Chapter 6: The List ADT and Java Collections

Data Structures

The List ADT | Iterable and Collection | The List interface | Iterator and ListIterator | ArrayList | LinkedList | Positional lists | Vector and Stack | CopyOnWriteArrayList | Arrays.asList

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/data_structures/slides/Chapter_06_The_List_ADT_and_Java_Collections.html

INTRODUCTION

Learning Outcomes

The List ADT

Explain the basic operations of an ordered collection.

Traversal

Know when to use `Iterator`, `ListIterator`, and `Spliterator`.

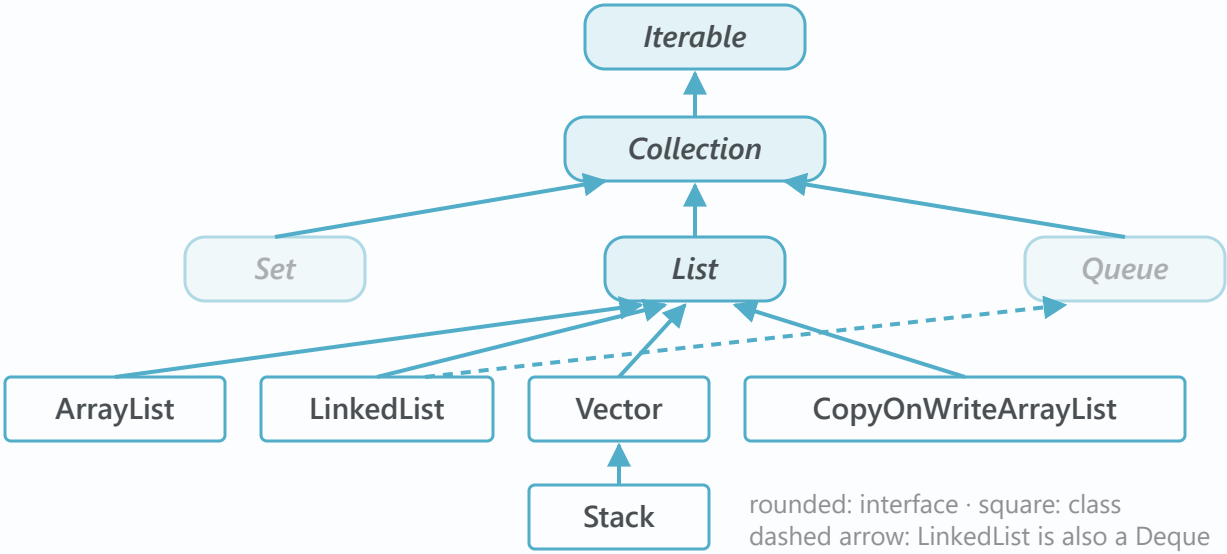
Implementations

Compare `ArrayList` and `LinkedList`.

Choosing an API

Pick the collection and method that fit the operations you need.

Java Collections Framework: The List Side



- An **interface** says what can be done; a **class** says how: List ADT ↔ data structure.
- Code written against the interface type can switch implementations easily: `List<String> l = new ArrayList<>();`

The Iterable Interface

At the top of the hierarchy, it lets us **traverse** an object's elements. If an object is `Iterable`, we can:

iterator()
get an `Iterator` that visits the elements in order.

spliterator()
get a `Spliterator` that splits the elements into parts for parallel processing.

forEach(...)
run an action (a lambda) on every element.

```
// what we write
for (var x : a) {
    System.out.println(x);
}
```

```
// what the compiler generates (a is Iterabl
Iterator<T> it = a.iterator();
while (it.hasNext()) {
    var x = it.next();
    System.out.println(x);
}
```

A for-each loop compiles only if `a` is an array or an object of a class that implements `Iterable`. (For arrays it becomes an indexed loop.)

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
▶    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
7        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`for (int x : new Range(2, 5))` : the constructor runs first; the range is [2, 5).

Step 1 / 10

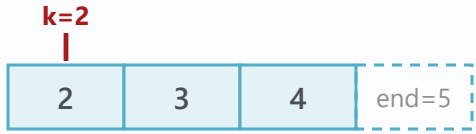
VARIABLES

`start = 2` `end = 5` `k = –` `x = –`

OUTPUT

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
▶    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
▶        int k = start;
7        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

for-each quietly calls `iterator()` : an anonymous inner class, `k = start` .

Step 2 / 10

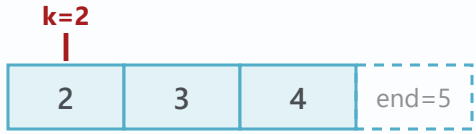
VARIABLES

`start = 2` `end = 5` `k = 2` `x = –`

OUTPUT

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
▶        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`hasNext()` : 2 < 5 → **true**

Step 3 / 10

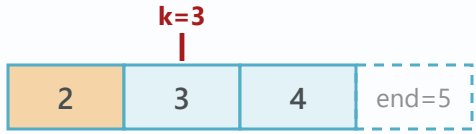
VARIABLES

`start = 2` `end = 5` `k = 2` `x = –`

OUTPUT

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
7        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
▶           return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`next()` : returns 2, then k++ → 3. The body prints `x = 2` .

Step 4 / 10

VARIABLES

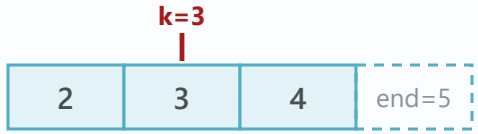
`start = 2` `end = 5` `k = 3` `x = 2`

OUTPUT

```
2
```

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
▶        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`hasNext()` : 3 < 5 → **true**

Step 5 / 10

VARIABLES

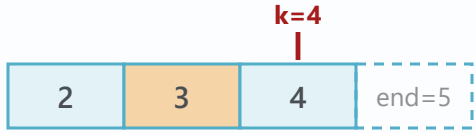
`start` = **2** `end` = **5** `k` = **3** `x` = **2**

OUTPUT

```
2
```

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
7        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
▶           return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`next()` : returns 3, then k++ → 4. The body prints `x = 3` .

Step 6 / 10

VARIABLES

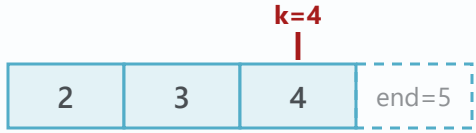
`start = 2` `end = 5` `k = 4` `x = 3`

OUTPUT

```
2
3
```

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
▶        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`hasNext()` : 4 < 5 → **true**

Step 7 / 10

VARIABLES

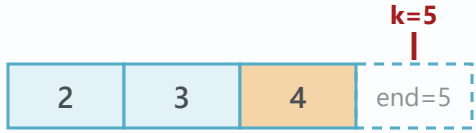
`start = 2` `end = 5` `k = 4` `x = 3`

OUTPUT

```
2
3
```

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
7        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
▶           return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`next()` : returns 4, then k++ → 5. The body prints `x = 4`.

Step 8 / 10

VARIABLES

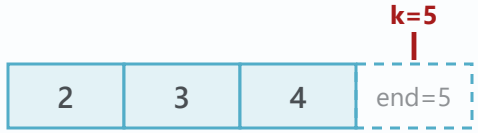
`start = 2` `end = 5` `k = 5` `x = 4`

OUTPUT

```
2
3
4
```

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
▶        public boolean hasNext() { return k < end; }
8        public Integer next() {
9          if (k >= end)
10           throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

`hasNext()` : 5 < 5 → **false**; the loop ends.

Step 9 / 10

VARIABLES

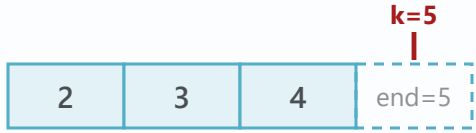
`start = 2` `end = 5` `k = 5` `x = 4`

OUTPUT

```
2
3
4
```

Code Trace: Our Own `Iterable` Class

```
1  class Range implements Iterable<Integer> {
2    private final int start, end;
3    Range(int s, int e) { start = s; end = e; }
4    public Iterator<Integer> iterator() {
5      return new Iterator<Integer>() {
6        int k = start;
7        public boolean hasNext() { return k < end; }
8        public Integer next() {
▶          if (k >= end)
▶            throw new NoSuchElementException();
11         return k++;
12       }
13     };
14   }
15 }
```



new Range(2, 5) · orange: last returned

The contract: `next()` with no elements left → `NoSuchElementException`.

Step 10 / 10

VARIABLES

`start = 2` `end = 5` `k = 5` `x = 4`

OUTPUT

```
2
3
4
```

The Collection Interface

It standardizes what every collection (list, set, queue) can do:

Capability	Methods	Example
Adding / removing elements	`add(e)`, `remove(o)`, `clear()`	`c.add("Ann")` → true
Size and emptiness	`size()`, `isEmpty()`	`c.size()` → 1
Membership test	`contains(o)`	`c.contains("Ann")` → true
Stream processing	`stream()`	`c.stream().count()`
Copying to an array	`toArray()`	`c.toArray(new String[0])`
Bulk operations	`addAll`, `removeAll`, `removeIf`	`c.removeIf(s -> s.isEmpty())`

A Collection has **no indices**: position-based operations such as `get(i)` come with the List interface.

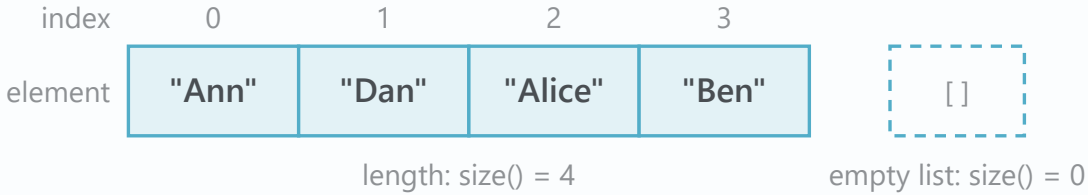
The List Interface

- Part of the Java Collections Framework, in the `java.util` package.
- An **ordered** collection (a *sequence*): elements stay where they were inserted.
- The user decides where each element goes and reaches it by an **integer index**, starting at 0.
- Allows duplicates and, in most implementations, `null`.
- Supports searching: `indexOf`, `contains`.



List is an **interface**, so `new List()` won't compile; we use a class that implements it (`ArrayList`, `LinkedList`, ...).

List Terminology



Element
A piece of data stored in the list.

Index
The number giving an element's position: 0 … length − 1.

Empty list
A list with no elements: `isEmpty()` → true.

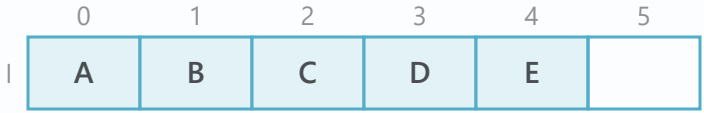
Length
The number of elements: `size()`. Not to be confused with capacity.

List Operations: The ADT and Java

ADT operation	Meaning	Java `List` method
Append	adds an element at the end	`add(e)`
Insert	adds at a given position; later elements shift	`add(i, e)`
Remove	removes the element at a position, or a given element	`remove(i)`, `remove(o)`
Indexing	reads / replaces the element at a position	`get(i)`, `set(i, e)`
Slicing	selects a range [start, end)	`subList(from, to)`
Search	finds an element's position	`indexOf(o)` → −1 if absent
Length	returns the number of elements	`size()`

`subList` is not a copy but a **view** of the original list: `l.subList(1, 3).clear()` removes the elements at indices 1 and 2 from the original.

Code Trace: `subList` Is a View



An ArrayList with 5 elements.

