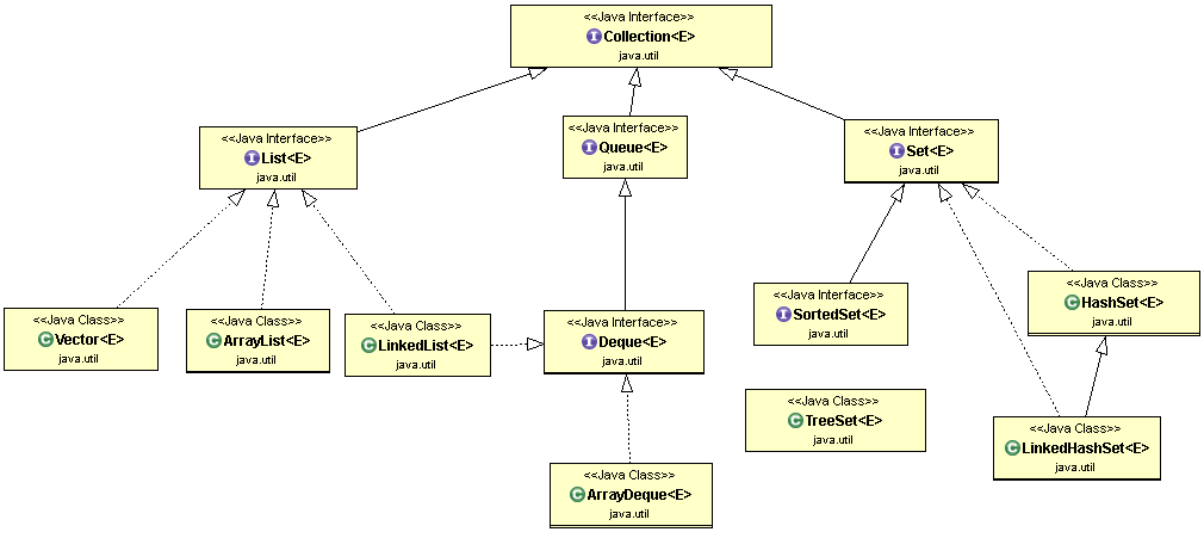
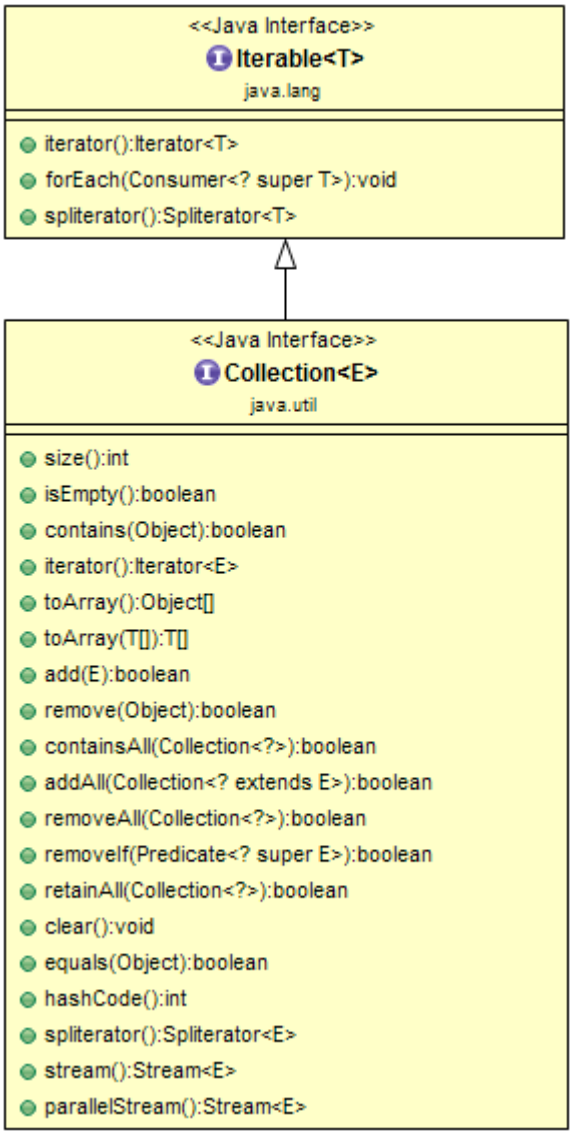
Java Collections



