# Annotated Outline of Collections Framework

The collections framework consists of:

- **Collection Interfaces** - The primary means by which collections are manipulated.
  - **Collection** - A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.
  - **Set** - The familiar set abstraction. No duplicate elements permitted. May or may not be ordered. Extends the `Collection` interface.
  - **List** - Ordered collection, also known as a *sequence*. Duplicates are generally permitted. Allows positional access. Extends the `Collection` interface.
  - **Queue** - A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations.
  - **Deque** - A *double ended queue*, supporting element insertion and removal at both ends. Extends the `Queue` interface.
  - **Map** - A mapping from keys to values. Each key can map to at most one value.
  - **SortedSet** - A set whose elements are automatically sorted, either in their *natural ordering* (see the `Comparable` interface), or by a `Comparator` object provided when a `SortedSet` instance is created. Extends the `Set` interface.
  - **SortedMap** - A map whose mappings are automatically sorted by key, either in the keys' *natural ordering* or by a comparator provided when a `SortedMap` instance is created. Extends the `Map` interface.
  - **NavigableSet** - A `SortedSet` extended with navigation methods reporting closest matches for given search targets. A `NavigableSet` may be accessed and traversed in either ascending or descending order.
  - **NavigableMap** - A `SortedMap` extended with navigation methods returning the closest matches for given search targets. A `NavigableMap` may be accessed and traversed in either ascending or descending key order.
  - **BlockingQueue** - A `Queue` with operations that wait for the queue to become non-empty when retrieving an element, and that wait for space to become available in the queue when storing an element. (This interface is part of `java.util.concurrent`.)
  - **BlockingDeque** - A `Deque` with operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element. Extends both the `Deque` and `BlockingQueue` interfaces. (This interface is part of `java.util.concurrent`.)
  - **ConcurrentMap** - A `Map` with atomic `putIfAbsent`, `remove`, and `replace` methods. (This interface is part of `java.util.concurrent`.)
  - **ConcurrentNavigableMap** - A `ConcurrentMap` that is also a `NavigableMap`.
- **General-Purpose Implementations** - The primary implementations of the collection interfaces.
  - **HashSet** - Hash table implementation of the `Set` interface. The best all-around implementation of the `Set` interface.
  - **TreeSet** - Red-black tree implementation of the `NavigableSet` interface.
  - **LinkedHashSet** - Hash table and linked list implementation of the `Set` interface. An insertion-ordered `Set` implementation that runs nearly as fast as `HashSet`.
  - **ArrayList** - Resizable-array implementation of the `List` interface. (Essentially an unsynchronized `Vector`.) The best all-around implementation of the `List` interface.
  - **ArrayDeque** - Efficient resizable-array implementation of the `Deque` interface.
  - **LinkedList** - Doubly-linked list implementation of the `List` interface. May provide better performance than the `ArrayList` implementation if elements are frequently inserted or deleted within the list. Also implements the `Deque` interface. When accessed via the `Queue` interface, `LinkedList` behaves as a FIFO queue.
  - **PriorityQueue** - Heap implementation of an unbounded priority queue.
  - **HashMap** - Hash table implementation of the `Map` interface. (Essentially an unsynchronized `Hashtable` that supports `null` keys and values.) The best all-around implementation of the `Map` interface.
  - **TreeMap** Red-black tree implementation of the `NavigableMap` interface.
  - **LinkedHashMap** - Hash table and linked list implementation of the `Map` interface. An insertion-ordered `Map` implementation that runs nearly as fast as `HashMap`. Also useful for building caches (see removeEldestEntry(Map.Entry) ).
- **Wrapper Implementations** - Functionality-enhancing implementations for use with other implementations. Accessed solely through static factory methods.
  - **Collections.unmodifiable*Interface*** - Return an unmodifiable view of a specified collection that throws an `UnsupportedOperationException` if the user attempts to modify it.
  - **Collections.synchronized*Interface*** - Return a synchronized collection that is backed by the specified (typically unsynchronized) collection. As long as all accesses to the backing collection are through the returned collection, thread-safety is guaranteed.
  - **Collections.checked*Interface*** - Return a dynamically typesafe 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 typesafe views eliminate this possibility entirely.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.

- **Arrays.asList** - Allows an array to be viewed as a list.
- **EMPTY_SET, EMPTY_LIST and EMPTY_MAP** - Constants representing the empty set and list (immutable).
- **singleton, singletonList, and singletonMap** - Returns an immutable "singleton" set, list, or map, containing only the specified object (or key-value mapping).
- **nCopies** - Returns an immutable list consisting of n copies of a specified object.

