Collection interface

11 November 2025 09:37

interface Drawable{

Java 1.0

basic idea

// static final variables

// methods are abstract by default.

// it only contains method prototype.

// the class which implements the

// interface has to implement all

// the methods of interface.

Java 9

// we can define private, default method in

// interface.

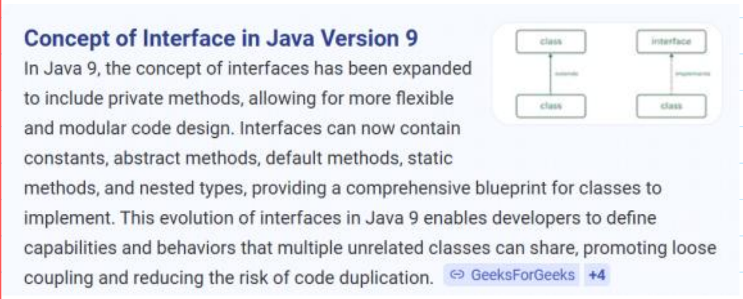
// We can define static method in interface.

// inner interface.

}

interface is a blueprint of class.

" class is a blueprint of object"



Concept of Interface in Java Version 9

In Java 9, the concept of interfaces has been expanded to include private methods, allowing for more flexible and modular code design. Interfaces can now contain constants, abstract methods, default methods, static methods, and nested types, providing a comprehensive blueprint for classes to implement. This evolution of interfaces in Java 9 enables developers to define capabilities and behaviors that multiple unrelated classes can share, promoting loose coupling and reducing the risk of code duplication. GeeksForGeeks +4

If we want to represent a group of individual object as a single entity, then we should go for Collection.
Collection interface defines the most common methods which are applicable for any Collection object.

Official Notes:
public interface **Collection<E>** extends Iterable<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 are unordered. Collections that have a defined encounter order are generally subtypes of the SequencedCollection interface. 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.

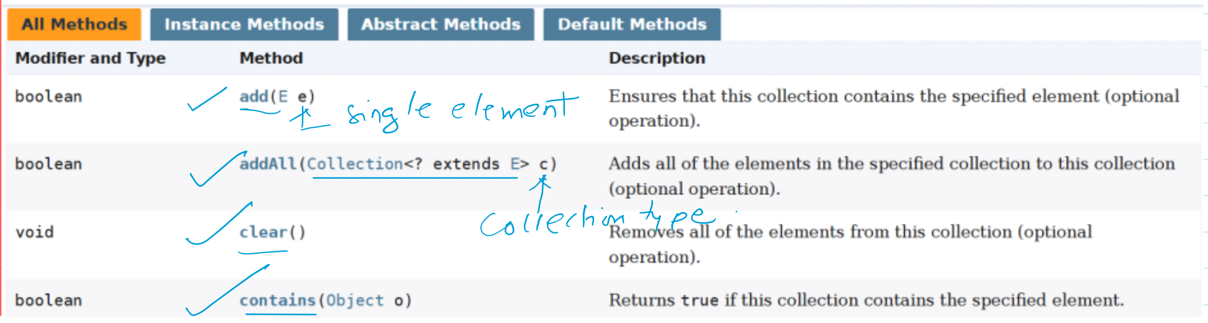
Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.
All general-purpose Collection implementation classes (which typically implement Collection indirectly through one of its subinterfaces) should provide two "standard" constructors: 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. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose Collection implementations in the Java platform libraries comply.
Certain methods are specified to be *optional*. If a collection implementation doesn't implement a particular operation, it should define the corresponding method to throw UnsupportedOperationException. Such methods are marked "optional operation" in method specifications of the collections interfaces.
Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.
It is up to each collection to determine its own synchronization policy. In the absence of a stronger guarantee by the implementation, undefined behavior may result from the invocation of any method on a collection that is being mutated by another thread; this includes direct invocations, passing the collection to a method that might perform invocations, and using an existing iterator to examine the collection.
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))." This specification should *not* be construed to imply that invoking Collection.contains with a non-null argument o will cause o.equals(e) to be invoked for any element e. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two elements. (The Object.hashCode() specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying Object methods wherever the implementor deems it appropriate.
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. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