Core Interfaces

- Collections Framework - The Collection Interface
- Collections Framework - The Set Interface
- Collections Framework - The SortedSet Interface
- Collections Framework - The List Interface
- Collections Framework - The Queue Interface
- Collections Framework - The Deque Interface
- Collections Framework - The Map Interface
- Collections Framework - The SortedMap Interface

Collection Interface



Collection Interface Example

The following example demonstrates the usage of important Collection interface methods:

```
import java.util.ArrayList;

import java.util.Collection;


public class CollectionDemo {
```

```
    public static void main(String[] args) {

        Collection<String> fruitCollection = new ArrayList<>();
        fruitCollection.add("banana");
        fruitCollection.add("apple");
        fruitCollection.add("mango");
        System.out.println(fruitCollection);

        fruitCollection.remove("banana");
        System.out.println(fruitCollection);

        System.out.println(fruitCollection.contains("apple"));

        fruitCollection.forEach((element) -> {
            System.out.println(element);
        });

        fruitCollection.clear();

        System.out.println(fruitCollection);
    }
}
```

Output:

[banana, apple, mango]

[apple, mango]

true

apple

mango

[]

In the above example, we can clearly see that the Collection is an interface and we used it as a reference type:

Collection<String> fruitCollection = new ArrayList<>();

Collection Interface Important Methods with Examples

boolean add()

Adds a specific element to the collection. Returns true if the collection is modified, false otherwise.

Collection<String> collection = new ArrayList<>();

boolean isAdded = collection.add("Hello");

System.out.println(isAdded); // Output: true

boolean addAll(Collection<? extends E> c)

Adds all elements from a specified collection to the current collection. Returns true if the collection is modified, false otherwise.

Collection<String> collection1 = new ArrayList<>();

Collection<String> collection2 = new ArrayList<>();

collection2.add("Hello");

```
collection2.add("World");
boolean isAllAdded = collection1.addAll(collection2);
System.out.println(isAllAdded);  // Output: true
```

void clear()

Removes all elements from the collection.

```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
collection.clear();
System.out.println(collection.size());  // Output: 0
```

boolean contains(Object o)

Checks if the collection contains the specified element. Returns true if it does, false otherwise.

```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
boolean contains = collection.contains("Hello");
System.out.println(contains);  // Output: true
```

boolean containsAll(Collection<?> c)

Checks if the collection contains all elements from a specified collection. Returns true if it does, false otherwise.

```
Collection<String> collection1 = new ArrayList<>();
Collection<String> collection2 = new ArrayList<>();
collection2.add("Hello");
collection2.add("World");
collection1.add("Hello");
collection1.add("World");
```

```
boolean containsAll = collection1.containsAll(collection2);
System.out.println(containsAll);  // Output: true
```

boolean isEmpty()

Checks if the collection is empty. Returns true if it is, false otherwise.

```
Collection<String> collection = new ArrayList<>();
boolean isEmpty = collection.isEmpty();
System.out.println(isEmpty);  // Output: true
```

Iterator iterator()

Returns an iterator that can be used to traverse the collection.

```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
Iterator<String> iterator = collection.iterator();
while(iterator.hasNext()) {
    System.out.println(iterator.next());  // Output: Hello
}
```

default Stream parallelStream()

Returns a possibly parallel Stream with the collection as its source.

```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
collection.parallelStream().forEach(System.out::println);  // Output: Hello
```

boolean remove(Object o)