Step 1 / 8

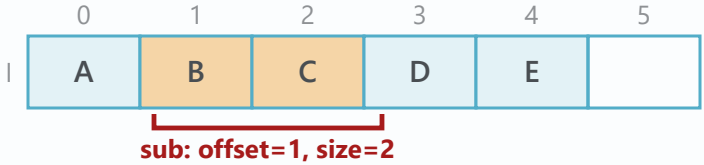
```
▶ List<String> l = new ArrayList<>(
▶     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
5 sub.set(0, "X");
6 sub.add("Y");
7 sub.clear();
8 System.out.println(l);
```

VARIABLES

`l = [A, B, C, D, E]` `sub = –` `offset = –` `size = –`

OUTPUT

Code Trace: `subList` Is a View



`subList(1, 3)` makes no copy: it is a view that stores only a **start (offset) and a length**. [1, 3) → B, C.

Step 2 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
▶ List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
5 sub.set(0, "X");
6 sub.add("Y");
7 sub.clear();
8 System.out.println(l);
```

VARIABLES

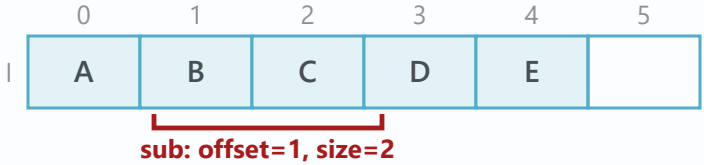
`l = [A, B, C, D, E]`

`sub = [B, C]` `offset = 1`

`size = 2`

OUTPUT

Code Trace: `subList` Is a View



Printing the view reads indices 1 and 2 of the original list.

Step 3 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
▶ System.out.println(sub);
5 sub.set(0, "X");
6 sub.add("Y");
7 sub.clear();
8 System.out.println(l);
```

VARIABLES

`l = [A, B, C, D, E]`

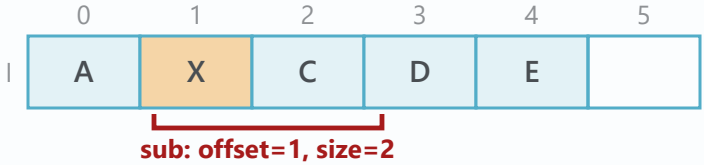
`sub = [B, C]` `offset = 1`

`size = 2`

OUTPUT

```
[B, C]
```

Code Trace: `subList` Is a View



`sub.set(0, …)` → `l.set(offset + 0, …)` on the original: l[1] = X.

Step 4 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
▶ sub.set(0, "X");
6 sub.add("Y");
7 sub.clear();
8 System.out.println(l);
```

VARIABLES

`l = [A, X, C, D, E]`

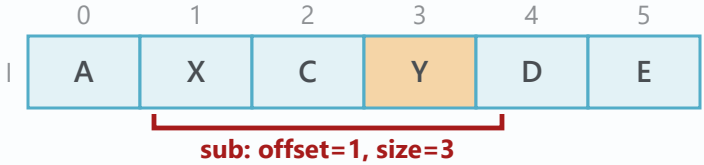
`sub = [X, C]` `offset = 1`

`size = 2`

OUTPUT

```
[B, C]
```

Code Trace: `subList` Is a View



`sub.add("Y")` appends to the view, which means inserting at index 3 of l; D and E shift right and the view grows.

Step 5 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
5 sub.set(0, "X");
▶ sub.add("Y");
7 sub.clear();
8 System.out.println(l);
```

VARIABLES

`l = [A, X, C, Y, D, E]`

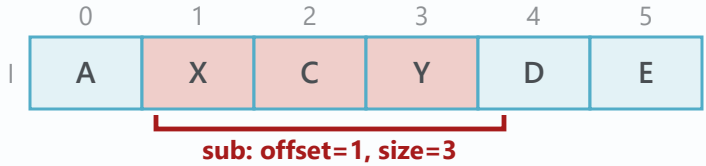
`sub = [X, C, Y]` `offset = 1`

`size = 3`

OUTPUT

```
[B, C]
```

Code Trace: `subList` Is a View



`sub.clear()` removes the view's 3 elements (indices 1–3) **from the original list**...

Step 6 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
5 sub.set(0, "X");
6 sub.add("Y");
▶ sub.clear();
8 System.out.println(l);
```

VARIABLES

`l = [A, X, C, Y, D, E]`

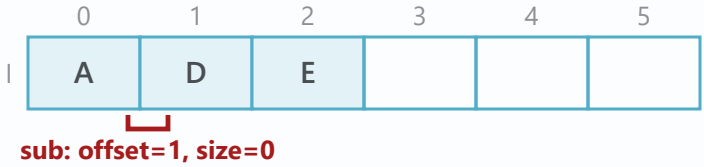
`sub = [X, C, Y]` `offset = 1`

`size = 3`

OUTPUT

```
[B, C]
```

Code Trace: `subList` Is a View



...and D, E shift left. `l.subList(a, b).clear()` is the short way to delete a range.

Step 7 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
5 sub.set(0, "X");
6 sub.add("Y");
▶ sub.clear();
8 System.out.println(l);
```

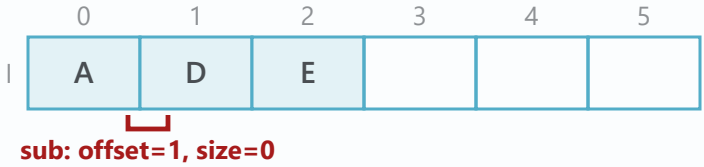
VARIABLES

`l = [A, D, E]` `sub = []` `offset = 1` `size = 0`

OUTPUT

```
[B, C]
```

Code Trace: `subList` Is a View



The result is [A, D, E]. Careful: if l itself is structurally modified (add/remove), the old `sub` becomes invalid → `ConcurrentModificationException`.

Step 8 / 8

```
1 List<String> l = new ArrayList<>(
2     List.of("A", "B", "C", "D", "E"));
3 List<String> sub = l.subList(1, 3);
4 System.out.println(sub);
5 sub.set(0, "X");
6 sub.add("Y");
7 sub.clear();
▶ System.out.println(l);
```

VARIABLES

`l = [A, D, E]` `sub = []` `offset = 1` `size = 0`

OUTPUT

```
[B, C]
[A, D, E]
```

Mutable and Immutable Lists

Mutable

Elements can be added, removed, and replaced after creation.

`new ArrayList<>(), new LinkedList<>()`

Immutable

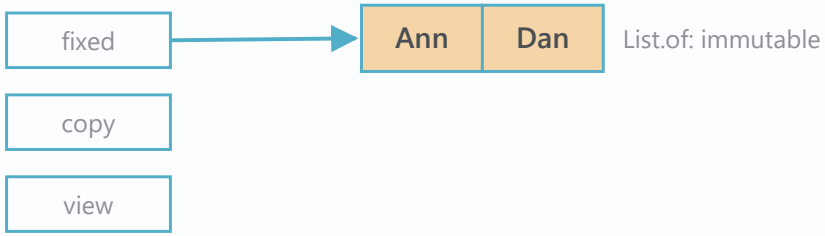
The contents never change after creation; to "change" it, we build a new list.

`List.of(...), List.copyOf(...)`

```
List<String> fixed = List.of("Ann", "Dan");
fixed.add("Ben");    // UnsupportedOperationException
List<String> copy = new ArrayList<>(fixed);   // mutable copy
copy.add("Ben");     // [Ann, Dan, Ben]; fixed is still [Ann, Dan]
```

- Immutability is a core idea of functional programming: immutable data is safe to share and to use in parallel.
- **Persistent data structures** are immutable; each "update" keeps the old version and avoids a full copy by **sharing** the unchanged parts.
- `Collections.unmodifiableList(l)` is only a read-only *view*: if the underlying `l` changes, so does the view.

Code Trace: Copy, View, or Immutable?



`List.of` builds an immutable list (and rejects `null`).

Step 1 / 8

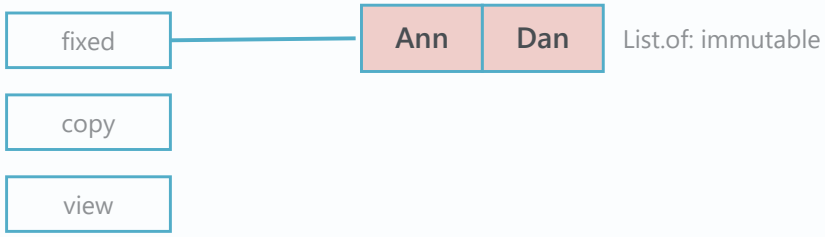
```
▶ List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
9 System.out.println(view);
```

VARIABLES

`fixed = [Ann, Dan]` `copy = –` `view = –`

OUTPUT

Code Trace: Copy, View, or Immutable?



If uncommented: structural changes are forbidden → `UnsupportedOperationException` (a runtime error, not a compile error).

Step 2 / 8

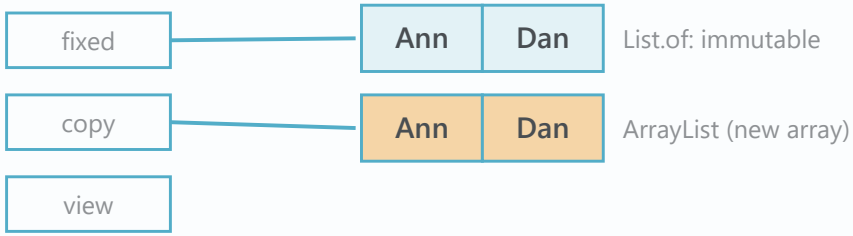
```
1 List<String> fixed = List.of("Ann", "Dan");
▶ // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
9 System.out.println(view);
```

VARIABLES

`fixed = [Ann, Dan]` `copy = –` `view = –`

OUTPUT

Code Trace: Copy, View, or Immutable?



`new ArrayList<>(fixed)` **copies the elements into a new array**; now there are two separate objects.

Step 3 / 8

```
1 List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
▶ List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
9 System.out.println(view);
```

VARIABLES

`fixed = [Ann, Dan]`

`copy = [Ann, Dan]` `view = –`

OUTPUT

Code Trace: Copy, View, or Immutable?



The copy is mutable, so Ben is added. `fixed` is still [Ann, Dan].

Step 4 / 8

```
1 List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
▶ copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
9 System.out.println(view);
```

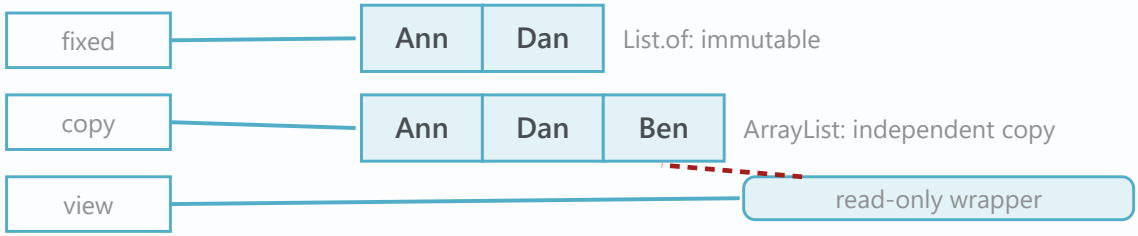
VARIABLES

`fixed = [Ann, Dan]`

`copy = [Ann, Dan, Ben]` `view = –`

OUTPUT

Code Trace: Copy, View, or Immutable?



`unmodifiableList` copies nothing: it returns a **view** that wraps `copy` and refuses writes.

