**ArrayList**

In Java, ArrayList is a part of the java.util package and provides a dynamic array that can grow or shrink in size. Here are all the main built-in functions (methods) of ArrayList:

**1. Basic Methods**

* add(E e): Appends the specified element to the end of this list.
* add(int index, E element): Inserts the specified element at the specified position in this list.
* remove(int index): Removes the element at the specified position in this list.
* remove(Object o): Removes the first occurrence of the specified element from this list.
* clear(): Removes all of the elements from this list.
* get(int index): Returns the element at the specified position in this list.
* set(int index, E element): Replaces the element at the specified position in this list with the specified element.
* size(): Returns the number of elements in this list.
* isEmpty(): Returns true if this list contains no elements.
* contains(Object o): Returns true if this list contains the specified element.
* indexOf(Object o): Returns the index of the first occurrence of the specified element, or -1 if this list does not contain the element.
* lastIndexOf(Object o): Returns the index of the last occurrence of the specified element, or -1 if this list does not contain the element.
* toArray(): Returns an array containing all of the elements in this list in proper sequence.
* toArray(T[] a): Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

**2. Iteration Methods**

* iterator(): Returns an iterator over the elements in this list in proper sequence.
* listIterator(): Returns a list iterator over the elements in this list.
* listIterator(int index): Returns a list iterator of the elements in this list, starting at the specified position in the list.
* forEach(Consumer<? super E> action): Performs the given action for each element of the list.

**3. Bulk Operations**

* addAll(Collection<? extends E> c): Appends all of the elements in the specified collection to the end of this list.
* addAll(int index, Collection<? extends E> c): Inserts all of the elements in the specified collection into this list at the specified position.
* removeAll(Collection<?> c): Removes from this list all of its elements that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this list that are contained in the specified collection.
* replaceAll(UnaryOperator<E> operator): Replaces each element of this list with the result of applying the operator to that element.
* sort(Comparator<? super E> c): Sorts this list according to the order induced by the specified Comparator.

**LinkedList**

In Java, LinkedList is a part of the java.util package and implements both the List and Deque interfaces, providing a doubly-linked list. Here are the main built-in functions (methods) of the LinkedList class:

**1. Basic Methods**

* add(E e): Appends the specified element to the end of this list.
* add(int index, E element): Inserts the specified element at the specified position in this list.
* remove(int index): Removes the element at the specified position in this list.
* remove(Object o): Removes the first occurrence of the specified element from this list.
* clear(): Removes all of the elements from this list.
* get(int index): Returns the element at the specified position in this list.
* set(int index, E element): Replaces the element at the specified position in this list with the specified element.
* size(): Returns the number of elements in this list.
* isEmpty(): Returns true if this list contains no elements.
* contains(Object o): Returns true if this list contains the specified element.
* indexOf(Object o): Returns the index of the first occurrence of the specified element, or -1 if this list does not contain the element.
* lastIndexOf(Object o): Returns the index of the last occurrence of the specified element, or -1 if this list does not contain the element.
* toArray(): Returns an array containing all of the elements in this list in proper sequence.
* toArray(T[] a): Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

**2. Iteration Methods**

* iterator(): Returns an iterator over the elements in this list in proper sequence.
* descendingIterator(): Returns an iterator over the elements in this list in reverse sequential order.
* listIterator(): Returns a list iterator over the elements in this list.
* listIterator(int index): Returns a list iterator of the elements in this list, starting at the specified position in the list.
* forEach(Consumer<? super E> action): Performs the given action for each element of the list.

**3. Bulk Operations**

* addAll(Collection<? extends E> c): Appends all of the elements in the specified collection to the end of this list.
* addAll(int index, Collection<? extends E> c): Inserts all of the elements in the specified collection into this list at the specified position.
* removeAll(Collection<?> c): Removes from this list all of its elements that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this list that are contained in the specified collection.
* replaceAll(UnaryOperator<E> operator): Replaces each element of this list with the result of applying the operator to that element.
* sort(Comparator<? super E> c): Sorts this list according to the order induced by the specified Comparator.

