Java Collection Framework Table

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Class/Interface | Allow Duplicates | Maintain Insertion Order | Order by Priority | Blocking Queue | Methods for Sorted Sets | Constant-Time Positional Access | Concurrent Read/Write Operations | Backed by a Hash Table | Resizable | Null Allowed | Synchronized | LIFO | FIFO | Thread-Safe | Initial Capacity and Load Factor Supported | Read Element Repeatedly Supported |
| ArrayList | Yes | No | No | No | No | Yes | No | No | Yes | Yes | No | No | No | No | Initial capacity, no load factor | Yes |
| LinkedList | Yes | Yes | No | No | No | No | No | No | No | Yes | No | Yes | Yes | No | No (doubly linked list, no capacity setting) | Yes |
| Vector | Yes | Yes | No | No | No | Yes | No | No | Yes | Yes | Yes | No | No | Yes | Initial capacity, no load factor | Yes |
| Stack | Yes | Yes | No | No | No | Yes | No | No | Yes | Yes | Yes | Yes | No | Yes | Inherits from `Vector`, initial capacity | Yes |
| CopyOnWriteArrayList | Yes | Yes | No | No | No | Yes | Yes | No | Yes | Yes | No | No | No | Yes | No initial capacity, no load factor | Yes |
| HashSet | No | No | No | No | No | No | No | Yes | No | Yes | No | No | No | No | Initial capacity, load factor supported | Yes |
| LinkedHashSet | No | Yes | No | No | No | No | No | Yes | No | Yes | No | No | No | No | Initial capacity, load factor supported | Yes |
| EnumSet | No | No | No | No | No | No | No | No | No | No | No | No | No | No | No (specific to enum type values) | Yes |
| CopyOnWriteArraySet | No | Yes | No | No | No | No | Yes | No | Yes | Yes | No | No | No | Yes | No initial capacity, no load factor | Yes |
| TreeSet | No | No | No | No | Yes | No | No | No | No | No | No | No | No | No | No initial capacity, no load factor | Yes |
| PriorityQueue | Yes | No | Yes | No | No | No | No | No | No | Yes | No | No | Yes | No | Initial capacity, no load factor | Yes |
| LinkedBlockingQueue | Yes | Yes | No | Yes | No | No | No | No | No | No | No | No | Yes | Yes | Initial capacity, no load factor | Yes |
| PriorityBlockingQueue | Yes | No | Yes | Yes | No | No | No | No | No | No | No | No | Yes | Yes | Initial capacity, no load factor | Yes |
| ConcurrentLinkedQueue | Yes | No | No | No | No | No | Yes | No | No | No | No | No | Yes | Yes | No initial capacity, no load factor | Yes |
| LinkedTransferQueue | Yes | No | No | Yes | No | No | Yes | No | Yes | No | No | No | Yes | Yes | No initial capacity, no load factor | Yes |
| ArrayDeque | Yes | Yes | No | No | No | No | No | No | Yes | Yes | No | Yes | No | No | No initial capacity, no load factor | Yes |
| ConcurrentHashMap | No | No | No | No | No | No | Yes | Yes | No | No | No | No | No | Yes | Initial capacity, load factor supported | Yes |
| ConcurrentLinkedDeque | Yes | Yes | No | No | No | No | Yes | No | No | No | No | Yes | Yes | Yes | No initial capacity, no load factor | Yes |
| LinkedBlockingDeque | Yes | Yes | No | Yes | No | No | Yes | No | Yes | No | Yes | Yes | Yes | Yes | Initial capacity, no load factor | Yes |
| Collections | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Arrays | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

### Java Collection Framework Hierarchy

1. **Iterable (Interface)**

A computer screen shot of a diagram

Description automatically generated

1. **Map (Interface)**

A diagram of a map

Description automatically generated

1. **Utility Classes**
   * **Collections**
   * **Arrays**

Here is the complete merged version of the hierarchy and a deep dive into the Collection framework elements:

### 1. Iterable ****(Interface)****

* The root interface of the collection hierarchy, representing a collection that can be iterated over.
* Key Methods:
  + iterator()
  + forEach()
  + spliterator()

### 2. Collection ****(Interface)****

* Represents a group of objects (elements) with basic methods for adding, removing, and querying elements.
* Subinterfaces: List, Set, Queue, Deque, etc.
* Key Methods:
  + add(), remove(), size(), contains()

### 3. List ****(Interface)****

* Ordered collection allowing duplicates, maintains insertion order.
* Implementing Classes:
  + **ArrayList**: Resizable array.
  + **LinkedList**: Doubly linked list.
  + **Vector**: Synchronized resizable array.
  + **Stack**: LIFO stack, extends Vector.
  + **CopyOnWriteArrayList**: Thread-safe list with copy-on-write.