Removes a single instance of the specified element from the collection, if present. Returns true if the collection is modified, false otherwise.

```
Collection<String> collection = new ArrayList<>();

collection.add("Hello");

boolean isRemoved = collection.remove("Hello");

System.out.println(isRemoved);  // Output: true
```

boolean removeAll(Collection<?> c)

Removes all elements in the collection that are also contained in the specified collection. Returns true if the collection is modified, false otherwise.

```
Collection<String> collection1 = new ArrayList<>();

Collection<String> collection2 = new ArrayList<>();

collection2.add("Hello");

collection1.add("Hello");

collection1.add("World");

boolean isAllRemoved = collection1.removeAll(collection2);

System.out.println(isAllRemoved);  // Output: true
```

boolean retainAll(Collection<?> c)

Retains only the elements in the collection that are also contained in the specified collection. Returns true if the collection is modified, false otherwise.

```
Collection<String> collection1 = new ArrayList<>();

Collection<String> collection2 = new ArrayList<>();

collection2.add("Hello");

collection1.add("Hello");

collection1.add("World");
```

```
boolean isRetained = collection1.retainAll(collection2);
System.out.println(isRetained);  // Output: true
```

int size()

Returns the number of elements in the collection.

```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
int size = collection.size();
System.out.println(size);  // Output: 1
```

default Spliterator spliterator()

Returns a Spliterator over the elements in the collection.

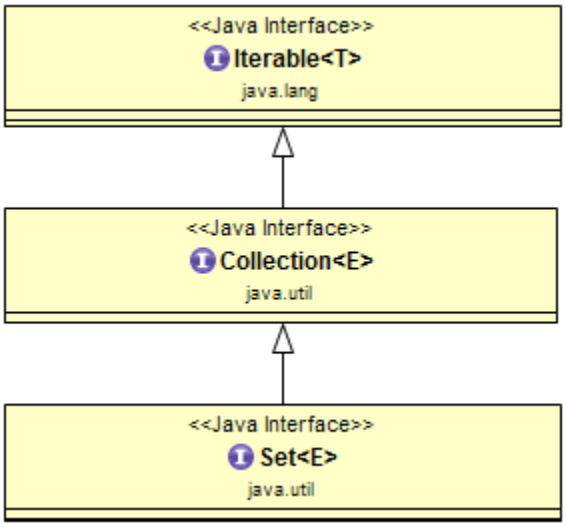
```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
Spliterator<String> spliterator = collection.spliterator();
spliterator.forEachRemaining(System.out::println);  // Output: Hello
```

default Stream stream()

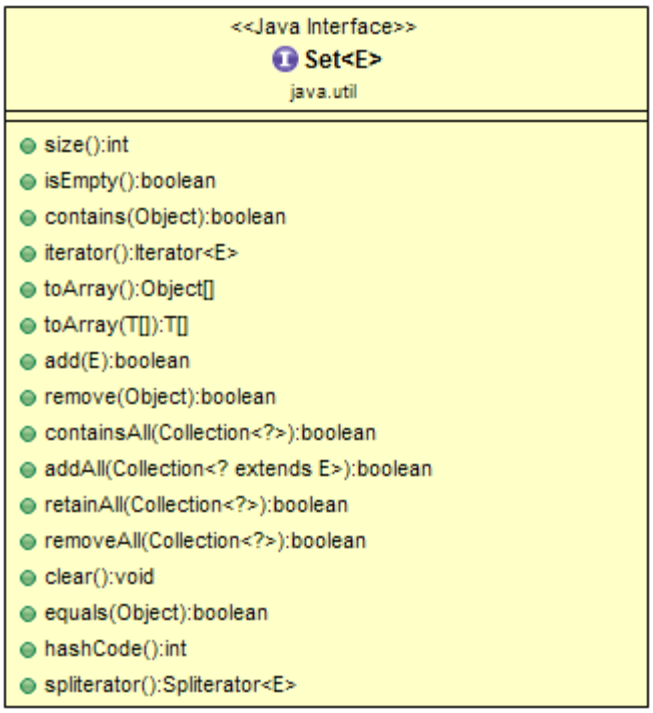
Returns a sequential Stream with the collection as its source.

```
Collection<String> collection = new ArrayList<>();
collection.add("Hello");
collection.stream().forEach(System.out::println);  // Output: Hello
```

Set Interface



Set Interface APIs/Methods



Below are the important *Set* interface methods with descriptions:

- **boolean add(E e)** - This method adds the specified element to this set if it is not already present (optional operation).
- **boolean addAll(Collection<? extends E> c)** - This method adds all of the elements in the specified collection

to this set if they're not already present (optional operation).