**Vector**

In Java, Vector is a part of the java.util package and is a legacy class that implements the List interface, providing a dynamic array that is synchronized. Here are the main built-in functions (methods) of the Vector class:

**1. Basic Methods**

* add(E e): Appends the specified element to the end of this vector.
* add(int index, E element): Inserts the specified element at the specified position in this vector.
* remove(int index): Removes the element at the specified position in this vector.
* remove(Object o): Removes the first occurrence of the specified element from this vector.
* clear(): Removes all of the elements from this vector.
* get(int index): Returns the element at the specified position in this vector.
* set(int index, E element): Replaces the element at the specified position in this vector with the specified element.
* size(): Returns the number of elements in this vector.
* isEmpty(): Returns true if this vector contains no elements.
* contains(Object o): Returns true if this vector contains the specified element.
* indexOf(Object o): Returns the index of the first occurrence of the specified element, or -1 if this vector does not contain the element.
* indexOf(Object o, int index): Returns the index of the first occurrence of the specified element, starting the search at the specified index.
* lastIndexOf(Object o): Returns the index of the last occurrence of the specified element, or -1 if this vector does not contain the element.
* lastIndexOf(Object o, int index): Returns the index of the last occurrence of the specified element, searching backwards starting at the specified index.
* toArray(): Returns an array containing all of the elements in this vector in proper sequence.
* toArray(T[] a): Returns an array containing all of the elements in this vector in proper sequence; the runtime type of the returned array is that of the specified array.

**2. Capacity and Size Methods**

* capacity(): Returns the current capacity of this vector.
* ensureCapacity(int minCapacity): Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
* trimToSize(): Trims the capacity of this vector to be the vector’s current size.
* setSize(int newSize): Sets the size of this vector.

**3. Iteration Methods**

* iterator(): Returns an iterator over the elements in this vector in proper sequence.
* listIterator(): Returns a list iterator over the elements in this vector.
* listIterator(int index): Returns a list iterator of the elements in this vector, starting at the specified position in the list.
* forEach(Consumer<? super E> action): Performs the given action for each element of the vector.

**4. Bulk Operations**

* addAll(Collection<? extends E> c): Appends all of the elements in the specified collection to the end of this vector.
* addAll(int index, Collection<? extends E> c): Inserts all of the elements in the specified collection into this vector at the specified position.
* removeAll(Collection<?> c): Removes from this vector all of its elements that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this vector that are contained in the specified collection.
* replaceAll(UnaryOperator<E> operator): Replaces each element of this vector with the result of applying the operator to that element.
* sort(Comparator<? super E> c): Sorts this vector according to the order induced by the specified comparator.

**Stack**

In Java, Stack is a part of the java.util package and extends Vector, implementing a last-in, first-out (LIFO) stack. Here are the main built-in functions (methods) of the Stack class:

**1. Basic Stack Methods**

* push(E item): Pushes an item onto the top of the stack.
* pop(): Removes and returns the object at the top of this stack.
* peek(): Looks at the object at the top of this stack without removing it.
* empty(): Tests if this stack is empty.
* search(Object o): Returns the 1-based position where an object is located on the stack (returns -1 if not found).

**2. Inherited Methods from Vector**

Since Stack extends Vector, it inherits many of Vector's methods. Here are the relevant ones:

* **Add and Remove**
  + add(E e): Appends the specified element to the end of the vector.
  + add(int index, E element): Inserts the specified element at the specified position in the vector.
  + remove(int index): Removes the element at the specified position in the vector.
  + remove(Object o): Removes the first occurrence of the specified element from the vector.
  + clear(): Removes all elements from the vector.
* **Access and Search**
  + get(int index): Returns the element at the specified position in the vector.
  + set(int index, E element): Replaces the element at the specified position in the vector.
  + size(): Returns the number of elements in the vector.
  + isEmpty(): Returns true if the vector contains no elements.
  + contains(Object o): Returns true if the vector contains the specified element.
  + indexOf(Object o): Returns the index of the first occurrence of the specified element.
  + lastIndexOf(Object o): Returns the index of the last occurrence of the specified element.
* **Capacity Management (Inherited from Vector)**
  + capacity(): Returns the current capacity of the vector.
  + ensureCapacity(int minCapacity): Increases the vector's capacity if necessary.
  + trimToSize(): Trims the vector's capacity to match its current size.
* **Array Operations**
  + toArray(): Returns an array containing all of the elements in the vector in proper sequence.
  + toArray(T[] a): Returns an array containing all of the elements in the vector in proper sequence.
* **Bulk Operations**
  + addAll(Collection<? extends E> c): Appends all of the elements in the specified collection to the end of the vector.
  + addAll(int index, Collection<? extends E> c): Inserts all of the elements in the specified collection into the vector.
  + removeAll(Collection<?> c): Removes from the vector all of its elements that are contained in the specified collection.
  + retainAll(Collection<?> c): Retains only the elements in the vector that are contained in the specified collection.
* **Iteration**
  + iterator(): Returns an iterator over the elements in the vector in proper sequence.
  + listIterator(): Returns a list iterator over the elements in the vector.
  + listIterator(int index): Returns a list iterator of the elements in the vector, starting at the specified position in the list.
  + forEach(Consumer<? super E> action): Performs the given action for each element in the vector.
* **Sorting**
  + sort(Comparator<? super E> c): Sorts the vector according to the order induced by the specified comparator.
  + replaceAll(UnaryOperator<E> operator): Replaces each element in the vector with the result of applying the given operator.