Step 5 / 8

```
1 List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
▶ List<String> view =
▶     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
9 System.out.println(view);
```

VARIABLES

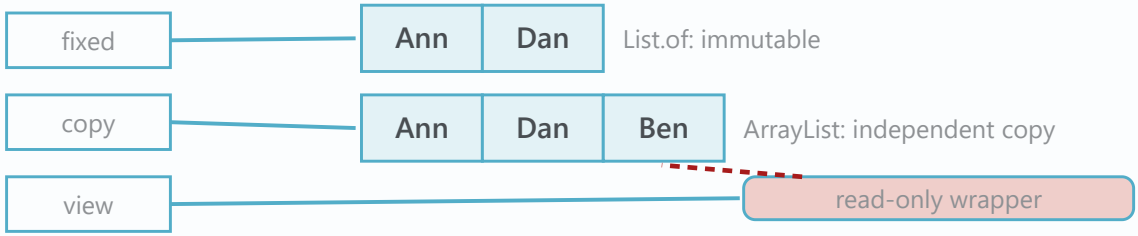
`fixed = [Ann, Dan]`

`copy = [Ann, Dan, Ben]`

`view = [Ann, Dan, Ben]`

OUTPUT

Code Trace: Copy, View, or Immutable?



If uncommented: writing through the view is forbidden → `UnsupportedOperationException`.

Step 6 / 8

```
1 List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
▶ // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
9 System.out.println(view);
```

VARIABLES

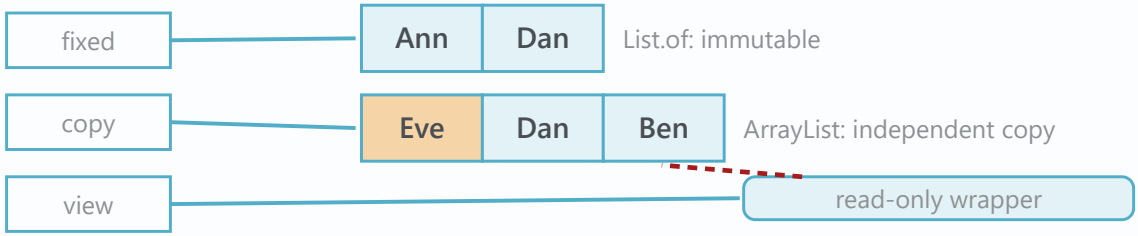
`fixed = [Ann, Dan]`

`copy = [Ann, Dan, Ben]`

`view = [Ann, Dan, Ben]`

OUTPUT

Code Trace: Copy, View, or Immutable?



But the underlying `copy` can change, and since the view looks at the same array, it sees the change.

Step 7 / 8

```
1 List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
▶ copy.set(0, "Eve");
9 System.out.println(view);
```

VARIABLES

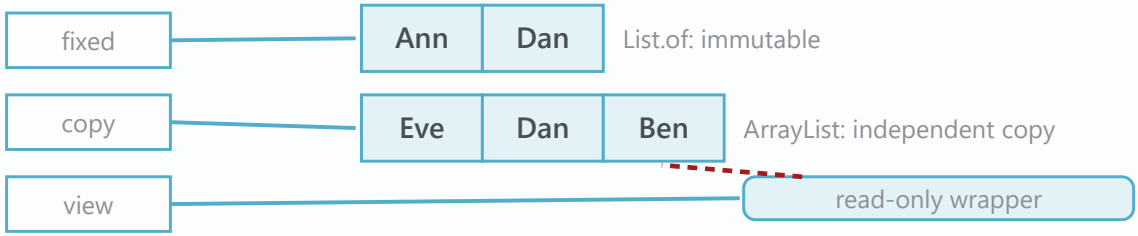
`fixed = [Ann, Dan]`

`copy = [Eve, Dan, Ben]`

`view = [Eve, Dan, Ben]`

OUTPUT

Code Trace: Copy, View, or Immutable?



Read-only ≠ immutable. For true immutability, use `List.copyOf(copy)`.

Step 8 / 8

```
1 List<String> fixed = List.of("Ann", "Dan");
2 // fixed.add("Ben");      → error
3 List<String> copy = new ArrayList<>(fixed);
4 copy.add("Ben");
5 List<String> view =
6     Collections.unmodifiableList(copy);
7 // view.set(0, "Eve"); → error
8 copy.set(0, "Eve");
▶ System.out.println(view);
```

VARIABLES

`fixed = [Ann, Dan]`

`copy = [Eve, Dan, Ben]`

`view = [Eve, Dan, Ben]`

OUTPUT

```
[Eve, Dan, Ben]
```

LIST METHODS

Main Methods of the List Interface

Group	Method	Description
Adding	`add(E e)`	appends; returns `true`
	`add(int index, E e)`	inserts at the index; later elements shift one place right
Removing	`remove(Object o)`	removes the **first** occurrence; `true` if found
	`remove(int index)`	removes the element at the index and **returns** it
Access	`get(int index)`	returns the element at the index
	`set(int index, E e)`	replaces the element; returns the old one
Length	`size()`	number of elements

An invalid index (`index < 0` or `index ≥ size()` ; for `add` , `> size()`) → `IndexOutOfBoundsException` . Loops traverse the elements (→ Traversal).

LIST METHODS

ListDemo: Duplicates and null

```
public class ListDemo {   // import java.util.ArrayList; import java.util.List;
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Ann");
        names.add("Ann");            // duplicate: allowed
        names.add(null);             // null: allowed
        names.add("Ann");
        System.out.println(names);          // [Ann, Ann, null, Ann]
        System.out.println(names.size());   // 4
        System.out.println(names.indexOf("Ann"));      // 0 (first occurrence)
        System.out.println(names.lastIndexOf("Ann"));  // 3
    }
}
```

List ≠ Set

A list allows duplicates; for unique elements, use a `Set`.

Careful

A list built with `List.of(...)` rejects `null`.

LIST METHODS · STEP THROUGH THE CODE

Code Trace: Duplicates, `null`, and Search



An empty ArrayList.

Step 1 / 12

```
▶ List<String> names = new ArrayList<>();
2 names.add("Ann");
3 names.add("Ann");      // duplicate: allowed
4 names.add(null);       // null: allowed
5 names.add("Ann");
6 System.out.println(names);
7 System.out.println(names.size());
8 System.out.println(names.indexOf("Ann"));
9 System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

`names = []` `size = 0`

OUTPUT

Code Trace: Duplicates, `null`, and Search



Appended at index 0.

Step 2 / 12

```
1  List<String> names = new ArrayList<>();
▶  names.add("Ann");
3  names.add("Ann");      // duplicate: allowed
4  names.add(null);       // null: allowed
5  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

`names = [Ann]` `size = 1`

OUTPUT

Code Trace: Duplicates, `null`, and Search



The same value goes in again: a list (unlike a set) **allows duplicates**.

Step 3 / 12

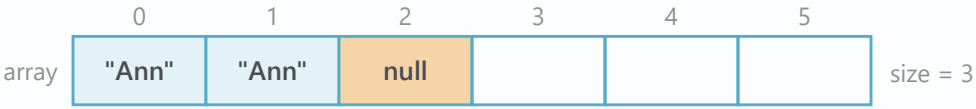
```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
▶  names.add("Ann");      // duplicate: allowed
4  names.add(null);       // null: allowed
5  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

`names = [Ann, Ann]` `size = 2`

OUTPUT

Code Trace: Duplicates, `null`, and Search



ArrayList stores `null` too.

Step 4 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
▶  names.add(null);        // null: allowed
5  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

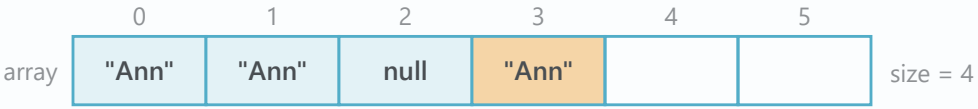
VARIABLES

`names = [Ann, Ann, null]`

`size = 3`

OUTPUT

Code Trace: Duplicates, `null`, and Search



The third "Ann" goes to index 3.

Step 5 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
4  names.add(null);        // null: allowed
▶  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

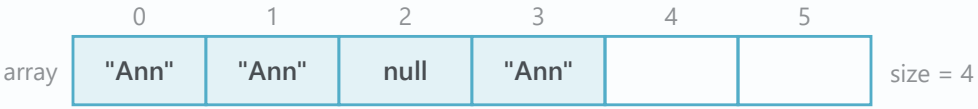
VARIABLES

`names = [Ann, Ann, null, Ann]`

`size = 4`

OUTPUT

Code Trace: Duplicates, `null`, and Search



`toString()` prints the elements in order, in square brackets.

Step 6 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
4  names.add(null);        // null: allowed
5  names.add("Ann");
▶  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

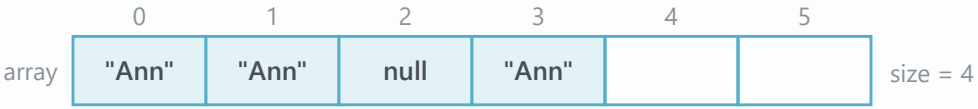
`names = [Ann, Ann, null, Ann]`

`size = 4`

OUTPUT

```
[Ann, Ann, null, Ann]
```

Code Trace: Duplicates, `null`, and Search



`size()` : the number of elements (not the capacity).

Step 7 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
4  names.add(null);        // null: allowed
5  names.add("Ann");
6  System.out.println(names);
▶  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

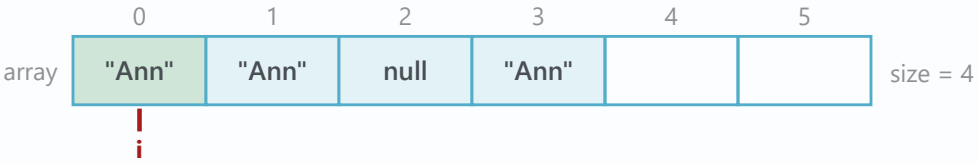
`names = [Ann, Ann, null, Ann]`

`size = 4`

OUTPUT

```
[Ann, Ann, null, Ann]
4
```

Code Trace: Duplicates, `null`, and Search



`indexOf` scans from the front and stops at the **first** match: `"Ann".equals(array[0])` → 0.

Step 8 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
4  names.add(null);        // null: allowed
5  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
▶  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

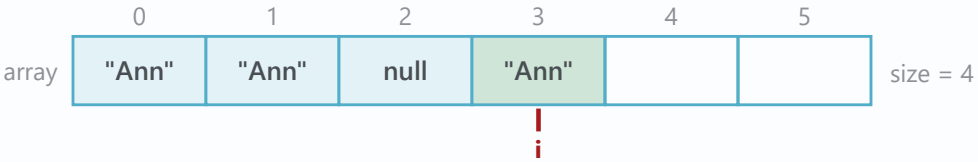
`names = [Ann, Ann, null, Ann]`

`size = 4` `i = 0` `result = 0`

OUTPUT

```
[Ann, Ann, null, Ann]
4
0
```

Code Trace: Duplicates, `null`, and Search



`lastIndexOf` scans from the back: the first index it checks, 3, matches → 3.

Step 9 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
4  names.add(null);        // null: allowed
5  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
▶  System.out.println(names.lastIndexOf("Ann"));
10 System.out.println(names.indexOf(null));
```

VARIABLES

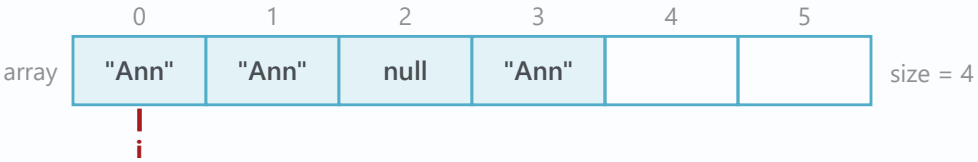
`names = [Ann, Ann, null, Ann]`

`size = 4` `i = 3` `result = 3`

OUTPUT

```
[Ann, Ann, null, Ann]
4
0
3
```

Code Trace: Duplicates, `null`, and Search



`indexOf(null)`: since the target is null, it uses `==` instead of `equals`. array[0] == null? No.