Generics

public interface **Collection<E>** extends Iterable<E>

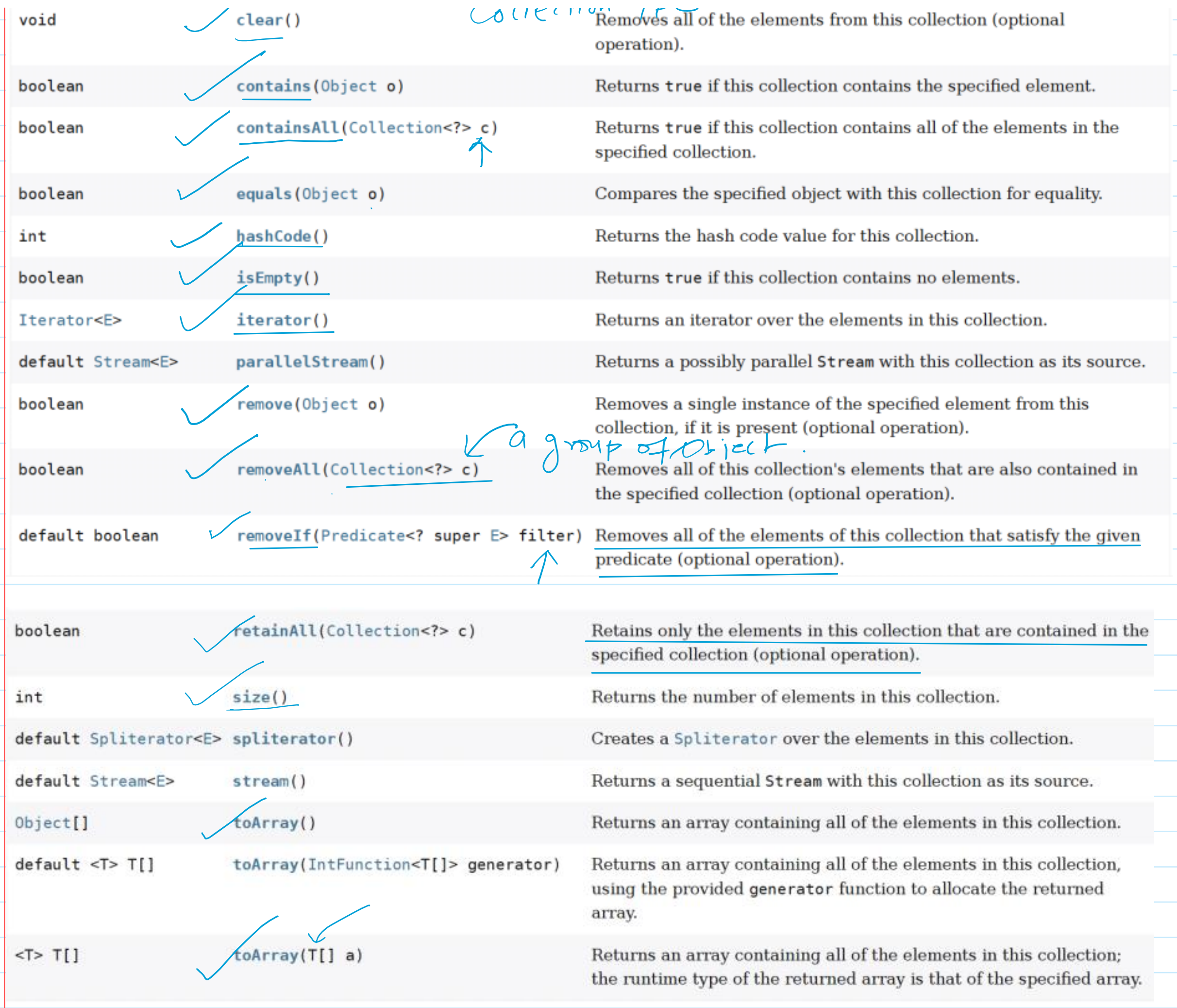
↑ type of Objects that we want to keep in Collection.



All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
boolean	✓ add(E e) ↑ single element	Ensures that this collection contains the specified element (optional operation).
boolean	✓ addAll(Collection<? extends E> c) ↑ Collection type	Adds all of the elements in the specified collection to this collection (optional operation).
void	✓ clear()	Removes all of the elements from this collection (optional operation).
boolean	✓ contains(Object o)	Returns true if this collection contains the specified element.