- _void clear() _- This method removes all of the elements from this set (optional operation).
- **boolean contains(Object o)** - This method returns true if this set contains the specified element.
- **boolean containsAll(Collection<?> c)** - This method returns true if this set contains all of the elements of the specified collection.
- **boolean equals(Object o)** - This method compares the specified object with this set for equality.
- **int hashCode()** - This method returns the hash code value for this set.
- **boolean isEmpty()** - This method returns true if this set contains no elements.
- **Iterator iterator()** - This method returns an iterator over the elements in this set.
- **boolean remove(Object o)** - This method removes the specified element from this set if it is present (optional operation).
- **boolean removeAll(Collection<?> c)** - This method removes from this set all of its elements that are contained in the specified collection (optional operation).
- **boolean retainAll(Collection<?> c)** - This method retains only the elements in this set that are contained in the specified collection (optional operation).
- **int size()** - This method returns the number of elements in this set (its cardinality).
- **default Spliterator spliterator()** - This method creates a Spliterator over the elements in this set.

Example 1: Set Interface with Its HashSet Implementation Class

Let's create a simple example of *Set* interface using the **HashSet implementation** class:

```
import java.util.HashSet;


import java.util.Set;


public class CreateHashSetExample {
    public static void main(String[] args) {
        // Creating a HashSet
     Set<String> daysOfWeek = new HashSet<>();


        // Adding new elements to the HashSet
        daysOfWeek.add("Monday");
        daysOfWeek.add("Tuesday");
        daysOfWeek.add("Wednesday");
        daysOfWeek.add("Thursday");
        daysOfWeek.add("Friday");
        daysOfWeek.add("Saturday");
        daysOfWeek.add("Sunday");


        // Adding duplicate elements will be ignored
        daysOfWeek.add("Monday");
        System.out.println(daysOfWeek);
```

```
    }
}
```

Output:

[Monday, Thursday, Friday, Sunday, Wednesday, Tuesday, Saturday]

Example 2: Set Interface with Its LinkedHashSet Implementation Class

```
// Creating a HashSet
LinkedHashSet<String> daysOfWeek = new LinkedHashSet<>();

// Adding new elements to the HashSet
daysOfWeek.add("Monday");
daysOfWeek.add("Tuesday");
daysOfWeek.add("Wednesday");
daysOfWeek.add("Thursday");
daysOfWeek.add("Friday");
daysOfWeek.add("Saturday");
daysOfWeek.add("Sunday");

// Adding duplicate elements will be ignored
daysOfWeek.add("Monday");

System.out.println(daysOfWeek);
```

Output:

[Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday]

Example 3: Set Interface with Its TreeSet Implementation Class

```
// Creating a TreeSet
TreeSet<String> fruits = new TreeSet<>();

// Adding new elements to a TreeSet
fruits.add("Banana");
fruits.add("Apple");
fruits.add("Pineapple");
fruits.add("Orange");

System.out.println("Fruits Set : " + fruits);

// Duplicate elements are ignored
fruits.add("Apple");
System.out.println("After adding duplicate element \"Apple\" : " + fruits);

// This will be allowed because it's in lowercase.
fruits.add("banana");
System.out.println("After adding \"banana\" : " + fruits);
```

Output:

Fruits Set : [Apple, Banana, Orange, Pineapple]

After adding duplicate element "Apple" : [Apple, Banana, Orange, Pineapple]

After adding "banana" : [Apple, Banana, Orange, Pineapple, banana]

Note that in this example, duplicate elements are ignored.