**PriorityQueue**

In Java, PriorityQueue is part of the java.util package and implements a priority queue, where elements are ordered based on their natural ordering or by a Comparator provided at queue construction. It does not permit null elements. Here are the main built-in functions (methods) of the PriorityQueue class:

**1. Basic Methods**

* add(E e): Inserts the specified element into this priority queue. Throws an exception if the element cannot be added.
* offer(E e): Inserts the specified element into this priority queue. Returns true if successful, or false if not.
* remove(Object o): Removes a single instance of the specified element from this priority queue, if it is present.
* poll(): Retrieves and removes the head of this queue, or returns null if this queue is empty.
* peek(): Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
* clear(): Removes all of the elements from this priority queue.

**2. Capacity and Size**

* size(): Returns the number of elements in this queue.
* isEmpty(): Returns true if this queue contains no elements.

**3. Element Access**

* contains(Object o): Returns true if this queue contains the specified element.
* remove(): Removes the head of this queue, throwing an exception if the queue is empty.
* element(): Retrieves, but does not remove, the head of this queue, throwing an exception if this queue is empty.

**4. Iteration**

* iterator(): Returns an iterator over the elements in this priority queue. The iterator does not traverse the elements in any particular order.
* forEach(Consumer<? super E> action): Performs the given action for each element of the queue.

**5. Bulk Operations**

* addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to this queue.
* removeAll(Collection<?> c): Removes all elements in the specified collection from this queue.
* retainAll(Collection<?> c): Retains only the elements in this queue that are contained in the specified collection.

**6. Array Operations**

* toArray(): Returns an array containing all of the elements in this queue in proper sequence.
* toArray(T[] a): Returns an array containing all of the elements in this queue in proper sequence; the runtime type of the returned array is that of the specified array.

**ArrayDeque**

In Java, ArrayDeque is a part of the java.util package and implements a resizable-array double-ended queue (deque). It supports both stack and queue operations, and is often preferred for queue implementations due to its efficiency. It does not allow null elements. Here are the main built-in functions (methods) of the ArrayDeque class:

**1. Basic Deque Methods (Double-Ended Queue Operations)**

* addFirst(E e): Inserts the specified element at the front of this deque.
* addLast(E e): Inserts the specified element at the end of this deque.
* offerFirst(E e): Inserts the specified element at the front of this deque. Returns true if successful, false otherwise.
* offerLast(E e): Inserts the specified element at the end of this deque. Returns true if successful, false otherwise.
* removeFirst(): Retrieves and removes the first element of this deque. Throws an exception if the deque is empty.
* removeLast(): Retrieves and removes the last element of this deque. Throws an exception if the deque is empty.
* pollFirst(): Retrieves and removes the first element of this deque, or returns null if the deque is empty.
* pollLast(): Retrieves and removes the last element of this deque, or returns null if the deque is empty.
* getFirst(): Retrieves, but does not remove, the first element of this deque. Throws an exception if the deque is empty.
* getLast(): Retrieves, but does not remove, the last element of this deque. Throws an exception if the deque is empty.
* peekFirst(): Retrieves, but does not remove, the first element of this deque, or returns null if the deque is empty.
* peekLast(): Retrieves, but does not remove, the last element of this deque, or returns null if the deque is empty.

**2. Queue Methods**

* add(E e): Inserts the specified element at the end of this deque. Throws an exception if unable to add the element.
* offer(E e): Inserts the specified element at the end of this deque. Returns true if successful, false otherwise.
* remove(): Retrieves and removes the head of this deque. Throws an exception if the deque is empty.
* poll(): Retrieves and removes the head of this deque, or returns null if the deque is empty.
* element(): Retrieves, but does not remove, the head of this deque. Throws an exception if the deque is empty.
* peek(): Retrieves, but does not remove, the head of this deque, or returns null if the deque is empty.

**3. Stack Methods**

* push(E e): Pushes an element onto the stack represented by this deque (adds it to the front).
* pop(): Pops an element from the stack represented by this deque (removes the first element). Throws an exception if the deque is empty.
* peek(): Retrieves, but does not remove, the head element of this stack (equivalent to peekFirst()).