Step 10 / 12

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Ann");       // duplicate: allowed
4  names.add(null);        // null: allowed
5  names.add("Ann");
6  System.out.println(names);
7  System.out.println(names.size());
8  System.out.println(names.indexOf("Ann"));
9  System.out.println(names.lastIndexOf("Ann"));
▶  System.out.println(names.indexOf(null));
```

VARIABLES

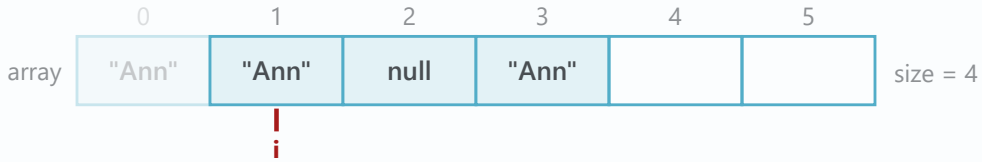
`names = [Ann, Ann, null, Ann]`

`size = 4` `i = 0` `result = –`

OUTPUT

```
[Ann, Ann, null, Ann]
4
0
3
```

Code Trace: Duplicates, `null`, and Search



array[1] == null? No.

Step 11 / 12

```
1 List<String> names = new ArrayList<>();
2 names.add("Ann");
3 names.add("Ann");      // duplicate: allowed
4 names.add(null);       // null: allowed
5 names.add("Ann");
6 System.out.println(names);
7 System.out.println(names.size());
8 System.out.println(names.indexOf("Ann"));
9 System.out.println(names.lastIndexOf("Ann"));
▶ System.out.println(names.indexOf(null));
```

VARIABLES

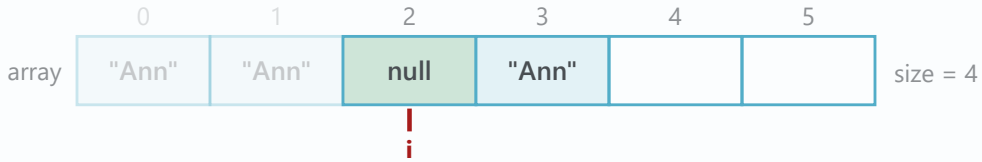
`names = [Ann, Ann, null, Ann]`

`size = 4` `i = 1` `result = –`

OUTPUT

```
[Ann, Ann, null, Ann]
4
0
3
```

Code Trace: Duplicates, `null`, and Search



array[2] == null? **Yes** → 2. (Calling null.equals would have thrown a NullPointerException.)

Step 12 / 12

```
1 List<String> names = new ArrayList<>();
2 names.add("Ann");
3 names.add("Ann");      // duplicate: allowed
4 names.add(null);       // null: allowed
5 names.add("Ann");
6 System.out.println(names);
7 System.out.println(names.size());
8 System.out.println(names.indexOf("Ann"));
9 System.out.println(names.lastIndexOf("Ann"));
▶ System.out.println(names.indexOf(null));
```

VARIABLES

`names = [Ann, Ann, null, Ann]`

`size = 4` `i = 2` `result = 2`

OUTPUT

```
[Ann, Ann, null, Ann]
4
0
3
2
```

A Trap: `remove(int)` or `remove(Object)`?

When a list holds `Integer`s, the two methods are easy to mix up: an `int` argument is taken as an **index**.

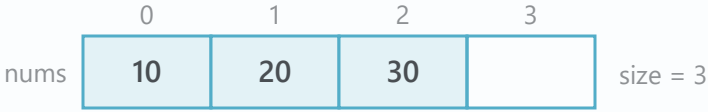
```
List<Integer> nums = new ArrayList<>(List.of(10, 20, 30));
nums.remove(1);                    // removes index 1 → [10, 30]
nums.remove(Integer.valueOf(10));  // removes the value 10 → [30]
nums.remove(Integer.valueOf(99));  // absent → false, no change
nums.remove(5);                    // IndexOutOfBoundsException
```

remove(int index)
Returns the removed element.
`E remove(int)`

remove(Object o)
Returns whether anything was removed.
`boolean remove(Object)`

Cost
In an array-based list, both shift the later elements.
`O(n)`

Code Trace: Which `remove` Is Called?



[10, 20, 30]. On a `List<Integer>`, both `remove` overloads apply.

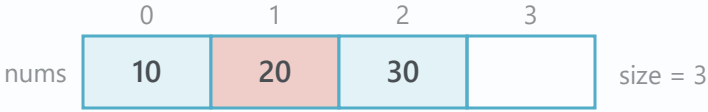
Step 1 / 7

```
▶ List<Integer> nums =
▶     new ArrayList<>(List.of(10, 20, 30));
3 Integer a = nums.remove(1);
4 boolean b = nums.remove(Integer.valueOf(10));
5 boolean c = nums.remove(Integer.valueOf(99));
6 nums.remove(5);   // IndexOutOfBoundsException
```

VARIABLES

`nums = [10, 20, 30]` `a = –` `b = –` `c = –`

Code Trace: Which `remove` Is Called?



The argument is an `int`, so the compiler picks **`remove(int index)`**, which needs no boxing. Index 1 goes.

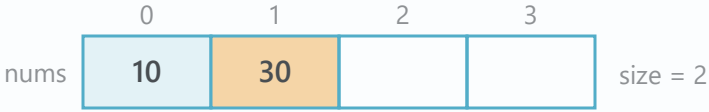
Step 2 / 7

```
1 List<Integer> nums =
2     new ArrayList<>(List.of(10, 20, 30));
▶ Integer a = nums.remove(1);
4 boolean b = nums.remove(Integer.valueOf(10));
5 boolean c = nums.remove(Integer.valueOf(99));
6 nums.remove(5);    // IndexOutOfBoundsException
```

VARIABLES

`nums = [10, 20, 30]` `a = –` `b = –` `c = –`

Code Trace: Which `remove` Is Called?



30 shifts one place left; the removed element is **returned**: a = 20.

Step 3 / 7

```
1 List<Integer> nums =
2     new ArrayList<>(List.of(10, 20, 30));
▶ Integer a = nums.remove(1);
4 boolean b = nums.remove(Integer.valueOf(10));
5 boolean c = nums.remove(Integer.valueOf(99));
6 nums.remove(5);   // IndexOutOfBoundsException
```

VARIABLES

`nums = [10, 30]` `a = 20` `b = –` `c = –`

Code Trace: Which `remove` Is Called?



An `Integer` argument → `remove(Object o)`: search by value.
`Integer.valueOf(10).equals(10)` → a match at index 0.

Step 4 / 7

```
1 List<Integer> nums =
2     new ArrayList<>(List.of(10, 20, 30));
3 Integer a = nums.remove(1);
▶ boolean b = nums.remove(Integer.valueOf(10));
5 boolean c = nums.remove(Integer.valueOf(99));
6 nums.remove(5);   // IndexOutOfBoundsException
```

VARIABLES

`nums = [10, 30]` `a = 20` `b = –` `c = –`

Code Trace: Which `remove` Is Called?



The first match is removed and 30 shifts; the result is a **boolean**: b = true.

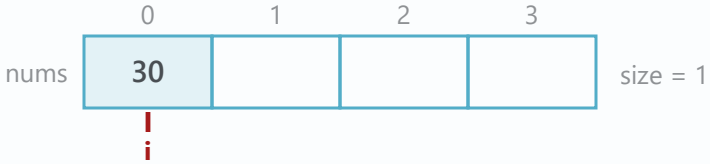
Step 5 / 7

```
1 List<Integer> nums =
2     new ArrayList<>(List.of(10, 20, 30));
3 Integer a = nums.remove(1);
▶ boolean b = nums.remove(Integer.valueOf(10));
5 boolean c = nums.remove(Integer.valueOf(99));
6 nums.remove(5);   // IndexOutOfBoundsException
```

VARIABLES

`nums = [30]` `a = 20` `b = true` `c = –`

Code Trace: Which `remove` Is Called?



Searching for 99: 30 ≠ 99, end of list → not found, no change, c = false.

Step 6 / 7

```
1 List<Integer> nums =
2     new ArrayList<>(List.of(10, 20, 30));
3 Integer a = nums.remove(1);
4 boolean b = nums.remove(Integer.valueOf(10));
▶ boolean c = nums.remove(Integer.valueOf(99));
6 nums.remove(5);   // IndexOutOfBoundsException
```

VARIABLES

`nums = [30]` `a = 20` `b = true` `c = false`

Code Trace: Which `remove` Is Called?



`remove(5)` : the index check 5 < size (1) fails → `IndexOutOfBoundsException` : "Index 5 out of bounds for length 1".

Step 7 / 7

```
1 List<Integer> nums =
2     new ArrayList<>(List.of(10, 20, 30));
3 Integer a = nums.remove(1);
4 boolean b = nums.remove(Integer.valueOf(10));
5 boolean c = nums.remove(Integer.valueOf(99));
▶ nums.remove(5);   // IndexOutOfBoundsException
```

VARIABLES

`nums = [30]` `a = 20` `b = true` `c = false`

LIST METHODS

Bulk Operations with Lambdas: `replaceAll` and `sort`

replaceAll

```
List<String> names =
        new ArrayList<>();
names.add("Ann");
names.add("Tom");
names.add("Bob");
names.replaceAll(
        name -> name.toUpperCase());
System.out.println(names);
// [ANN, TOM, BOB]
```

Each element is replaced **in place** by the function's result.

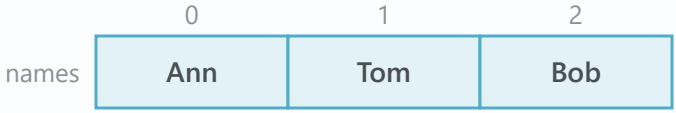
sort

```
List<Integer> nums =
        new ArrayList<>();
nums.add(5);
nums.add(3);
nums.add(1);
nums.sort((s1, s2) ->
        Integer.compare(s1, s2));
System.out.println(nums);
// [1, 3, 5]
```

Shorter: `nums.sort(null)` or `Comparator.naturalOrder()`; reverse order: `Comparator.reverseOrder()`.

Note: under a Turkish locale, `toUpperCase()` turns i into İ ("Kim" → "KİM"); for a locale-independent result, use `toUpperCase(Locale.ROOT)`.

Code Trace: `replaceAll` and `sort`



An ArrayList of three names.

Step 1 / 9