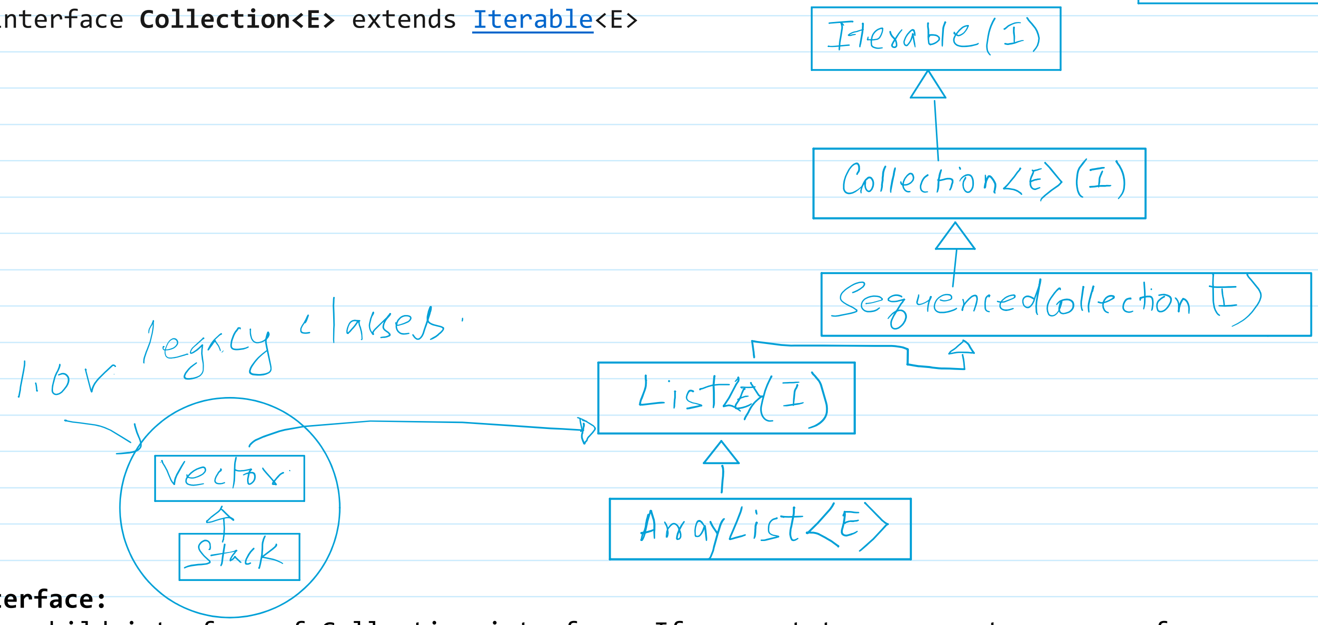
Convention
↓
E ← Element
T ←
U ←



void	clear()	Removes all of the elements from this collection (optional operation).
boolean	contains(Object o)	Returns true if this collection contains the specified element.
boolean	containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
boolean	equals(Object o)	Compares the specified object with this collection for equality.
int	hashCode()	Returns the hash code value for this collection.
boolean	isEmpty()	Returns true if this collection contains no elements.
Iterator<E>	iterator()	Returns an iterator over the elements in this collection.
default Stream<E>	parallelStream()	Returns a possibly parallel Stream with this collection as its source.
boolean	remove(Object o)	Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate (optional operation).
boolean	retainAll(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	size()	Returns the number of elements in this collection.
default Spliterator<E>	spliterator()	Creates a Spliterator over the elements in this collection.
default Stream<E>	stream()	Returns a sequential Stream with this collection as its source.
Object[]	toArray()	Returns an array containing all of the elements in this collection.
default <T> T[]	toArray(IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.
<T> T[]	toArray(T[] a)	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