**4. Bulk Operations**

* addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to this deque.
* removeAll(Collection<?> c): Removes all of the elements in this deque that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this deque that are contained in the specified collection.

**5. Size and Capacity**

* size(): Returns the number of elements in this deque.
* isEmpty(): Returns true if this deque contains no elements.

**6. Element Access and Removal**

* contains(Object o): Returns true if this deque contains the specified element.
* remove(Object o): Removes a single instance of the specified element from this deque, if it is present.
* clear(): Removes all elements from this deque.

**7. Iteration**

* iterator(): Returns an iterator over the elements in this deque in proper sequence (from first to last).
* descendingIterator(): Returns an iterator over the elements in this deque in reverse sequential order (from last to first).
* forEach(Consumer<? super E> action): Performs the given action for each element in this deque.

**8. Array Operations**

* toArray(): Returns an array containing all of the elements in this deque in proper sequence.
* toArray(T[] a): Returns an array containing all of the elements in this deque in proper sequence; the runtime type of the returned array is that of the specified array.

**11. Comparator**

* comparator(): Returns null as ArrayDeque elements are not sorted by default (does not support a comparator).

**12. Miscellaneous**

* clone(): Returns a shallow copy of this deque.
* removeIf(Predicate<? super E> filter): Removes all of the elements for which the predicate returns true.

**13. Limitations**

* **No Indexed Access**: Unlike ArrayList, ArrayDeque does not provide random access by index (e.g., get(int index) or set(int index, E element)).
* **No Null Elements**: ArrayDeque does not allow null elements.

**HashSet**

In Java, HashSet is a part of the java.util package and implements the Set interface. It is backed by a hash table and does not allow duplicate elements. It does not maintain any order of its elements. Here are the main built-in functions (methods) of the HashSet class:

**1. Basic Set Methods**

* add(E e): Adds the specified element to this set if it is not already present.
* remove(Object o): Removes the specified element from this set if it is present.
* contains(Object o): Returns true if this set contains the specified element.
* clear(): Removes all of the elements from this set.
* size(): Returns the number of elements in this set.
* isEmpty(): Returns true if this set contains no elements.

**2. Bulk Operations**

* addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to this set.
* removeAll(Collection<?> c): Removes all of the elements in this set that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this set that are contained in the specified collection.
* containsAll(Collection<?> c): Returns true if this set contains all of the elements in the specified collection.

**3. Iteration**

* iterator(): Returns an iterator over the elements in this set.
* forEach(Consumer<? super E> action): Performs the given action for each element of this set.

**4. Array Operations**

* toArray(): Returns an array containing all of the elements in this set.
* toArray(T[] a): Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

**5. Comparison and Hashing**

* equals(Object o): Compares the specified object with this set for equality.
* hashCode(): Returns the hash code value for this set.

**8. Miscellaneous**

* clone(): Returns a shallow copy of this HashSet.
* removeIf(Predicate<? super E> filter): Removes all elements of this set that satisfy the given predicate.

**9. Limitations**

* **No Order**: HashSet does not guarantee any specific order of its elements. If you need order, consider using LinkedHashSet or TreeSet.
* **No Duplicate Elements**: It automatically prevents duplicate entries. Attempting to add a duplicate element has no effect on the set.

The HashSet class is suitable for applications where the main requirement is to prevent duplicate elements and where the order of elements is not important. It is generally faster than TreeSet for basic operations due to its constant-time performance for the basic operations (add, remove, contains, and size), assuming the hash function disperses elements properly among the buckets.

**LinkedHashSet**

In Java, LinkedHashSet is a part of the java.util package and is an implementation of the Set interface. It extends HashSet and maintains a linked list of the entries in the set, which provides predictable iteration order based on insertion order. Here are the main built-in functions (methods) of the LinkedHashSet class:

### 1. ****Basic Set Methods****

* add(E e): Adds the specified element to this set if it is not already present.
* remove(Object o): Removes the specified element from this set if it is present.
* contains(Object o): Returns true if this set contains the specified element.
* clear(): Removes all of the elements from this set.
* size(): Returns the number of elements in this set.
* isEmpty(): Returns true if this set contains no elements.

### 2. ****Bulk Operations****

* addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to this set.
* removeAll(Collection<?> c): Removes all of the elements in this set that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this set that are contained in the specified collection.
* containsAll(Collection<?> c): Returns true if this set contains all of the elements in the specified collection.