```
List<String> names = new ArrayList<>(
    List.of("Ann", "Tom", "Bob"));
names.replaceAll(
    name -> name.toUpperCase(Locale.ROOT));
List<Integer> nums = new ArrayList<>(
    List.of(5, 3, 1));
nums.sort((s1, s2) -> Integer.compare(s1, s2));
System.out.println(names + " " + nums);
```

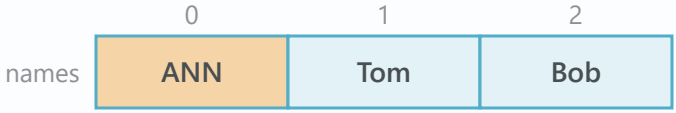
VARIABLES

`names = [Ann, Tom, Bob]`

`nums = –`

OUTPUT

Code Trace: `replaceAll` and `sort`



`replaceAll` calls the lambda for each index and writes the result **back in place**: array[0] = "ANN".

Step 2 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
▶ names.replaceAll(
▶     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
7 nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

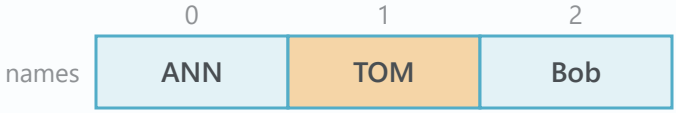
VARIABLES

`names = [ANN, Tom, Bob]`

`nums = –` `name = "Ann"`

OUTPUT

Code Trace: `replaceAll` and `sort`



`replaceAll` calls the lambda for each index and writes the result **back in place**: array[1] = "TOM".

Step 3 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
▶ names.replaceAll(
▶     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
7 nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

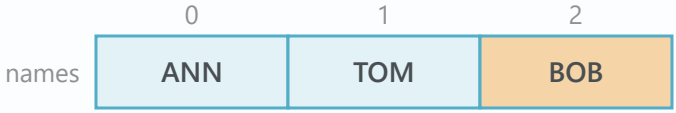
VARIABLES

`names = [ANN, TOM, Bob]`

`nums = –` `name = "Tom"`

OUTPUT

Code Trace: `replaceAll` and `sort`



`replaceAll` calls the lambda for each index and writes the result **back in place**: array[2] = "BOB".

Step 4 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
▶ names.replaceAll(
▶     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
7 nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

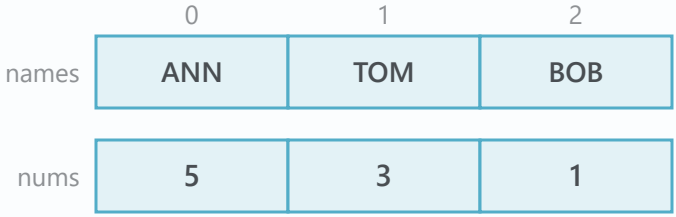
VARIABLES

`names = [ANN, TOM, BOB]`

`nums = –` `name = "Bob"`

OUTPUT

Code Trace: `replaceAll` and `sort`



[5, 3, 1]. Java's `sort` uses TimSort; our lambda does the comparing.

Step 5 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
3 names.replaceAll(
4     name -> name.toUpperCase(Locale.ROOT));
▶ List<Integer> nums = new ArrayList<>(
▶     List.of(5, 3, 1));
7 nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

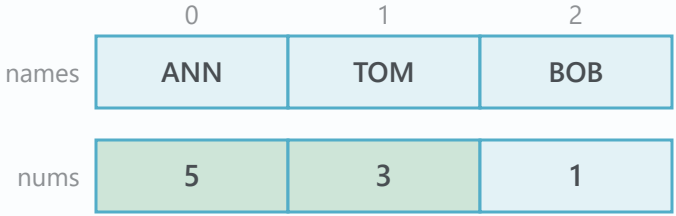
VARIABLES

`names = [ANN, TOM, BOB]`

`nums = [5, 3, 1]`

OUTPUT

Code Trace: `replaceAll` and `sort`



The first call, `compare(3, 5)` → -1: the array opens with a **descending** run.

Step 6 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
3 names.replaceAll(
4     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
▶ nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

VARIABLES

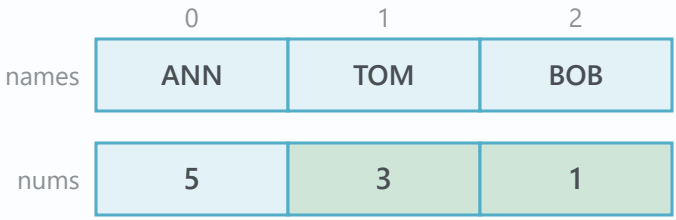
`names = [ANN, TOM, BOB]`

`nums = [5, 3, 1]` `s1 = 3`

`s2 = 5` `result = -1`

OUTPUT

Code Trace: `replaceAll` and `sort`



`compare(1, 3)` → -1: the descending run continues; its length is 3.

Step 7 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
3 names.replaceAll(
4     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
▶ nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

VARIABLES

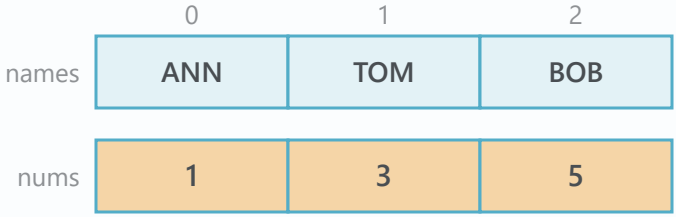
`names = [ANN, TOM, BOB]`

`nums = [5, 3, 1]` `s1 = 1`

`s2 = 3` `result = -1`

OUTPUT

Code Trace: `replaceAll` and `sort`



The whole array is one descending run, so TimSort **reverses** it → [1, 3, 5]. Only 2 comparisons.

Step 8 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
3 names.replaceAll(
4     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
▶ nums.sort((s1, s2) -> Integer.compare(s1, s2));
8 System.out.println(names + " " + nums);
```

VARIABLES

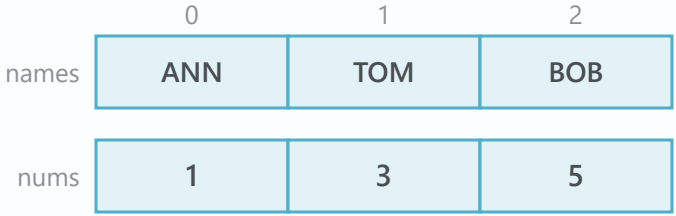
`names = [ANN, TOM, BOB]`

`nums = [1, 3, 5]` `s1 = –`

`s2 = –` `result = –`

OUTPUT

Code Trace: `replaceAll` and `sort`



With Locale.ROOT the result is the same everywhere (under a Turkish locale, an i would become İ).

Step 9 / 9

```
1 List<String> names = new ArrayList<>(
2     List.of("Ann", "Tom", "Bob"));
3 names.replaceAll(
4     name -> name.toUpperCase(Locale.ROOT));
5 List<Integer> nums = new ArrayList<>(
6     List.of(5, 3, 1));
7 nums.sort((s1, s2) -> Integer.compare(s1, s2));
▶ System.out.println(names + " " + nums);
```

VARIABLES

`names = [ANN, TOM, BOB]`

`nums = [1, 3, 5]`

OUTPUT

```
[ANN, TOM, BOB] [1, 3, 5]
```

Indexed Access and Generics

```
List<String> names =
        new ArrayList<>();
names.add("Ann");
names.add("Tom");
names.add("Bob");
String firstName = names.get(0);
System.out.println(firstName);  // Ann
names.get(3);  // IndexOutOfBounds
```

```
List<String> names =
        new ArrayList<>();
names.add("Ann");
names.add(5);   // compile error

List<Integer> nums =
        new ArrayList<>();
nums.add(5);   // autoboxing
int x = nums.get(0);  // unboxing
```

- Indices start at 0. The type parameter `<String>` catches a wrong type **at compile time**, and the result of `get` needs no cast.
- A type parameter cannot be primitive: not `List<int>` but `List<Integer>` (autoboxing).

Code Trace: `get`, Generics, and Boxing



Type parameter `<String>`: only Strings can go into this list (checked at compile time).

Step 1 / 10

```
▶ List<String> names = new ArrayList<>();
2 names.add("Ann");
3 names.add("Tom");
4 names.add("Bob");
5 String firstName = names.get(0);
6 System.out.println(firstName);
7 List<Integer> nums = new ArrayList<>();
8 nums.add(5);          // Integer.valueOf(5)
9 int x = nums.get(0);  // .intValue()
10 names.get(3);         // IndexOutOfBounds
```

VARIABLES

`names = []` `nums = –`

OUTPUT

Code Trace: `get`, Generics, and Boxing



"Ann" goes in at index 0.

Step 2 / 10

```
1  List<String> names = new ArrayList<>();
▶  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

VARIABLES

`names = [Ann]` `nums = –`

OUTPUT

Code Trace: `get`, Generics, and Boxing



"Tom" goes in at index 1.

Step 3 / 10

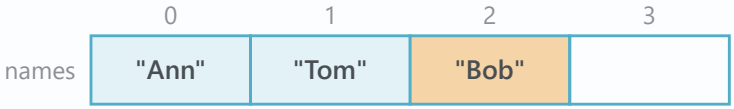
```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
▶  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

VARIABLES

`names = [Ann, Tom]` `nums = –`

OUTPUT

Code Trace: `get`, Generics, and Boxing



"Bob" goes in at index 2.

Step 4 / 10

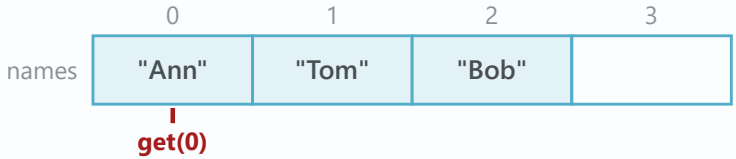
```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
▶  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

VARIABLES

`names = [Ann, Tom, Bob]` `nums = –`

OUTPUT

Code Trace: `get`, Generics, and Boxing



`get(0)`: the check 0 < size (3) passes and `array[0]` is read directly → O(1). No cast needed.

Step 5 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
▶  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

VARIABLES

`names = [Ann, Tom, Bob]` `nums = –`

`firstName = "Ann"`

OUTPUT

Code Trace: `get`, Generics, and Boxing



Prints Ann.

Step 6 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
▶  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

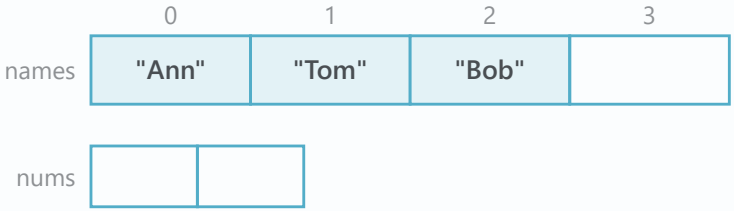
VARIABLES

`names = [Ann, Tom, Bob]` `nums = –` `firstName = "Ann"`

OUTPUT

```
Ann
```

Code Trace: `get`, Generics, and Boxing



`List<int>` is not allowed; we use the wrapper class `Integer`.

Step 7 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
▶  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

VARIABLES

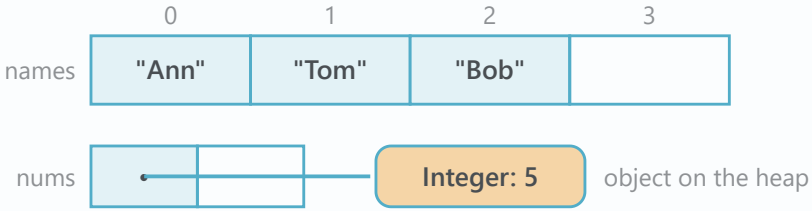
`names = [Ann, Tom, Bob]`

`nums = []` `firstName = "Ann"`

OUTPUT

```
Ann
```

Code Trace: `get`, Generics, and Boxing



Autoboxing: the compiler inserts `Integer.valueOf(5)`; the list holds a reference to the object.

Step 8 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
▶  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
10 names.get(3);          // IndexOutOfBounds
```

VARIABLES

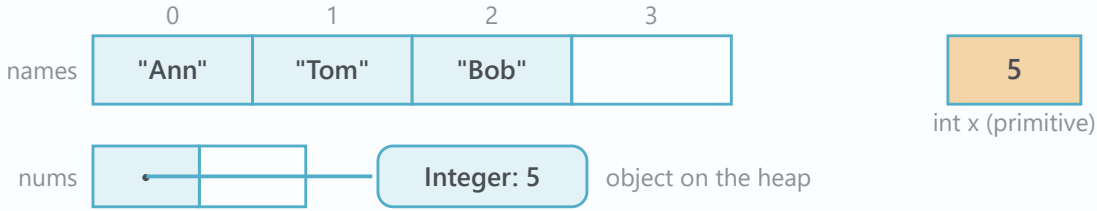
`names = [Ann, Tom, Bob]`

`nums = [5]` `firstName = "Ann"`

OUTPUT

```
Ann
```

Code Trace: `get`, Generics, and Boxing



Unboxing: `get(0).intValue()` → the primitive 5. (A null element would throw a NullPointerException.)