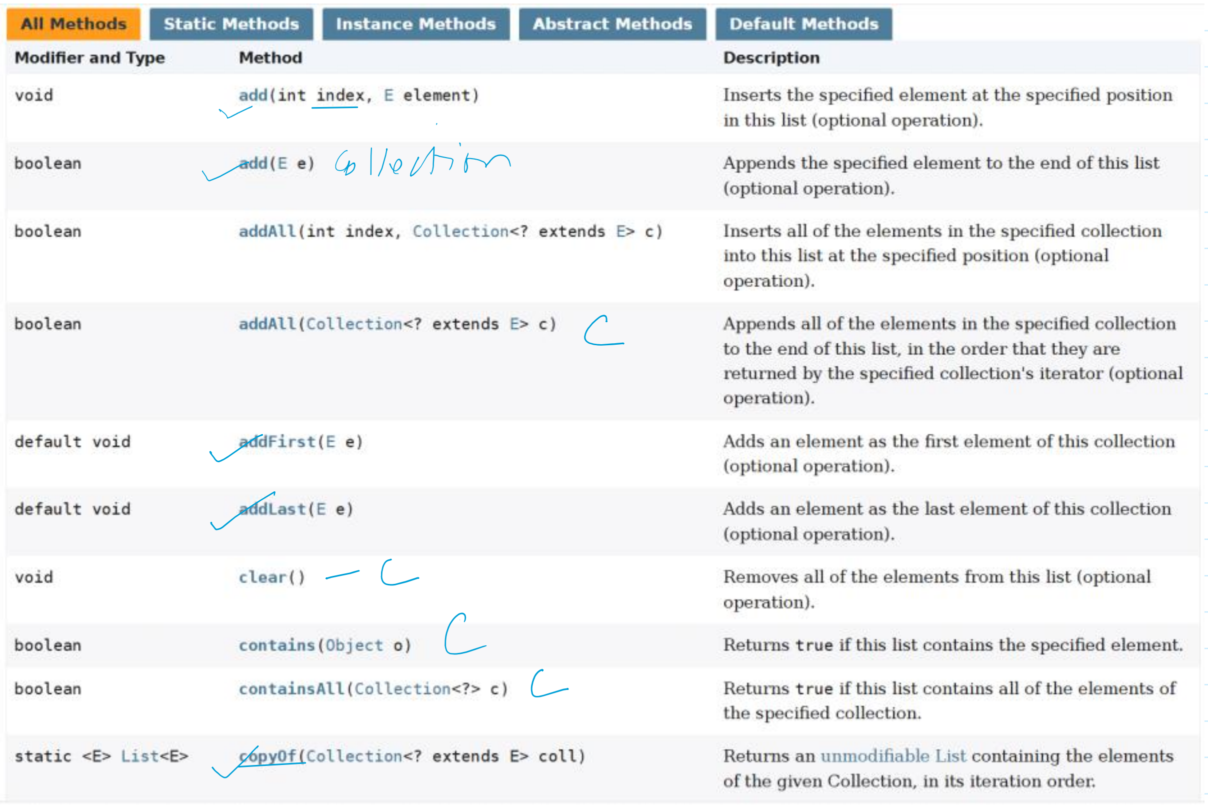
I = interface

public interface **Collection<E>** extends Iterable<E>



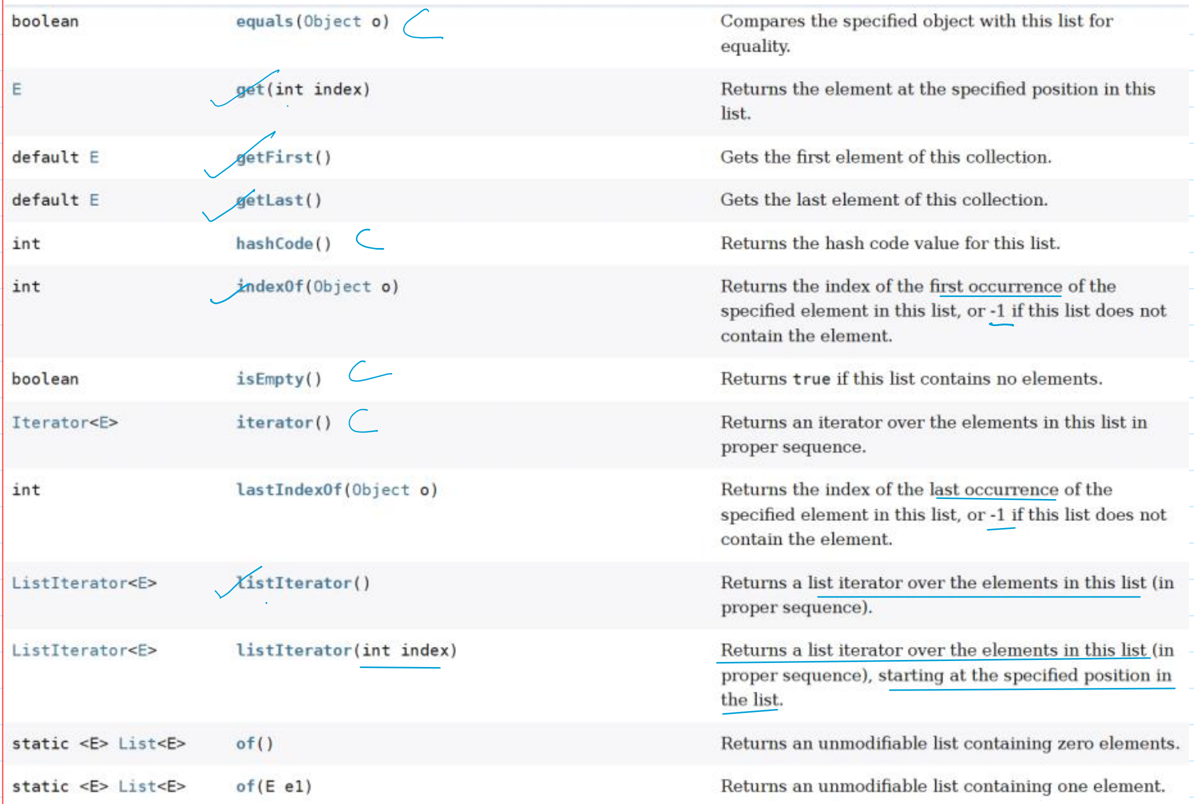
List interface:
List is a child interface of Collection interface. If we want to represent a group of individual object as a single entity where duplicates are allowed and insertion order must be preserved, then we should go for List interface.
We can preserve insertion order with index. And we can differentiate duplicate object by using index. Here index will play very important role in the list. List interface defines the following specific methods:

```
public interface List<E> extends SequencedCollection<E>
```

