection – `containsAll`, `addAll`, `removeAll`, `retainAll`, `clear`.

The `toArray` methods are provided as a bridge between collections and older APIs that expect arrays on input.

Iterator Interface

Iterator interface provides methods to iterate over any Collection. We can get iterator instance from a Collection using `iterator` method. Iterator takes the place of `Enumeration` in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Iterators in collection classes implement Iterator Design Pattern.

Set Interface

Set is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

The Java platform contains three general-purpose Set implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. Set interface doesn't allow random-access to an element in the Collection. You can use iterator or foreach loop to traverse the elements of a Set.

List Interface

List is an ordered collection and can contain duplicate elements. You can access any element from it's index. List is more like array with dynamic length. List is one of the most used Collection type. `ArrayList` and `LinkedList` are implementation classes of List interface.

List interface provides useful methods to add an element at specific index, remove/replace element based on index and to get a sub-list using index.

```java
List strList = new ArrayList<>();
//add at last
strList.add(0, "0");
//add at specified index
strList.add(1, "1");
//replace
strList.set(1, "2");
//remove
strList.remove("1");
```

Collections class provide some useful algorithm for List – `sort`, `shuffle`, `reverse`, `binarySearch` etc.

Queue Interface

Queue is 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.

Dequeue Interface

A linear collection that supports element insertion and removal at both ends. The name deque is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element.

Map Interface

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 Java platform contains three general-purpose Map implementations: `HashMap`, `TreeMap`, and `LinkedHashMap`.

The basic operations of Map are `put`, `get`, `containsKey`, `containsValue`, `size`, and `isEmpty`.

ListIterator Interface

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator’s current position in the list.

A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

SortedSet Interface

SortedSet is 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 Interface

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.

Java Collection Classes

Java Collections framework comes with many implementation classes for the interfaces. Most common implementations are ArrayList, HashMap and HashSet. Java 1.5 included Concurrent implementations; for example ConcurrentHashMap and CopyOnWriteArrayList. Usually Collection classes are not thread-safe and their iterator is fail-fast. In this section, we will learn about commonly used collection classes.

HashSet Class

This is the basic implementation the Set interface that is backed by a Hashtable. It makes no guarantees for iteration order of the set and permits the **null** element.

This class offers constant time performance for basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. We can set the initial capacity and load factor for this collection. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

TreeSet Class

A `NavigableSet` implementation based on a `TreeMap`. The elements are ordered using their natural ordering, or by a `Comparator` provided at set creation time, depending on which constructor is used.

Refer: Java Comparable Comparator

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal.

ArrayList Class

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Further reading: Java ArrayList and Iterator

LinkedList Class

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

HashMap Class

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits null. This class makes no guarantees for the order of the map.

This implementation provides constant-time performance for the basic operations (`get` and `put`). It provides constructors to set initial capacity and load factor for the collection.

Further Read: HashMap vs ConcurrentHashMap

TreeMap Class

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

PriorityQueue Class

Queue processes it's elements in FIFO order but sometimes we want elements to be processed based on their priority. We can use PriorityQueue in this case and we need to provide a Comparator implementation while instantiation the PriorityQueue. PriorityQueue doesn't allow null values and it's unbounded. For more details about this, please head over to **Java Priority Queue** where you can check it's usage with a sample program.

Collections class

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse etc.

Synchronized Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — Collection, Set, List, Map, SortedSet, and SortedMap — has one static factory method.

```
public static  Collection synchronizedCollection(Collection c);
public static  Set synchronizedSet(Set s);
public static  List synchronizedList(List list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static  SortedSet synchronizedSortedSet(SortedSet 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.

Unmodifiable wrappers

Unmodifiable wrappers take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an `UnsupportedOperationException`. It's main usage are;

- 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.

These methods are;

```
public static  Collection unmodifiableCollection(Collection<? extends T> c);
public static  Set unmodifiableSet(Set<? extends T> s);
public static  List unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)
public static  SortedSet unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? exten
```

Thread Safe Collections

Java 1.5 Concurrent package (`java.util.concurrent`) contains thread-safe collection classes that allow collections to be modified while iterating. By design iterator is fail-fast and throws ConcurrentModificationException. Some of these classes are `CopyOnWriteArrayList`, `ConcurrentHashMap`, `CopyOnWriteArraySet`.

Read these posts to learn about them in more detail.

- Avoid ConcurrentModificationException
- CopyOnWriteArrayList Example
- HashMap vs ConcurrentHashMap

Collections API Algorithms

Java Collections Framework provides algorithm implementations that are commonly used such as sorting and searching. Collections class contain these method implementations. Most of these algorithms work on List but some of them are applicable for all kinds of collections.

Sorting

The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship. Two forms of the operation are provided. The simple form takes a List and sorts it according to its elements' natural ordering. The second form of sort takes a Comparator in addition to a List and sorts the elements with the Comparator.

Shuffling

The shuffle algorithm destroys any trace of order that may have been present in a List. That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. This algorithm is useful in implementing games of chance.

Searching

The binarySearch algorithm searches for a specified element in a sorted List. This algorithm has two forms. The first takes a List and an element to search for (the "search key"). This form assumes that the List is sorted in ascending order according to the natural ordering of its elements. The second form takes a Comparator in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator. The sort algorithm can be used to sort the List prior to calling binarySearch.

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more Collections.

- **frequency**: counts the number of times the specified element occurs in the specified collection
- **disjoint**: determines whether two Collections are disjoint; that is, whether they contain no elements in common

Min and Max values

The min and the max algorithms return, respectively, the minimum and maximum element contained in a specified Collection. Both of these operations come in two forms. The simple form takes only a Collection and returns the minimum (or maximum) element according to the elements' natural ordering.
The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

Java 8 Collections API Features

Java 8 biggest changes are related to Collection APIs. Some of the important changes and improvements are:

1. Introduction of Stream API for sequential as well as parallel processing, you should read Java Stream API Tutorial for more details.
2. Iterable interface has been extended with forEach() default method for iterating over a collection.
3. Lambda Expression and Functional interfaces are mostly beneficial with Collection API classes.

Collection classes in a Nutshell

Below table provides basic details of commonly used collection classes.

Download URL: Java Collection Classes

Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element	Thread Safety
ArrayList	Yes	Yes	No	Yes	Yes	No
LinkedList	Yes	No	No	Yes	Yes	No
HashSet	No	No	No	No	Yes	No
TreeSet	Yes	No	No	No	No	No
HashMap	No	Yes	Yes	No	Yes	No
TreeMap	Yes	Yes	Yes	No	No	No
Vector	Yes	Yes	No	Yes	Yes	Yes
Hashtable	No	Yes	Yes	No	No	Yes
Properties	No	Yes	Yes	No	No	Yes

Stack	Yes	No	No	Yes	Yes	Yes
CopyOnWriteArrayList	Yes	Yes	No	Yes	Yes	Yes
ConcurrentHashMap	No	Yes	Yes	No	No	Yes
CopyOnWriteArraySet	No	No	No	No	Yes	Yes

I hope this tutorial explains most of the topics in java collections framework, please share your opinion with comments.