Step 9 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
▶  int x = nums.get(0);  // .intValue()
10 names.get(3);         // IndexOutOfBounds
```

VARIABLES

`names = [Ann, Tom, Bob]`

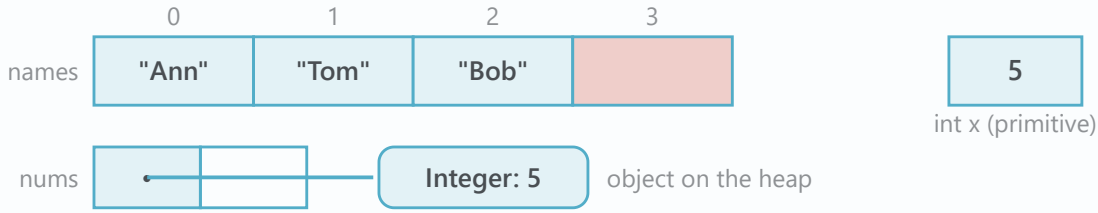
`nums = [5]` `firstName = "Ann"`

`x = 5`

OUTPUT

```
Ann
```

Code Trace: `get`, Generics, and Boxing



`get(3)` : 3 < size (3) is false → `IndexOutOfBoundsException`. Valid indices are 0 … size − 1.

Step 10 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann");
3  names.add("Tom");
4  names.add("Bob");
5  String firstName = names.get(0);
6  System.out.println(firstName);
7  List<Integer> nums = new ArrayList<>();
8  nums.add(5);          // Integer.valueOf(5)
9  int x = nums.get(0);  // .intValue()
▶  names.get(3);          // IndexOutOfBounds
```

VARIABLES

`names = [Ann, Tom, Bob]`

`nums = [5]` `firstName = "Ann"`

`x = 5`

OUTPUT

```
Ann
```

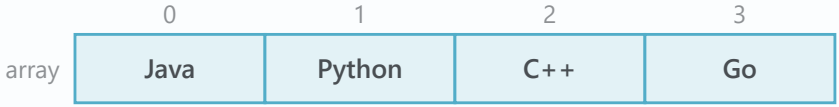
Spliterator: A Splittable Iterator

```
List<String> words = new ArrayList<>();
words.add("Java");
words.add("Python");
words.add("C++");
Spliterator<String> split =
        words.spliterator();
while (split.tryAdvance(
        System.out::println)) {
    // each call handles one element
}
```

- `tryAdvance(action)` : if there is a next element, applies the action and returns `true` ; otherwise `false` .
- `trySplit()` : hands part of the remaining elements to a new Spliterator, so the work can be shared among threads.
- Parallel streams (`parallelStream()`) use Spliterators behind the scenes.

```
Java
Python
C++
```

Code Trace: `trySplit` and `tryAdvance`



A list of 4 words.

Step 1 / 9

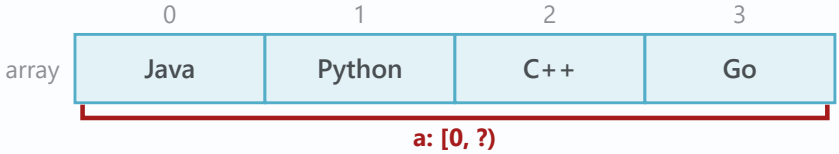
```
▶ List<String> words = new ArrayList<>(
▶     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
5 while (b.tryAdvance(System.out::println)) { }
6 while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

a = – b = –

OUTPUT

Code Trace: `trySplit` and `tryAdvance`



`spliterator()` : index 0, fence −1 → the bound is set from `size()` on first use (lazily).

Step 2 / 9

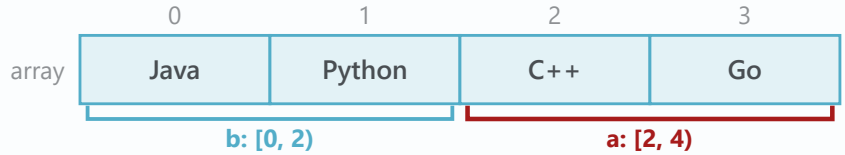
```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
▶ Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
5 while (b.tryAdvance(System.out::println)) { }
6 while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

`a = index=0, fence=-1` `b = –`

OUTPUT

Code Trace: `trySplit` and `tryAdvance`



`trySplit()` : fence = 4, mid = (0 + 4) / 2 = 2. The **first half** [0, 2) goes to the new `b` ; `a` keeps [2, 4).

Step 3 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
▶ Spliterator<String> b = a.trySplit();
5 while (b.tryAdvance(System.out::println)) { }
6 while (a.tryAdvance(System.out::println)) { }
```

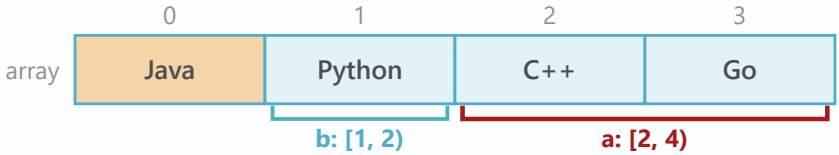
VARIABLES

`a = index=2, fence=4`

`b = index=0, fence=2`

OUTPUT

Code Trace: `trySplit` and `tryAdvance`



`b.tryAdvance`: an element is left → "Java" goes to the action (println), index moves to 1; true.

Step 4 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
▶ while (b.tryAdvance(System.out::println)) { }
6 while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

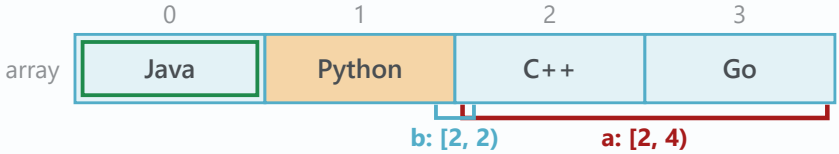
`a = index=2, fence=4`

`b = index=1, fence=2`

OUTPUT

```
Java
```

Code Trace: `trySplit` and `tryAdvance`



`b.tryAdvance`: an element is left → "Python" goes to the action (println), index moves to 2; true.

Step 5 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
▶ while (b.tryAdvance(System.out::println)) { }
6 while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

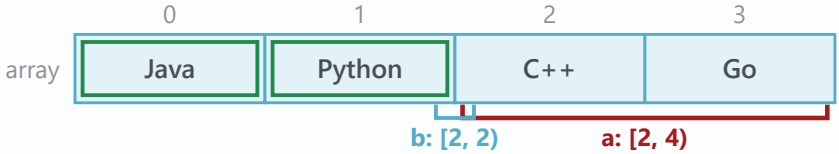
`a = index=2, fence=4`

`b = index=2, fence=2`

OUTPUT

```
Java
Python
```

Code Trace: `trySplit` and `tryAdvance`



`b.tryAdvance`: index = fence → nothing left, **false**; the loop ends.

Step 6 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
▶ while (b.tryAdvance(System.out::println)) { }
6 while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

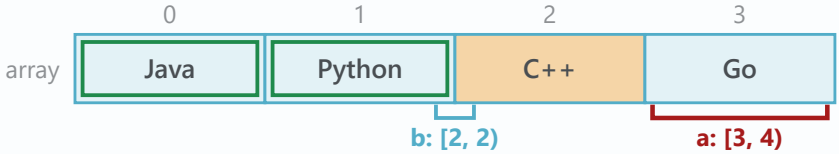
`a = index=2, fence=4`

`b = index=2, fence=2`

OUTPUT

```
Java
Python
```

Code Trace: `trySplit` and `tryAdvance`



`a.tryAdvance`: an element is left → "C++" goes to the action (println), index moves to 3; true.

Step 7 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
5 while (b.tryAdvance(System.out::println)) { }
▶ while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

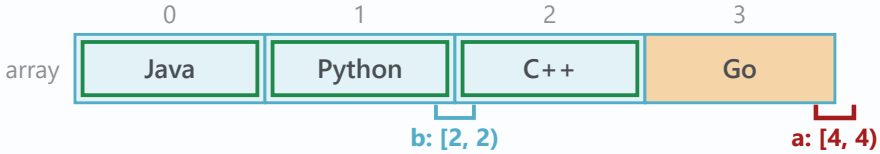
`a = index=3, fence=4`

`b = index=2, fence=2`

OUTPUT

```
Java
Python
C++
```

Code Trace: `trySplit` and `tryAdvance`



`a.tryAdvance`: an element is left → "Go" goes to the action (println), index moves to 4; true.

Step 8 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
5 while (b.tryAdvance(System.out::println)) { }
▶ while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

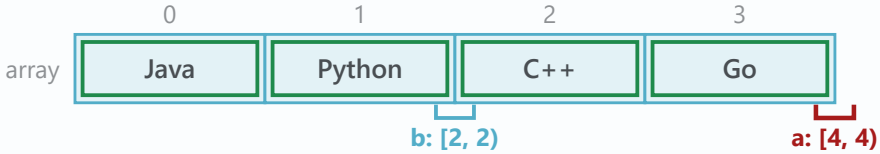
`a = index=4, fence=4`

`b = index=2, fence=2`

OUTPUT

```
Java
Python
C++
Go
```

Code Trace: `trySplit` and `tryAdvance`



`a.tryAdvance`: index = fence → nothing left, **false**; the loop ends. The two halves could just as well run on two threads.

Step 9 / 9

```
1 List<String> words = new ArrayList<>(
2     List.of("Java", "Python", "C++", "Go"));
3 Spliterator<String> a = words.spliterator();
4 Spliterator<String> b = a.trySplit();
5 while (b.tryAdvance(System.out::println)) { }
▶ while (a.tryAdvance(System.out::println)) { }
```

VARIABLES

`a = index=4, fence=4`

`b = index=2, fence=2`

OUTPUT

```
Java
Python
C++
Go
```

TRAVERSAL

Three Ways to Traverse a Collection

1 · indexed for

```
int n = list.size();
for (int i = 0;
     i < n; i++) {
  String s =
      list.get(i);
  ...
}
```

Lists only; when you also need the index.

2 · forEach

```
list.forEach(
  System.out::println);