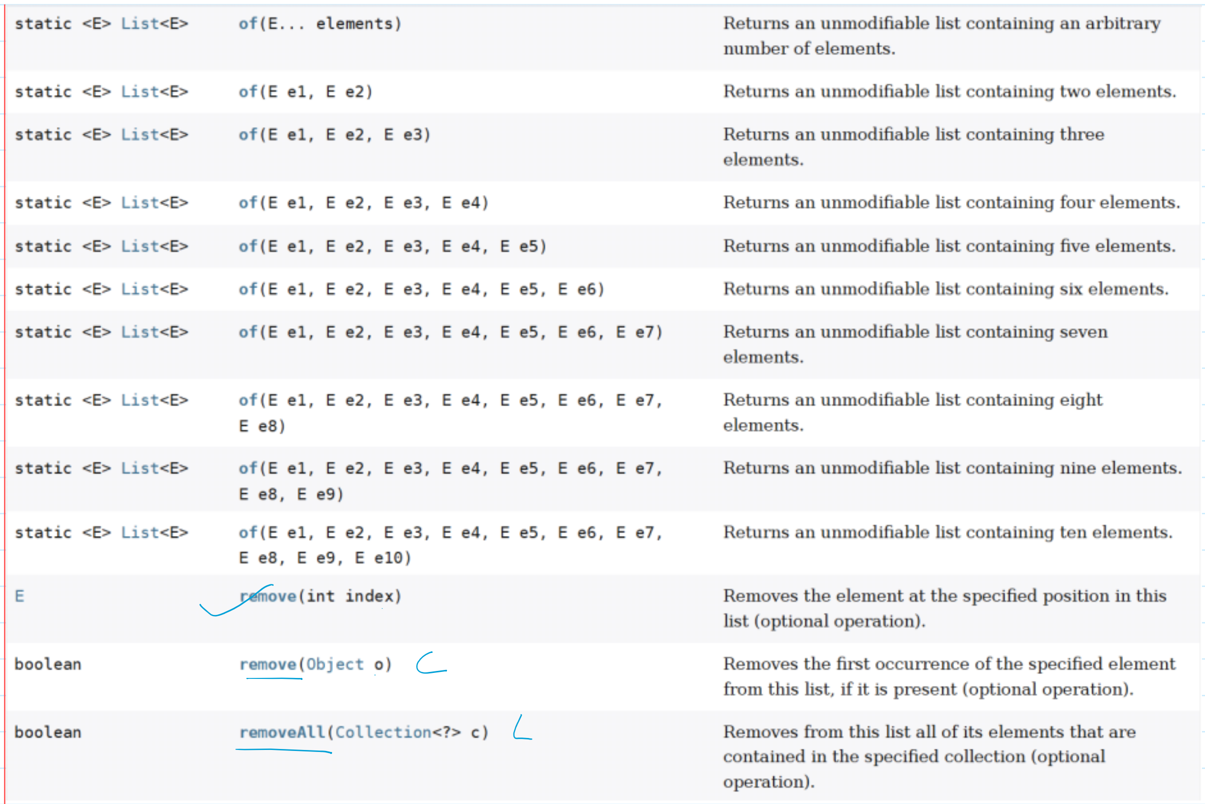


All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

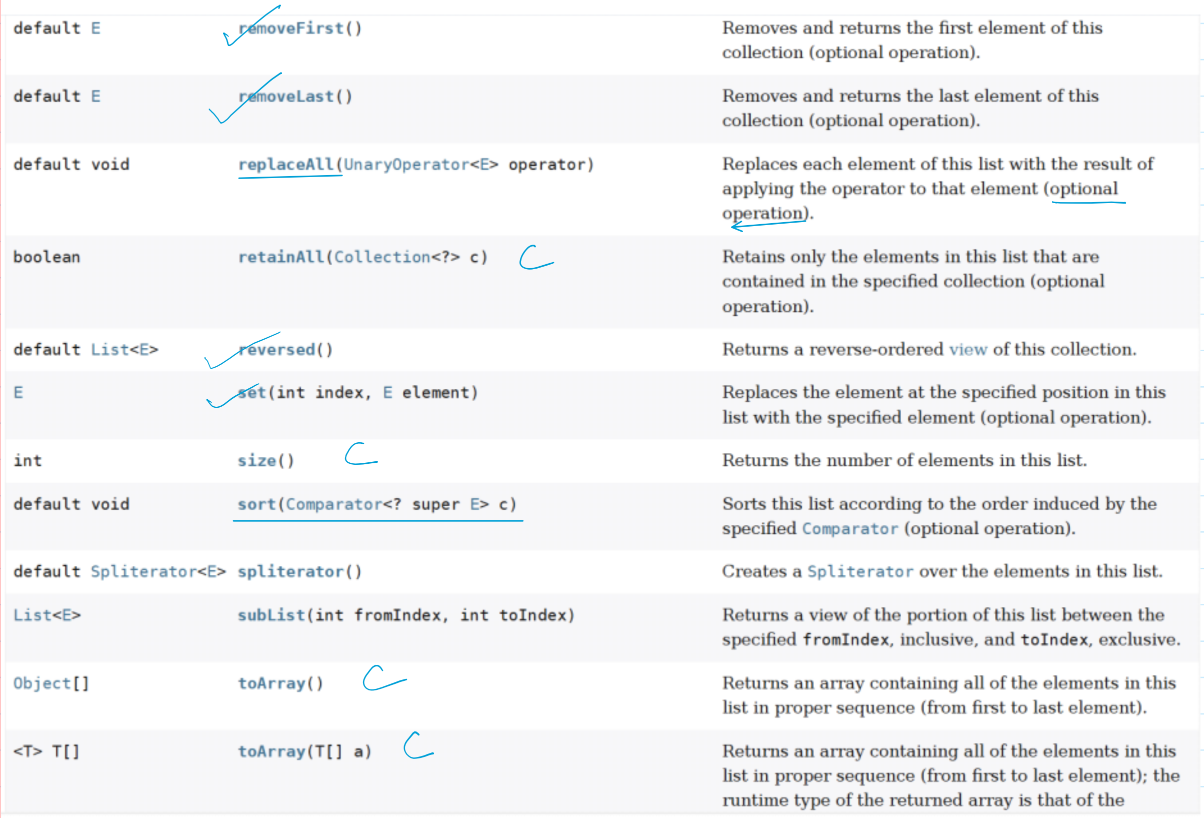
Modifier and Type	Method	Description
void	✓ add(int index, E element)	Inserts the specified element at the specified position in this list (optional operation).
boolean	✓ add(E e) Collection	Appends the specified element to the end of this list (optional operation).
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
boolean	addAll(Collection<? extends E> c) C	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
default void	✓ addFirst(E e)	Adds an element as the first element of this collection (optional operation).
default void	✓ addLast(E e)	Adds an element as the last element of this collection (optional operation).
void	clear() — C	Removes all of the elements from this list (optional operation).
boolean	contains(Object o) C	Returns true if this list contains the specified element.
boolean	containsAll(Collection<?> c) C	Returns true if this list contains all of the elements of the specified collection.
static <E> List<E>	✓ copyOf(Collection<? extends E> coll)	Returns an unmodifiable List containing the elements of the given Collection, in its iteration order.



Modifier and Type	Method	Description
boolean	equals(Object o) C	Compares the specified object with this list for equality.
E	✓ get(int index)	Returns the element at the specified position in this list.
default E	✓ getFirst()	Gets the first element of this collection.
default E	✓ getLast()	Gets the last element of this collection.
int	hashCode() C	Returns the hash code value for this list.
int	✓ indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() C	Returns true if this list contains no elements.
Iterator<E>	iterator() C	Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	✓ listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
static <E> List<E>	of()	Returns an unmodifiable list containing zero elements.
static <E> List<E>	of(E e1)	Returns an unmodifiable list containing one element.



