Iterator Types

fail-fast

- collections maintain an internal counter called modCount.
  - each time an item is added or removed from the Collection, counter gets incremented.
  - when iterating, on each next() call, the current value of modCount gets compared with the initial value.
  - if there's a mismatch, it throws ConcurrentModificationException which aborts the entire operation.
  - however, this check is done without synchronization, so there is a risk of seeing a stale value of the modification count and therefore that the iterator does not realize a modification has been made.
  - this was a deliberate design tradeoff to reduce the performance impact of the concurrent modification detection code
- ConcurrentModificationException can arise in single-threaded code as well; this happens when objects are removed from the collection directly.
- if during iteration over a Collection, an item is removed using Iterator's remove() method, that's entirely safe and doesn't throw an exception.
- example: Default iterators on Collections from java.util package such as ArrayList, HashMap, etc.

weakly consistent

- reflects some but not necessarily all of the changes that have been made to their backing collection since they were created.
  - e.g. , if elements in the collection have been modified or removed before the iterator reaches them, it definitely will reflect these changes, but no such guarantee is made for insertions.
- the default iterator for the ConcurrentHashMap is weakly consistent.
  - this means that this Iterator can tolerate concurrent modification, traverses elements as they existed when Iterator was constructed and may (but isn't guaranteed to) reflect modifications to the Collection after the construction of the Iterator.
- example: Default iterators on Collections from java.util.concurrent package such as ConcurrentHashMap, ConcurrentSkipListSet etc.

fail-safe

- these iterators create a clone of the actual Collection and iterate over it. If any modification happens after the iterator is created, the copy still remains untouched. Hence, these Iterators continue looping over the Collection even if it's modified.
- disadvantage is the overhead of creating a copy of the Collection, both regarding time and memory.
- example: CopyOnWriteArrayList, CopyOnWriteArraySet

interface | Iterable<T>

- implementing this interface allows an object to be the target of the "for-each loop" statement

interface | Collection<E>

- the root interface in the collection hierarchy
- a collection represents a group of objects, known as its elements
  - some collections allow duplicate elements and others do not.
  - some are ordered and others unordered.
  - some implementations prohibit null elements, and
  - some have restrictions on the types of their elements
- the JDK does not provide any direct implementations of this interface
  - it provides implementations of more specific subinterfaces like Set and List.
  - this interface is typically used to pass collections around and manipulate them where maximum generality is desired.
  - bags or multisets (unordered collections that may contain duplicate elements) should implement this interface directly.
- all general-purpose Collection implementation classes should provide two "standard" constructors:
  - a void (no arguments) constructor, which creates an empty collection, and
  - a constructor with a single argument of type Collection,
    - which creates a new collection with the same elements as its argument.
    - in effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type.
- many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the contains(Object o) method says: "returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e))
  - thus wherever applicabale impleimentation of equals, hashCode, Comparable & Comparator should be consistent with each other.
- some collection operations which perform recursive traversal of the collection may fail with an exception for self-referential instances where the collection directly or indirectly contains itself. This includes the clone(), equals(), hashCode() and toString() methods.

interface | Queue<E>

- a collection designed for holding elements prior to processing
- queues offer 2 flavor of methods for basic operations
  - one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation)
  - special value returning methods are designed specifically for use with capacity-restricted Queue implementations
  -

	Throws exception	Returns special value
Insert	*add(e)*	*offer(e)*
Remove	*remove()*	*poll()*
Examine	*element()*	*peek()*

- 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,
  - and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out)
  - whatever the ordering used, the head of the queue is that element which 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 Queue implementation must specify its ordering properties.
- queue implementations generally do not allow insertion of null elements,
  - although some implementations, such as LinkedList, do not prohibit insertion of null
  - even in the implementations that permit it, null should not be inserted into a Queue, as null is also used as a special return value by the poll method to indicate that the queue contains no elements

Ordered

interface | List<E>

- an ordered collection (also known as a sequence)
- it allows index based operations for read, write, deletion of elements
  - index based operations may execute in time proportional to the index value for some implementations (the LinkedList class, for example).
  - thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.
- unlike sets, lists typically allow duplicate elements.
- lists (like Java arrays) are zero based
- the List interface provides a special iterator, called a ListIterator
  - it allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides.

Ordered

interface | RandomAccess

- marker interface used by List implementations to indicate that they support fast (generally constant time) random access.
- the primary purpose of this interface is to allow generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

interface | Deque<E>

- deque aka "double ended queue" pronounced as "deck" supports element insertion and removal at both ends
- this interface defines methods to access the elements at both ends of the deque
- basic operations are offered in 2 flavor of methods
  - one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation)
  - special value returning methods are designed specifically for use with capacity-restricted deque implementations
  -

	First Element (Head)		Last Element (Tail)	
	Throws exception	Returns special value	Throws exception	Returns special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

- deques can also be used as LIFO (Last-In-First-Out) stacks.
  - stack methods for deque | push(e) | pop() | peek()
  - This interface should be used in preference to the legacy Stack class
- unlike the List interface, this interface does not provide support for indexed access to elements
- while null elements are not prohibited for most implementations, its strongly recommended not to do so

Ordered

class | LinkedList<E>

- Doubly-linked list implementation of the List and Deque interfaces
- Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.
- time complexity | Insertion O(1), Deletion O(1), Search O(n), Indexing O(n)
- iterator and listIterator methods are fail-fast