// or for-each:
for (String s : list) ...
```

Any `Iterable`; the shortest form.

3 · Iterator

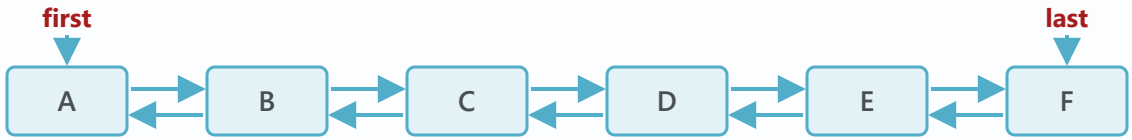
```
Iterator<String> it =
    list.iterator();
while (it.hasNext()) {
  String s = it.next();
}
```

When you need to remove safely while traversing.

Careful: on a `LinkedList`, `get(i)` is O(n), so the indexed loop takes **$O(n^2)$** in total; forEach / Iterator is **O(n)** on both lists.

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: The Cost of an Indexed Loop on LinkedList



A doubly linked list with 6 nodes.

Step 1 / 9

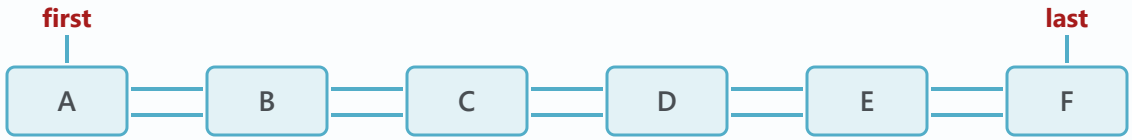
```
List<String> list = new LinkedList<>(
    List.of("A", "B", "C", "D", "E", "F"));
int n = list.size(), steps = 0;
for (int i = 0; i < n; i++) {
  String s = list.get(i);  // node(i) walks
  steps += Math.min(i, n - 1 - i);
}
System.out.println(steps);
```

VARIABLES

n = – i = – steps = – s = –

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`steps` counts how many links the `get` calls walk in total.

Step 2 / 9

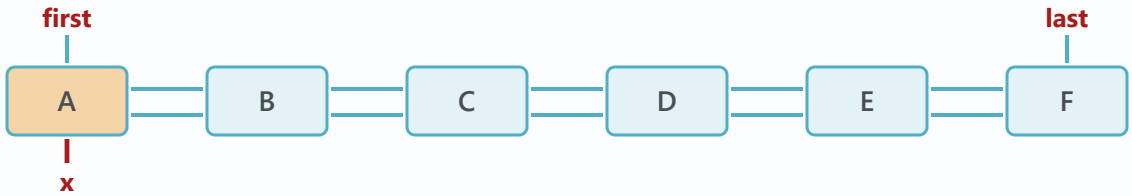
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
▶ int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
5   String s = list.get(i);  // node(i) walks
6   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = –` `steps = 0` `s = –`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`get(0)` : 0 < n/2 = 3 → walks from first, **0** links. Total 0.

Step 3 / 9

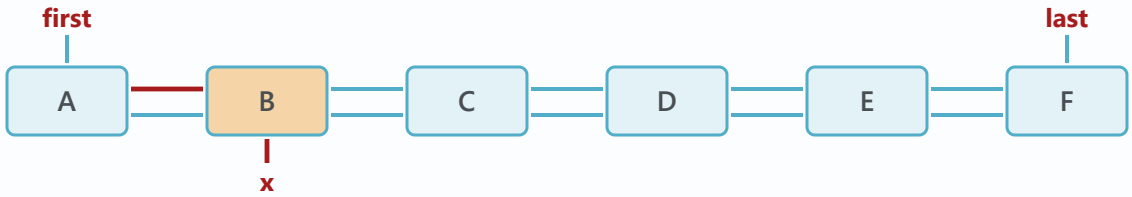
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
▶   String s = list.get(i);  // node(i) walks
▶   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = 0` `steps = 0` `s = "A"`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`get(1)` : 1 < n/2 = 3 → walks from first, **1** link. Total 1.

Step 4 / 9

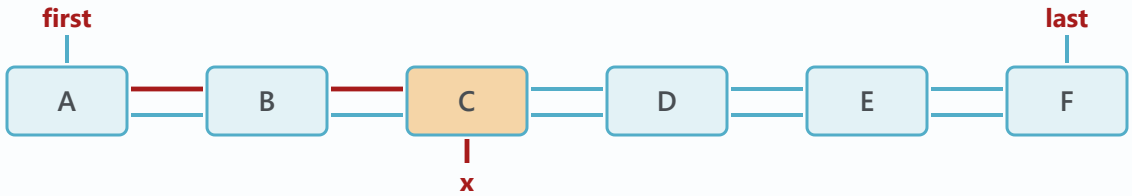
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
▶   String s = list.get(i);  // node(i) walks
▶   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = 1` `steps = 1` `s = "B"`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`get(2)` : 2 < n/2 = 3 → walks from first, **2** links. Total 3.

Step 5 / 9

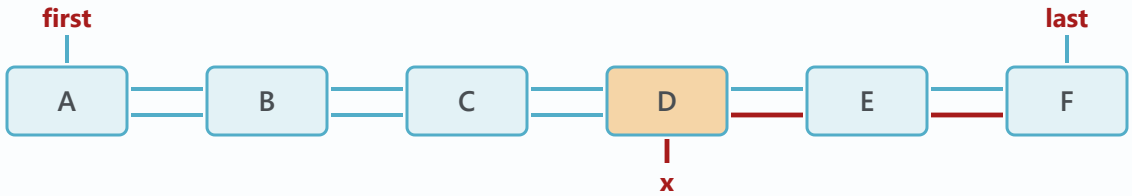
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
▶   String s = list.get(i);  // node(i) walks
▶   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = 2` `steps = 3` `s = "C"`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`get(3)` : 3 ≥ n/2 = 3 → walks from last, **2** links. Total 5.

Step 6 / 9

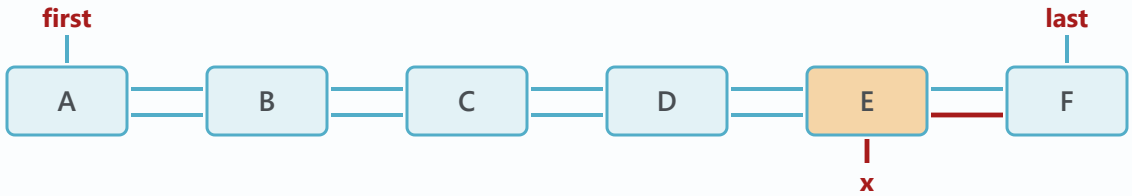
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
▶   String s = list.get(i);  // node(i) walks
▶   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = 3` `steps = 5` `s = "D"`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`get(4)` : 4 ≥ n/2 = 3 → walks from last, **1** link. Total 6.

Step 7 / 9

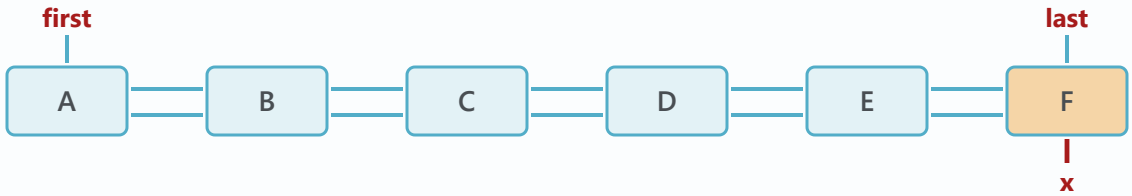
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
▶   String s = list.get(i);  // node(i) walks
▶   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = 4` `steps = 6` `s = "E"`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



`get(5)` : 5 ≥ n/2 = 3 → walks from last, **0** links. Total 6.

Step 8 / 9

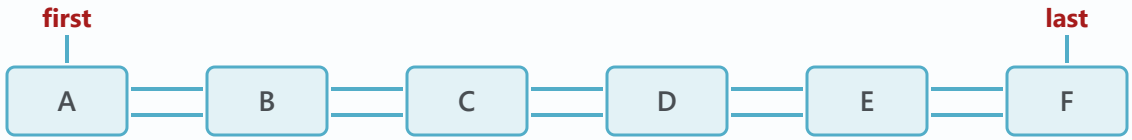
```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
▶   String s = list.get(i);  // node(i) walks
▶   steps += Math.min(i, n - 1 - i);
7 }
8 System.out.println(steps);
```

VARIABLES

`n = 6` `i = 5` `steps = 6` `s = "F"`

OUTPUT

Code Trace: The Cost of an Indexed Loop on LinkedList



Total 6 = $n^2/4 - n/2 \approx n^2/4 \rightarrow$ **O(n^2)**. An Iterator walks 1 link per `next()` : n − 1 = 5 in total → O(n).

Step 9 / 9

```
1 List<String> list = new LinkedList<>(
2     List.of("A", "B", "C", "D", "E", "F"));
3 int n = list.size(), steps = 0;
4 for (int i = 0; i < n; i++) {
5   String s = list.get(i);  // node(i) walks
6   steps += Math.min(i, n - 1 - i);
7 }
▶ System.out.println(steps);
```

VARIABLES

n = 6 | i = 6 | steps = 6 | s = –

OUTPUT

```
6
```

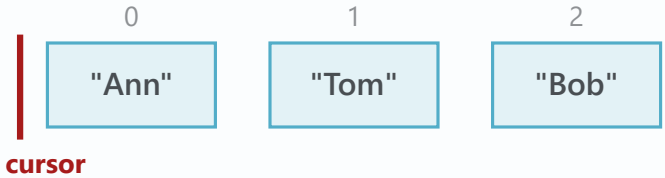
TRAVERSAL

The Iterator Interface

hasNext()
Are there elements left? `false` at the end.

next()
Returns the next element and advances the cursor. With nothing left: `NoSuchElementException`.

remove()
Removes the element returned by the last `next()` (an optional operation).



`iterator()` : the cursor sits **before** the first element. `hasNext()` → true

Step 1 / 5

```
while (iterator.hasNext()) { String name = iterator.next(); System.out.println(name); }
```

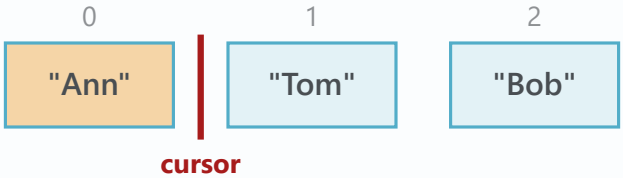
TRAVERSAL

The Iterator Interface

hasNext()
Are there elements left? `false` at the end.

next()
Returns the next element and advances the cursor. With nothing left: `NoSuchElementException`.

remove()
Removes the element returned by the last `next()` (an optional operation).



`next()` → "Ann" (the cursor jumps over the element)

Step 2 / 5

```
while (iterator.hasNext()) { String name = iterator.next(); System.out.println(name); }
```

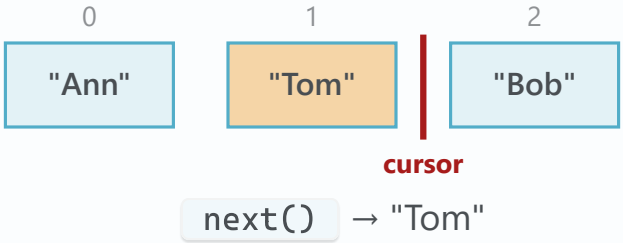
TRAVERSAL

The Iterator Interface

hasNext()
Are there elements left? `false` at the end.

next()
Returns the next element and advances the cursor. With nothing left: `NoSuchElementException`.

remove()
Removes the element returned by the last `next()` (an optional operation).



Step 3 / 5

```
while (iterator.hasNext()) { String name = iterator.next(); System.out.println(name); }
```

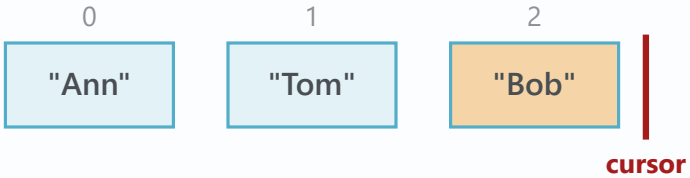
TRAVERSAL

The Iterator Interface

hasNext()
Are there elements left? `false` at the end.

next()
Returns the next element and advances the cursor. With nothing left: `NoSuchElementException`.

remove()
Removes the element returned by the last `next()` (an optional operation).



`next()` → "Bob"; now `hasNext()` → false and the loop ends


```
while (iterator.hasNext()) { String name = iterator.next(); System.out.println(name); }
```

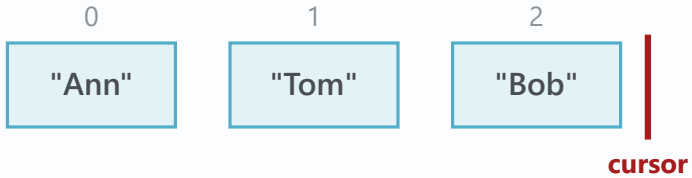
TRAVERSAL

The Iterator Interface

hasNext()
Are there elements left? `false` at the end.

next()
Returns the next element and advances the cursor. With nothing left: `NoSuchElementException`.

remove()
Removes the element returned by the last `next()` (an optional operation).



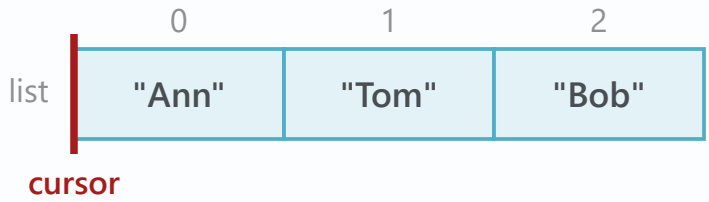
One more `next()` → `NoSuchElementException`

Step 5 / 5