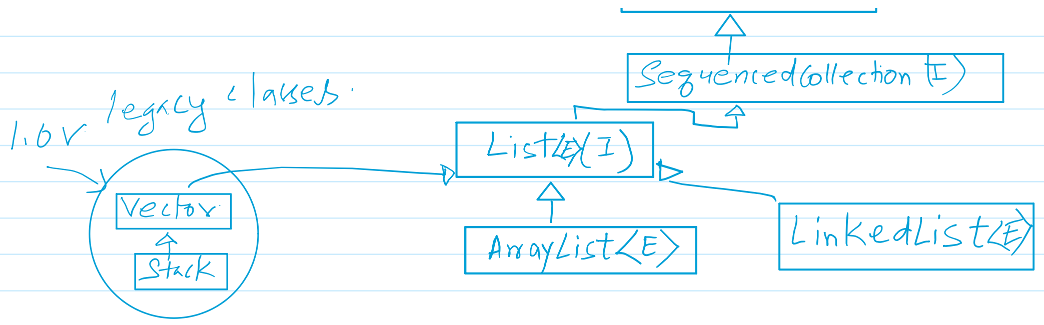
`static <E> List<E>`	`of(E... elements)`	Returns an unmodifiable list containing an arbitrary number of elements.
`static <E> List<E>`	`of(E e1, E e2)`	Returns an unmodifiable list containing two elements.
`static <E> List<E>`	`of(E e1, E e2, E e3)`	Returns an unmodifiable list containing three elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4)`	Returns an unmodifiable list containing four elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5)`	Returns an unmodifiable list containing five elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6)`	Returns an unmodifiable list containing six elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)`	Returns an unmodifiable list containing seven elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)`	Returns an unmodifiable list containing eight elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)`	Returns an unmodifiable list containing nine elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)`	Returns an unmodifiable list containing ten elements.
`E`	✓ `remove(int index)`	Removes the element at the specified position in this list (optional operation).
`boolean`	`remove(Object o)` C	Removes the first occurrence of the specified element from this list, if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)` C	Removes from this list all of its elements that are contained in the specified collection (optional operation).



`default E`	✓ `removeFirst()`	Removes and returns the first element of this collection (optional operation).
`default E`	✓ `removeLast()`	Removes and returns the last element of this collection (optional operation).
`default void`	`replaceAll(UnaryOperator<E> operator)`	Replaces each element of this list with the result of applying the operator to that element (optional operation).
`boolean`	`retainAll(Collection<?> c)` C	Retains only the elements in this list that are contained in the specified collection (optional operation).
`default List<E>`	✓ `reversed()`	Returns a reverse-ordered view of this collection.
`E`	✓ `set(int index, E element)`	Replaces the element at the specified position in this list with the specified element (optional operation).
`int`	`size()` C	Returns the number of elements in this list.
`default void`	`sort(Comparator<? super E> c)`	Sorts this list according to the order induced by the specified `Comparator` (optional operation).
`default Spliterator<E>`	`spliterator()`	Creates a `Spliterator` over the elements in this list.
`List<E>`	`subList(int fromIndex, int toIndex)`	Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
`Object[]`	`toArray()` C	Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[]`	`toArray(T[] a)` C	Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the

$$\begin{tikzcd}
\boxed{\text{Iterable(I)}} \\
\boxed{\text{Collection<E>(I)}} \arrow[u] \\
\boxed{\text{SequencedCollection(I)}} \arrow[u]
\end{tikzcd}$$

[illegible] classes:



ArrayList is a class:
It has following properties:
1. Resizable array or growable or shrinkable array.
2. Duplicates are allowed.
3. Insertion order is preserved.
4. Can hold heterogenous objects. (Except TreeSet and TreeMap object)
5. "null" insertion is allowed.

Constructors

1. ArrayList<Object> al = new ArrayList<>(); => Creates an empty array list object with default initial capacity 10. Once array list reaches its maximum capacity then a new ArrayList object will be created with new capacity.

```
NewCapacity = currentCapacity * 3/2 + 1; //   10*3/2 + 1 = 16
```

2. ArrayList<Object> al = new ArrayList<>(int initialCapacity); => Creates an empty array list object with a specified initial capacity.

3. ArrayList al = new ArrayList(Collection c); => Creates an equivalent array list object for the given collection.

Official Document:
public class **ArrayList<E>** extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)
The size, isEmpty, get, set, getFirst, getLast, removeLast, iterator, listIterator, and reversed operations run in constant time. The add, and addLast operations 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. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
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.
Note that this implementation is not synchronized. If multiple threads access

an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
  List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's iterator and listIterator methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:
1.2

Collection Framework came in version Java version 1.2.

Element of any type.

public class **ArrayList<E>** extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

We can copy one ArrayList object into another one.

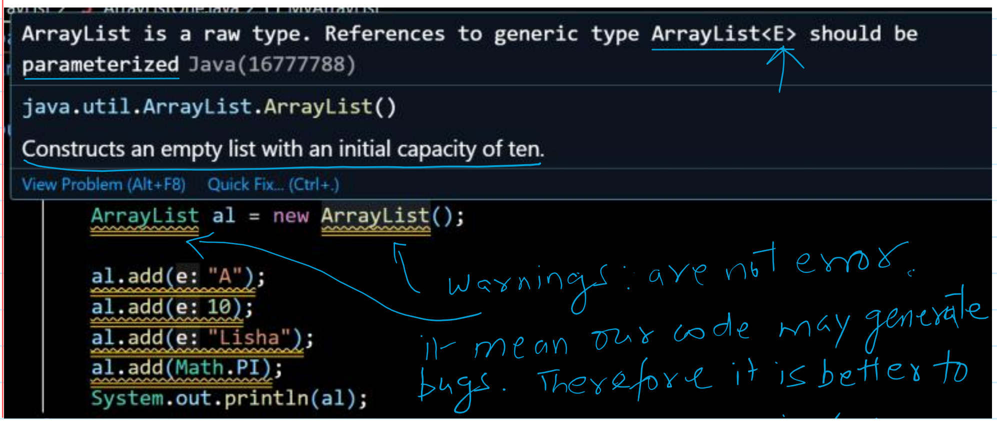
class

the indexing is possible by the help of RandomAccess interface

(1.4) → RandomAccess, Serializable } ← marker interface.

interface without body utilized by JVM.