Set Interface Implementation Classes

There are three general-purpose Set implementations:

- **HashSet**
- **TreeSet**
- **LinkedHashSet**

About SortedSet Interface

Here are some key points about the SortedSet interface:

Sorting:

SortedSet is a Set that maintains its elements in ascending order. The sorting can be based on either natural order or it can be customized through a comparator at *SortedSet* creation time.

Ordering:

Elements inserted into the SortedSet need to implement the Comparable interface. The elements are ordered by using their compareTo() method unless a Comparator is provided at set creation time.

No Duplicates:

Just like any other set, SortedSet doesn't allow duplicates.

Methods:

In addition to standard Set operations, the SortedSet interface provides operations for the following: Range view — allows arbitrary range operations on the sorted set.

Endpoints — returns the first or last element in the sorted set.

Subinterfaces and Implementations:

The primary SortedSet implementation in the Java Collections Framework is **TreeSet**.

Null Elements:

SortedSet implementations (like TreeSet) don't permit the use of null elements, because null is not comparable.

Use-cases:

SortedSet is ideal when you want to store unique elements in a sorted manner.

Thread Safety:

SortedSet implementations are not thread-safe, but you can make them thread-safe using synchronized wrappers obtained from the Collections utility class.

Iterators:

The iterator provided in the SortedSet interface iterates over the elements in the set in ascending order.

Head, Tail, and Subsets:

The SortedSet interface provides methods to get subsets from the set, which are still sorted according to the set's ordering.

Remember, SortedSet is an interface and you cannot instantiate interfaces. Therefore, you need to either use its subclass or a class that implements the SortedSet interface to create an object.

SortedSet Interface Example

This example demonstrates some important methods of the *SortedSet* Interface using **TreeSet** implementation class.

```
import java.util.Comparator;

import java.util.SortedSet;

import java.util.TreeSet;


public class CreateTreeSetExample {

    public static void main(String[] args) {

        // Creating a TreeSet

        SortedSet<String> fruits = new TreeSet<>();


        // Adding new elements to a TreeSet

        fruits.add("Banana");

        fruits.add("Apple");

        fruits.add("Pineapple");

        fruits.add("Orange");


        // Returns the first (lowest) element currently in this set.

        String first = fruits.first();

        System.out.println("First element : " + first);
```

```
        // Returns the last (highest) element currently in this set.
        String last = fruits.last();
        System.out.println("Last element : " + last);

        // Returns the comparator used to order the elements in
this set, or
        // null if this set uses the natural ordering of its elements.
        Comparator<?> comparator = fruits.comparator();

        SortedSet<String> tailSet = fruits.tailSet("Orange");
        System.out.println("tailSet :" + tailSet);
    }
}
```

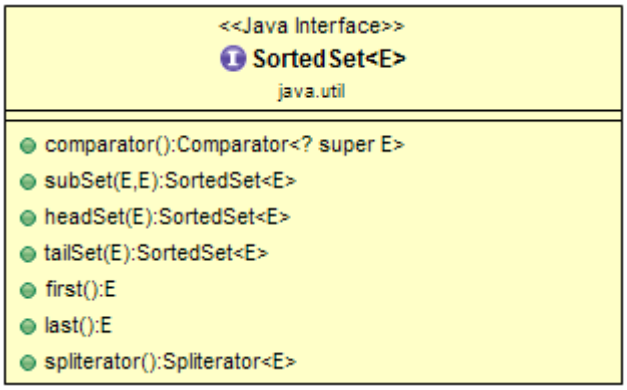
Output:

First element : Apple

Last element : Pineapple

tailSet :[Orange, Pineapple]

SortedSet interface methods



About List Interface

Here are some key points about the List Interface in Java:

Ordered Collection:

The *List* interface extends the **Collection interface** and represents an ordered collection (also known as a sequence). The elements in a *List* can be inserted or accessed at any position based on the index.