ThreadSafe | Ordered | Null Element

class | ArrayList<E>

- resizable-array implementation of the List interface
- 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.
- each ArrayList instance has a capacity.
  - the capacity is the size of the array used to store the elements in the list.
  - as elements are added to an ArrayList, its capacity grows automatically.
  - an application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.
- iterator and listIterator methods are fail-fast

ThreadSafe | Ordered | Null Element

class | CopyOnWriteArrayList<E>

- resizable-array implementation of the List interface
- all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array
- useful in scenario when traversal operations vastly outnumber mutations
- the "snapshot" style iterator uses a reference to the state of the array at the point that the iterator was created
  - iterator will not reflect additions, removals, or changes to the list since the iterator was created.
  - element-changing operations on iterators (remove, set, and add) are not supported

ThreadSafe | Ordered | Null Element

class | Vector<E>

- resizable-array implementation of the List interface
- as of the Java 2, this class was retrofitted to Collections Framework.
  - if a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector
  - performance time cost is inferior to ArrayList due to synchronization overhead
- Vector have 3 space parameters
  - size is the number of elements in this vector
  - capacity is size of internal array
  - capacityIncrement is the amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
    - default is double of size
  - an application can increase the capacity before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.
- iterator and listIterator methods are fail-fast
- Enumerations are not fail-fast

ThreadSafe | Ordered | Null Element

class | Stack<E>

- the Stack class represents a last-in-first-out (LIFO) stack of objects
- it extends class Vector with five operations that allow a vector to be treated as a stack
  - O(1) | push(E item)
  - O(1) | pop()
  - O(1) | peek()
  - O(1) | empty()
  - O(n) | search(Object o)
- a more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class

ThreadSafe | Ordered | Null Element

class | ArrayDeque<E>

- resizable-array implementation of the Deque interface
- this class is likely to be faster than Stack and LinkedList
- iterator is fail-fast
- performance time cost

enqueuing and dequeuing methods	`offer(E e) | add(E e)` `poll() | remove()`	O(1) amortized constant time
retrieval methods	`peek() | element()`	O(1)

specific object targeted operations methods	`contains(Object), remove(Object)`
	`removeFirstOccurrence(Object o)`
	`removeLastOccurrence(Object o)`
		O(n)

	`size()`	O(1)
bulk operations	`addAll, containsAll, removeAll, etc.`	O(n)

ThreadSafe | Ordered \| FIFO/LIFO | Null Element | Unbounded

class | PriorityQueue<E>

- queue based on a priority heap
- the elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time
- the head of this queue is the least element with respect to the specified ordering
- the queue retrieval operations poll, remove, peek, and element access the element at the head of the queue
- a priority queue is unbounded, but its internal dynamic array has capacity and expands as necessary
- performance time cost

enqueuing and dequeuing methods	`offer(E e) | add(E e)` `poll() | remove()`	O(log(n))
retrieval methods	`peek() | element()` `size()`	O(1)
specific object targeted operations methods	`remove(Object)` `contains(Object)`	O(n)

- iterator do not traverse elements in priority based order

ThreadSafe | Ordered | Null Element | Unbounded

class | ConcurrentLinkedDeque<E>

- a deque based on linked nodes | underlying data structure is doubly linked list
- thread safety is guaranteed by optimistic locking in the form of non-blocking compare-and-set (CAS) operations on separate VarHandles for the head and tail of the deque and the list node references
- iterators and spliterators are weakly consistent
- because of the asynchronous nature of these deques
  - the size method is NOT a constant-time operation
  - bulk operations are not guaranteed to be performed atomically
- performance time cost

enqueuing and dequeuing methods	`offer(E e) | add(E e)` `poll() | remove()`	O(1)
retrieval methods	`peek() | element()`	O(1)

specific object targeted operations methods	`contains(Object), remove(Object)`
	`removeFirstOccurrence(Object o)`
	`removeLastOccurrence(Object o)`
		O(n)

	`size()`	O(n)
bulk operations	`addAll, containsAll, removeAll, etc.`	O(n)

ThreadSafe | Ordered \| FIFO/LIFO | Null Element | Unbounded

class | ConcurrentLinkedQueue<E>

- queue based on linked nodes
- this queue orders elements FIFO (first-in-first-out)
  - the *head* of the queue is that element that has been on the queue the longest time
  - the tail of the queue is that element that has been on the queue the shortest time
  - new elements are inserted at the tail of the queue,
  - and the queue retrieval operations obtain elements at the head of the queue
- this implementation is based on non-blocking algorithm
  - thread-safety is achieved by optimistic locking. More precisely: by non-blocking compare-and-set (CAS) operations on separate VarHandles for the queue's head and tail
- iterators are weakly consistent
- because of the asynchronous nature of these queues
  - unlike in most collections, the size method is NOT a constant-time operation
  - additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically
- performance time cost

enqueuing and dequeuing methods	`offer(E e) | add(E e)` `poll() | remove()`	O(1)
retrieval methods	`peek() | element()`	O(1)
specific object targeted operations methods	`remove(Object)` `contains(Object)`	O(n)
	`size()`	O(n)

ThreadSafe | Ordered \| FIFO | Null Element | Unbounded