The primary purpose of these interfaces is to allow generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access.



ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized Java(16777788)

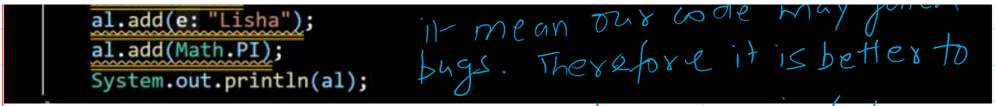
java.util.ArrayList.ArrayList()

Constructs an empty list with an initial capacity of ten.

```
ArrayList al = new ArrayList();

al.add(e: "A");
al.add(e: 10);
al.add(e: "Lisha");
al.add(Math.PI);
System.out.println(al);
```

Warnings: are not error. it mean our code may generate bugs. Therefore it is better to

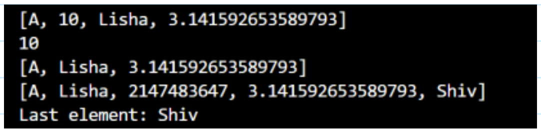


```
        al.add(e: "Lisha");
        al.add(Math.PI);
        System.out.println(al);
```

it mean our code may gener
bugs. Therefore it is better to
remove these warnings from your code.

```
package MyArrayList;
import java.util.ArrayList;
public class ArrayListOne {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("A");
        al.add(10);
        al.add("Lisha");
        al.add(Math.PI);
        System.out.println(al);
        System.out.println(al.remove(1));
        System.out.println(al);
        al.add(2, Integer.MAX_VALUE);
        al.add("Shiv");
        System.out.println(al);
        System.out.println("Last element: "+al.getLast());
    }
}
```

Output:



```
[A, 10, Lisha, 3.141592653589793]
10
[A, Lisha, 3.141592653589793]
[A, Lisha, 2147483647, 3.141592653589793, Shiv]
Last element: Shiv
```

ArrayList is the best choice if our frequent operation is retrieval operation. Because ArrayList implements RandomAccess interface.

ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle.

Home work

1. **Student Marks Manager**
   Write a Java program to store the marks of students in an ArrayList of integers. Your program should be menu-driven and offer the following features using methods:
   - Add a new mark
   - Remove an existing mark
   - Update a mark at a given index
   - Display all marks
   - Display the highest and lowest marks
   - Calculate and display the average marks

   Ensure that all operations handle invalid index and empty list cases gracefully.
2. **Inventory System for a Grocery Store**
   Create a class Item with fields name, price, and quantity. Store multiple Item objects inside an ArrayList. Write methods to:
   - Add a new item
   - Delete an item by name

- Update the quantity or price of an item
- Display the details of all items
- Search for an item by name (use equals properly)

  Write a main program that allows the user to perform these operations repeatedly until they choose to exit.

3. **Library Book Record Management**

   Define a class Book with attributes bookId, title, author, and status (Available/Issued).

   Use an ArrayList to store multiple Book objects. Implement methods to:

- Add a new book
- Issue a book (change status from Available to Issued)
- Return a book (change status from Issued to Available)
- Display all available books
- Search and display details of books written by a particular author

  Your program must handle cases where book IDs do not exist.

4. **Unique Names List**

   Write a Java program that reads names from the user and stores them in an ArrayList of strings until the user types "STOP".

   Before adding each new name, check whether it already exists in the list. If it does, display a message "Name already exists!" and do not add it.

   After input ends, perform the following operations:

- Display the total number of unique names
- Sort the names alphabetically and display them
- Ask the user for a substring and display all names that contain that substring

  Use appropriate ArrayList and String methods.

5. **Employee Salary Processing System**

   Create an Employee class with fields: id, name, and salary.

   Store multiple Employee objects in an ArrayList. Write methods to:

- Display all employees
- Display employees having salary above a given amount
- Increase salary for a given employee ID by a given percentage
- Sort the employees based on salary (ascending) and display them
- Sort the employees based on name (alphabetical) and display them

  Demonstrate all these operations from the main function.