Duplicates:

The List allows duplicates. It means you can insert duplicate elements in the List.

Null Elements:

The List allows any number of null elements. You can have a List with all elements as null.

Methods:

In addition to the methods inherited from the **Collection interface**, the List interface includes methods for position (index-based) access, search operations, and list iteration.

Subinterfaces and Implementations:

The commonly used classes that implement the *List* interface are **ArrayList**, **LinkedList**, Vector, and **Stack**.

ListIterator:

The List interface provides a special iterator, called **ListIterator**, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides.

Mutability:

The List interface supports elements insertion and removal, and it is typically more flexible than arrays.

Use-cases:

The List is a good choice if you need to maintain the insertion order, allow duplicates and nulls, and frequently access elements with the index.

Equality:

Two List objects are considered equal if they contain the same elements in the same order.

Example 1: List Interface with Its ArrayList Implementation Class

Here is a simple *List* interface example using the *ArrayList* implementation class. This example demonstrates that List allows storing duplicate elements, and null values, and maintains the insertion order:

```
package com.java.collections.interfaces;


import java.util.ArrayList;

import java.util.List;


public class ListDemo {

    public static void main(String[] args) {

        List < String > list = new ArrayList < > ();

        // List allows you to add duplicate elements

        list.add("element1");
```

```
    list.add("element1");
    list.add("element2");
    list.add("element2");
    System.out.println(list);

    // List allows you to have ‘null’ elements.
    list.add(null);
    list.add(null);
    System.out.println(list);

    // insertion order
    list.add("element1"); // 0
    list.add("element2"); // 1
    list.add("element4"); // 2
    list.add("element3"); // 3
    list.add("element5"); // 4

    System.out.println(list);

    // access elements from list

    System.out.println(list.get(0));
    System.out.println(list.get(4));

}
```

}

Output

[element1, element1, element2, element2]

[element1, element1, element2, element2, null, null]

[element1, element1, element2, element2, null, null, element1, element2, element4, element3, element5]

element1

null

Example 2: List Interface with Its LinkedList Implementation Class

The following example shows how to use the List interface with LinkedList implementation class:

```
package com.javaguides.collections.linkedlistexamples;


import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;


public class CreateLinkedListExample {
    public static void main(String[] args) {
        // Creating a LinkedList
        LinkedList<String> fruits = new LinkedList<>();


        // Adding new elements to the end of the LinkedList using add() method.
        fruits.add("Banana");
```

```
        fruits.add("Apple");
        fruits.add("mango");

        System.out.println("Initial LinkedList : " + fruits);

        // Adding an element at the specified position in the
LinkedList
        fruits.add(2, "Watermelon");

        System.out.println("After add(2, \"D\") : " + fruits);

        // Adding an element at the beginning of the LinkedList
        fruits.addFirst("Strawberry");
        System.out.println("After addFirst(\"Strawberry\") : " +
fruits);

        // Adding an element at the end of the LinkedList
        // (This method is equivalent to the add() method)
        fruits.addLast("Orange");
        System.out.println("After addLast(\"F\") : " + fruits);

        // Adding all the elements from an existing collection to
        // the end of the LinkedList
        List<String> moreFruits = new ArrayList<>();
        moreFruits.add("Grapes");
```

```
        moreFruits.add("Pyrus");


        fruits.addAll(moreFruits);
        System.out.println("After addAll(moreFruits) : " + fruits);
    }
}
```

Output:

Initial LinkedList : [Banana, Apple, mango]