### 3. ****Iteration****

* iterator(): Returns an iterator over the elements in this set, in insertion order.
* forEach(Consumer<? super E> action): Performs the given action for each element of this set.

### 4. ****Array Operations****

* toArray(): Returns an array containing all of the elements in this set, in insertion order.
* toArray(T[] a): Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

### 5. ****Comparison and Hashing****

* equals(Object o): Compares the specified object with this set for equality.
* hashCode(): Returns the hash code value for this set.

### 8. ****Miscellaneous****

* clone(): Returns a shallow copy of this LinkedHashSet.
* removeIf(Predicate<? super E> filter): Removes all elements of this set that satisfy the given predicate.

### 9. ****Order and Performance****

* **Insertion Order**: Maintains the order of elements based on their insertion order.
* **Performance**: Combines the hash table and linked list, giving you the benefits of fast lookups and predictable iteration order.

### 10. ****Limitations****

* **No Sorted Order**: Unlike TreeSet, LinkedHashSet does not sort elements; it maintains insertion order.
* **No Null Elements**: Although LinkedHashSet allows null elements, it's good practice to handle nulls carefully.

The LinkedHashSet class is useful when you need a set with predictable iteration order that is based on the order in which elements were inserted, while still providing constant-time performance for basic operations.

**TreeSet**

In Java, TreeSet is a part of the java.util package and implements the SortedSet interface, which extends Set. It is a NavigableSet that uses a Red-Black tree for storage, which provides a sorted order of elements. Here are the main built-in functions (methods) of the TreeSet class:

### 1. ****Basic Set Methods****

* add(E e): Adds the specified element to this set if it is not already present.
* remove(Object o): Removes the specified element from this set if it is present.
* contains(Object o): Returns true if this set contains the specified element.
* clear(): Removes all of the elements from this set.
* size(): Returns the number of elements in this set.
* isEmpty(): Returns true if this set contains no elements.

### 2. ****SortedSet Methods****

* first(): Returns the first (lowest) element currently in this set.
* last(): Returns the last (highest) element currently in this set.
* headSet(E toElement): Returns a view of the portion of this set whose elements are strictly less than toElement.
* tailSet(E fromElement): Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
* subSet(E fromElement, E toElement): Returns a view of the portion of this set between fromElement, inclusive, and toElement, exclusive.

### 3. ****NavigableSet Methods****

* lower(E e): Returns the greatest element less than the specified element, or null if there is no such element.
* floor(E e): Returns the greatest element less than or equal to the specified element, or null if there is no such element.
* ceiling(E e): Returns the least element greater than or equal to the specified element, or null if there is no such element.
* higher(E e): Returns the least element greater than the specified element, or null if there is no such element.
* pollFirst(): Retrieves and removes the first (lowest) element, or returns null if this set is empty.
* pollLast(): Retrieves and removes the last (highest) element, or returns null if this set is empty.

### 4. ****Bulk Operations****

* addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to this set.
* removeAll(Collection<?> c): Removes all of the elements in this set that are contained in the specified collection.
* retainAll(Collection<?> c): Retains only the elements in this set that are contained in the specified collection.
* containsAll(Collection<?> c): Returns true if this set contains all of the elements in the specified collection.

### 5. ****Iteration****

* iterator(): Returns an iterator over the elements in this set, in ascending order.
* descendingIterator(): Returns an iterator over the elements in this set, in descending order.
* forEach(Consumer<? super E> action): Performs the given action for each element of this set.

### 6. ****Array Operations****

* toArray(): Returns an array containing all of the elements in this set, in ascending order.
* toArray(T[] a): Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

### 9. ****Comparator****

* comparator(): Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.

### 10. ****Miscellaneous****

* clone(): Returns a shallow copy of this TreeSet.
* removeIf(Predicate<? super E> filter): Removes all elements of this set that satisfy the given predicate.

### 11. ****Order and Performance****

* **Sorted Order**: Maintains elements in a sorted order based on their natural ordering or a specified comparator.
* **Performance**: Provides O(log n) time cost for basic operations (add, remove, contains) due to the underlying Red-Black tree structure.

### 12. ****Limitations****

* **No Null Elements**: TreeSet does not allow null elements, as null cannot be compared.
* **Non-Concurrent**: TreeSet is not synchronized. If multiple threads access a TreeSet concurrently and at least one of the threads modifies the set structurally, it must be synchronized externally.

The TreeSet class is useful when you need a set that maintains elements in a sorted order and provides efficient access to the smallest and largest elements. It is suitable for applications requiring sorted data and operations that depend on the order of elements.