### 4. Set ****(Interface)****

* Unordered collection without duplicate elements.
* Implementing Classes:
  + **HashSet**: Unordered, backed by a hash table.
  + **LinkedHashSet**: Maintains insertion order.
  + **EnumSet**: High-performance set for enum types.



* + **CopyOnWriteArraySet**: Thread-safe with copy-on-write.

### 5. SortedSet ****(Interface)**** – Extends Set

* Maintains elements in ascending order.
* Subinterfaces:
  + **NavigableSet (Interface)**: Provides navigation methods (lower(), higher(), etc.).
* Implementing Classes:
  + **TreeSet**: Sorted set using a Red-Black tree.



### 6. Queue ****(Interface)****

* FIFO collection for holding elements before processing.
* Implementing Classes:
  + **PriorityQueue**: Ordered by priority.
  + **LinkedBlockingQueue**: Thread-safe blocking queue.
  + **PriorityBlockingQueue**: Thread-safe priority queue.
  + **ConcurrentLinkedQueue**: Thread-safe, non-blocking queue.
  + **LinkedTransferQueue**: Efficient, non-blocking, supports "transfer" operations.

### 7. Deque ****(Interface)**** – Extends Queue

* Double-ended queue allowing element insertion/removal from both ends.
* Implementing Classes:
  + **ArrayDeque**: Resizable array-based deque.
  + **LinkedList**: Implements both Deque and List.
  + **ConcurrentLinkedDeque**: Thread-safe deque.
  + **LinkedBlockingDeque**: Thread-safe blocking deque.
  + **TransferQueue (Interface)**: Extends BlockingQueue.

### 8. NavigableSet ****(Interface)**** – Extends SortedSet

* Provides navigation methods for sorted sets.
* Implementing Classes:
  + **TreeSet**: Red-Black tree-based sorted set.

### 9. SortedMap ****(Interface)****

* Maintains mappings in ascending key order.
* Implementing Classes:
  + **TreeMap**: Red-Black tree-based map.

### 10. TransferQueue ****(Interface)**** – Extends BlockingQueue

* Allows producers to wait for consumers to receive elements.
* Implementing Classes:
  + **LinkedTransferQueue**: Efficient, thread-safe implementation.

### 11. ArrayDeque ****(Class)**** – Implements Deque

* Resizable array-based double-ended queue, efficient for both ends.

### 12. ArrayList ****(Class)****

* Resizable array-based List, efficient for constant-time positional access.

### 13. ConcurrentHashMap ****(Class)****

* Thread-safe version of HashMap, allows concurrent read/write operations.

### 14. ConcurrentLinkedDeque ****(Class)****

* Thread-safe, non-blocking deque suitable for concurrent environments.

### 15. ConcurrentLinkedQueue ****(Class)****

* Thread-safe, non-blocking queue suitable for high-concurrency environments.

### 16. CopyOnWriteArrayList ****(Class)****

* Thread-safe list where all write operations result in a new copy of the array.

### 17. CopyOnWriteArraySet ****(Class)****

* Thread-safe set that internally uses a CopyOnWriteArrayList.

### 18. EnumSet ****(Class)****

* High-performance Set implementation for enum types.

### 19. HashSet ****(Class)****

* Unordered set, backed by a HashMap.

### 20. LinkedBlockingDeque ****(Class)****

* Thread-safe deque with optional capacity, blocks if full/empty.

### 21. LinkedBlockingQueue ****(Class)****

* Thread-safe queue, blocks when full or empty.

### 22. LinkedHashSet ****(Class)****

* Set that maintains insertion order, backed by a hash table with a linked list.

### 23. LinkedList ****(Class)****

* Doubly linked list implementation of both List and Deque.

### 24. LinkedTransferQueue ****(Class)****

* Highly concurrent, non-blocking TransferQueue.

### 25. PriorityBlockingQueue ****(Class)****

* Thread-safe implementation of a priority queue.

### 26. PriorityQueue ****(Class)****

* A queue ordered by priority (natural or custom comparator).

### 27. Stack ****(Class)****

* Legacy LIFO stack implementation based on Vector.

### 28. TreeSet ****(Class)****

* Sorted set implementation using a Red-Black tree.

### 29. Vector ****(Class)****

* Legacy synchronized resizable array.

### 30. Collections ****(Utility Class)****

* Utility methods for collection operations such as sorting, searching, and making collections unmodifiable.

### 31. Arrays ****(Utility Class)****

* Utility methods for working with arrays, such as sorting, converting to lists, and copying arrays.