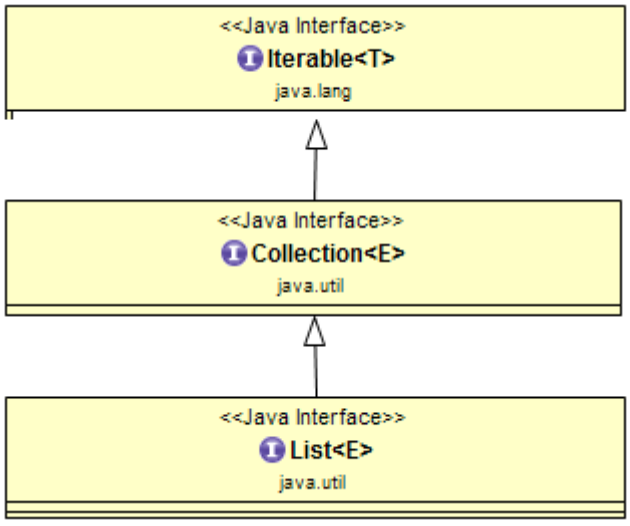
After add(2, "D") : [Banana, Apple, Watermelon, mango]

After addFirst("Strawberry") : [Strawberry, Banana, Apple, Watermelon, mango]

After addLast("F") : [Strawberry, Banana, Apple, Watermelon, mango, Orange]

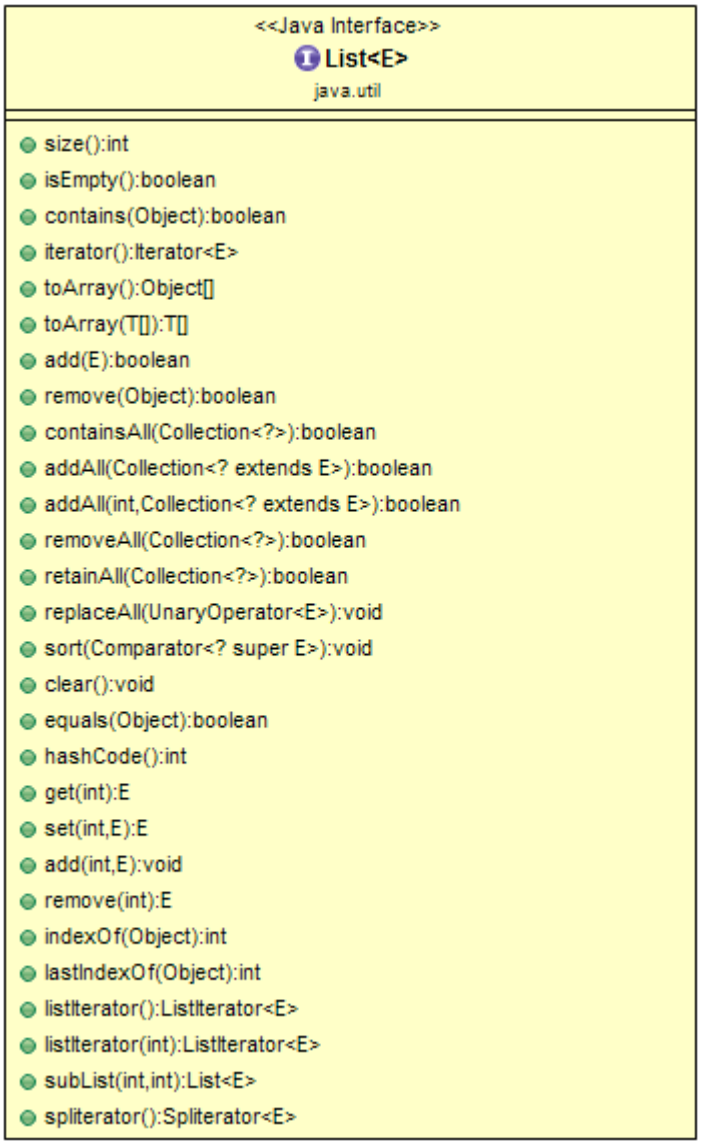
After addAll(moreFruits) : [Strawberry, Banana, Apple, Watermelon, mango, Orange, Grapes, Pyrus]

List Interface



List Interface Methods

This class diagram shows a list of APIs/Methods that the List interface provides.



List Interface Common Implementation

List interface implementations classes:

- **ArrayList**
- **LinkedList**
- Vector
- **Stack**