- **Legacy Implementations** - Older collection classes have been retrofitted to implement the collection interfaces.
  - **Vector** - Synchronized resizable-array implementation of the `List` interface with additional "legacy methods."
  - **Hashtable** - Synchronized hash table implementation of the `Map` interface that does not allow `null` keys or values, with additional "legacy methods."
- **Special Purpose Implementations**
  - **WeakHashMap** - An implementation of the `Map` interface that stores only *weak references* to its keys. Storing only weak references allows key-value pairs to be garbage-collected when the 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** - 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" resulting from intentionally perverse equals methods. (`IdentityHashMap` never invokes the equals method on its keys.) An added benefit of this implementation is that it is fast.
  - **CopyOnWriteArrayList** - a `List` implementation backed by an copy-on-write array. All mutative operations (such as `add`, `set`, and `remove`) are implemented by making a new copy of the array. 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 (where change is infrequent, and traversal is frequent and potentially time-consuming).
  - **CopyOnWriteArraySet** - A `Set` implementation backed by a copy-on-write array. This implementation is similar in nature to `CopyOnWriteArrayList`. Unlike most `Set` implementations, the `add`, `remove`, and `contains` methods require time proportional to the size of the set. This implementation is well-suited to maintaining event-handler lists that must prevent duplicates.
  - **EnumSet** - a high-performance `Set` implementation backed by a bit-vector. All elements of each `EnumSet` instance must be elements of a single enum type.
  - **EnumMap** - a high-performance `Map` implementation backed by an array. All keys in each `EnumMap` instance must be elements of a single enum type.
- **Concurrent Implementations** - These implementations are part of `java.util.concurrent`.
  - **ConcurrentLinkedQueue** - An unbounded FIFO (first-in first-out) queue based on linked nodes.
  - **LinkedBlockingQueue** - An optionally bounded FIFO blocking queue backed by linked nodes.
  - **ArrayBlockingQueue** - A bounded FIFO blocking queue backed by an array.
  - **PriorityBlockingQueue** - An unbounded blocking priority queue backed by a priority heap.
  - **DelayQueue** - A time-based scheduling queue backed by a priority heap.
  - **SynchronousQueue** - A simple rendezvous mechanism utilizing the `BlockingQueue` interface.
  - **LinkedBlockingDeque** - An optionally bounded FIFO blocking deque backed by linked nodes.
  - **ConcurrentHashMap** - A highly concurrent, high-performance `ConcurrentMap` implementation based on a hash table. This implementation never blocks when performing retrievals and allows the client to select the concurrency level for updates. It is intended as a drop-in replacement for `Hashtable`: in addition to implementing `ConcurrentMap`, it supports all of the "legacy" methods peculiar to `Hashtable`.
  - **ConcurrentSkipListSet** - Skip list implementation of the `NavigableSet` interface.
  - **ConcurrentSkipListMap** - Skip list implementation of the `ConcurrentNavigableMap` interface.
- **Abstract Implementations** - Skeletal implementations of the collection interfaces to facilitate custom implementations.
  - **AbstractCollection** - Skeletal `Collection` implementation that is neither a set nor a list (such as a "bag" or multiset).
  - **AbstractSet** - Skeletal `Set` implementation.
  - **AbstractList** - Skeletal `List` implementation backed by a random-access data store (such as an array).
  - **AbstractSequentialList** - Skeletal `List` implementation backed by a sequential-access data store (such as a linked list).
  - **AbstractQueue** - Skeletal `Queue` implementation.
  - **AbstractMap** - Skeletal `Map` implementation.
- **Algorithms** - The **Collections** class contains these useful static methods:
  - **sort(List)** - Sorts a list using a merge sort algorithm, which provides average-case performance comparable to a high-quality quicksort, guaranteed O(n*log n) performance (unlike quicksort), and *stability* (unlike quicksort). (A stable sort is one that does not reorder equal elements.)
  - **binarySearch(List, Object)** - Searches for an element in an ordered list using the binary search algorithm.
  - **reverse(List)** - Reverses the order of the elements in the a list.
  - **shuffle(List)** - Randomly permutes the elements in a list.
  - **fill(List, Object)** - Overwrites every element in a list with the specified value.
  - **copy(List dest, List src)** - Copies the source list into the destination list.
  - **min(Collection)** - Returns the minimum element in a collection.
  - **max(Collection)** - Returns the maximum element in a collection.
  - **rotate(List list, int distance)** - Rotates all of the elements in the list by the specified distance.
  - **replaceAll(List list, Object oldVal, Object newVal)** - Replaces all occurrences of one specified value with another.
  - **indexOfSubList(List source, List target)** - Returns the index of the first sublist of source that is equal to target.
  - **lastIndexOfSubList(List source, List target)** - Returns the index of the last sublist of source that is equal to target.
  - **swap(List, int, int)** - Swaps the elements at the specified positions in the specified list.
  - **frequency(Collection, Object)** - Counts the number of times the specified element occurs in the specified collection.
  - **disjoint(Collection, Collection)** - Determines whether two collections are disjoint, in other words, whether they contain no elements in

common.
- **addAll(Collection<? super T>, T...)** - Adds all of the elements in the specified array to the specified collection.
- **newSetFromMap(Map)** - Creates a general purpose `Set` implementation from a general purpose `Map` implementation.
- **asLifoQueue(Deque)** - Returns a view of a `Deque` as a Last-in-first-out (Lifo) `Queue`.

- **Infrastructure**
  - **Iterators** - Similar to the familiar Enumeration interface, but more powerful, and with improved method names.
    - **Iterator** - In addition to the functionality of the `Enumeration` interface, allows the user to remove elements from the backing collection with well defined, useful semantics.
    - **ListIterator** - Iterator for use with lists. In addition to the functionality of the `Iterator` interface, supports bi-directional iteration, element replacement, element insertion and index retrieval.
  - **Ordering**
    - **Comparable** - Imparts a *natural ordering* to classes that implement it. The natural ordering may be used to sort a list or maintain order in a sorted set or map. Many classes have been retrofitted to implement this interface.
    - **Comparator** - Represents an order relation, which may be used to sort a list or maintain order in a sorted set or map. Can override a type's natural ordering, or order objects of a type that does not implement the `Comparable` interface.
  - **Runtime Exceptions**
    - **UnsupportedOperationException** - Thrown by collections if an unsupported optional operation is called.
    - **ConcurrentModificationException** - Thrown by iterators and list iterators if the backing collection is modified unexpectedly while the iteration is in progress. Also thrown by *sublist* views of lists if the backing list is modified unexpectedly.
  - **Performance**
    - **RandomAccess** - Marker interface that allows `List` implementations to indicate that they support fast (generally constant time) random access. This allows generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.
- **Array Utilities**
  - **Arrays** - Contains static methods to sort, search, compare, hash, copy, resize, convert to `String`, and fill arrays of primitives and Objects.