```
while (iterator.hasNext()) { String name = iterator.next(); System.out.println(name); }
```

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: Traversing with an Iterator



`iterator()` : the cursor sits **before** the first element (cursor = 0). The iterator is a separate object that walks the list.

Step 1 / 11

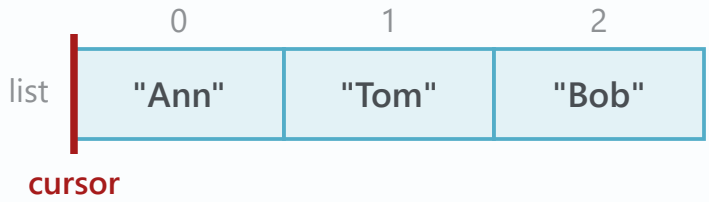
```
▶ Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
3     String name = iterator.next();
4     System.out.println(name);
5 }
```

VARIABLES

`cursor = 0` `name = –`

OUTPUT

Code Trace: Traversing with an Iterator



`hasNext()` : cursor 0 ≠ size 3 → **true**.

Step 2 / 11

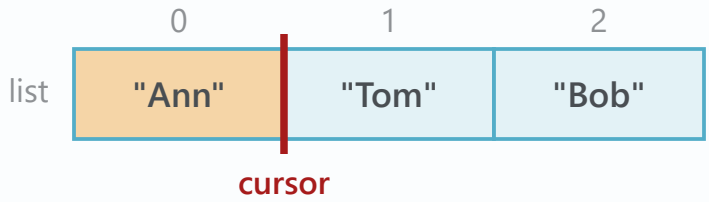
```
1 Iterator<String> iterator = names.iterator();
▶ while (iterator.hasNext()) {
3     String name = iterator.next();
4     System.out.println(name);
5 }
```

VARIABLES

`cursor = 0` `name = –`

OUTPUT

Code Trace: Traversing with an Iterator



`next()` → **"Ann"**; the cursor jumps over the element (cursor = 1).

Step 3 / 11

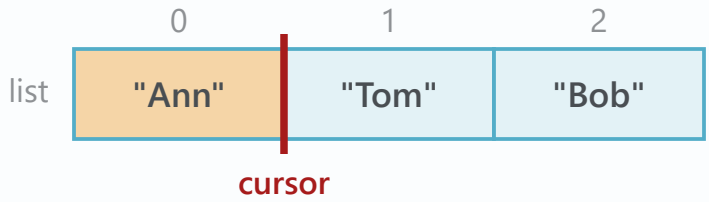
```
1 Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
▶     String name = iterator.next();
4     System.out.println(name);
5 }
```

VARIABLES

`cursor = 1` `name = "Ann"`

OUTPUT

Code Trace: Traversing with an Iterator



Prints Ann.

Step 4 / 11

```
1 Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
3     String name = iterator.next();
▶     System.out.println(name);
5 }
```

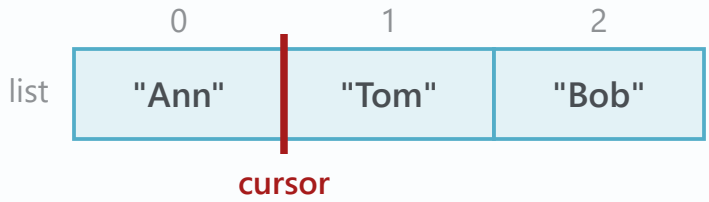
VARIABLES

`cursor = 1` `name = "Ann"`

OUTPUT

```
Ann
```

Code Trace: Traversing with an Iterator



`hasNext()` : cursor 1 ≠ size 3 → **true**.

Step 5 / 11

```
1 Iterator<String> iterator = names.iterator();
▶ while (iterator.hasNext()) {
3     String name = iterator.next();
4     System.out.println(name);
5 }
```

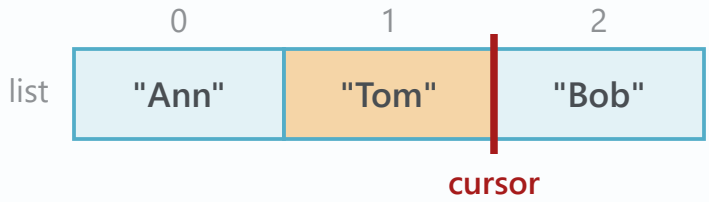
VARIABLES

`cursor = 1` `name = "Ann"`

OUTPUT

```
Ann
```

Code Trace: Traversing with an Iterator



`next()` → **"Tom"**; the cursor jumps over the element (cursor = 2).

Step 6 / 11

```
1 Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
▶     String name = iterator.next();
4     System.out.println(name);
5 }
```

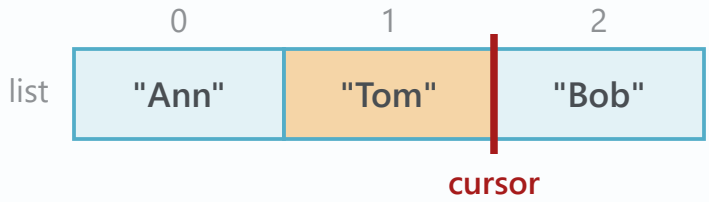
VARIABLES

`cursor = 2` `name = "Tom"`

OUTPUT

```
Ann
```

Code Trace: Traversing with an Iterator



Prints Tom.

Step 7 / 11

```
1 Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
3     String name = iterator.next();
▶     System.out.println(name);
5 }
```

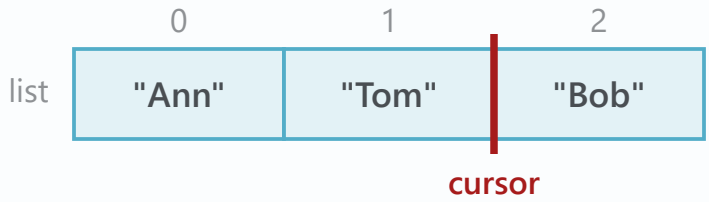
VARIABLES

`cursor = 2` `name = "Tom"`

OUTPUT

```
Ann
Tom
```

Code Trace: Traversing with an Iterator



`hasNext()` : cursor 2 ≠ size 3 → **true**.

Step 8 / 11

```
1 Iterator<String> iterator = names.iterator();
▶ while (iterator.hasNext()) {
3     String name = iterator.next();
4     System.out.println(name);
5 }
```

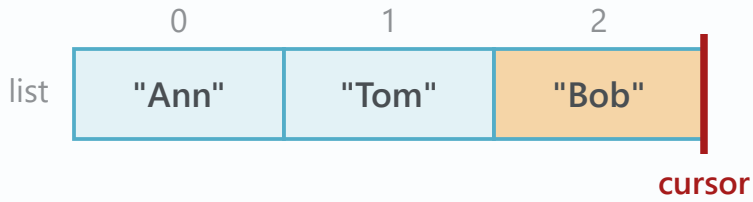
VARIABLES

`cursor = 2` `name = "Tom"`

OUTPUT

```
Ann
Tom
```

Code Trace: Traversing with an Iterator



`next()` → **"Bob"**; the cursor jumps over the element (cursor = 3).

Step 9 / 11

```
1 Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
▶     String name = iterator.next();
4     System.out.println(name);
5 }
```

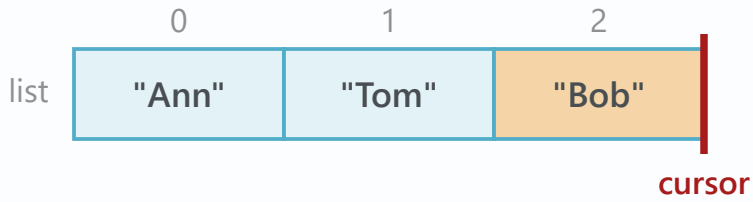
VARIABLES

`cursor = 3` `name = "Bob"`

OUTPUT

```
Ann
Tom
```

Code Trace: Traversing with an Iterator



Prints Bob.

Step 10 / 11

```
1 Iterator<String> iterator = names.iterator();
2 while (iterator.hasNext()) {
3     String name = iterator.next();
▶     System.out.println(name);
5 }
```

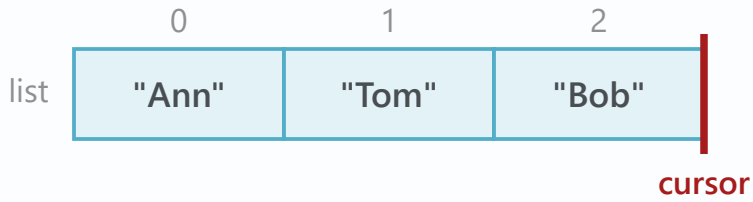
VARIABLES

`cursor = 3` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
```

Code Trace: Traversing with an Iterator



cursor 3 = size 3 → `hasNext()` is **false** and the loop ends. One more `next()` would throw `NoSuchElementException`.

Step 11 / 11

```
1 Iterator<String> iterator = names.iterator();
▶ while (iterator.hasNext()) {
3     String name = iterator.next();
4     System.out.println(name);
5 }
```

VARIABLES

`cursor = 3` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
```

TRAVERSAL

Removing While Traversing: ConcurrentModificationException

Wrong

```
for (String s : names)
    if (s.startsWith("A"))
        names.remove(s);
// ConcurrentModification-
// Exception (fail-fast)
```

Right

```
Iterator<String> it =
        names.iterator();
while (it.hasNext())
    if (it.next().startsWith("A"))
        it.remove();
// or, shorter:
names.removeIf(s -> s.startsWith("A"));
```

- for-each quietly uses an Iterator; if the list is modified behind the iterator's back, the next `next()` notices and stops (**fail-fast**).
- `it.remove()` may be called only **once** per `next()`; otherwise `IllegalStateException`.

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: `ConcurrentModificationException`

list

modCount = 0

An empty ArrayList. Every structural change (add/remove) increments the list's **modCount**.

Step 1 / 10

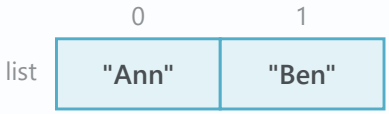
```
List<String> names = new ArrayList<>();
names.add("Ann"); names.add("Ben");
names.add("Alice");            // modCount = 3
// for (String s : names) ... compiled:
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String s = it.next();
  if (s.startsWith("A"))
    names.remove(s);          // not through it!
}
```

VARIABLES

`names = []` `modCount = 0`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 2

Two `add` calls → modCount = 2.

Step 2 / 10

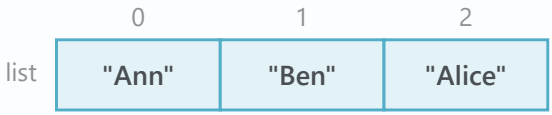
```
1  List<String> names = new ArrayList<>();
▶  names.add("Ann"); names.add("Ben");
3  names.add("Alice");            // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
7    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);           // not through it!
10 }
```

VARIABLES

`names = [Ann, Ben]` `modCount = 2`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 3

modCount = 3. Now we trace the for-each loop as the compiler expands it.

Step 3 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
▶  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
7    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);            // not through it!
10 }
```

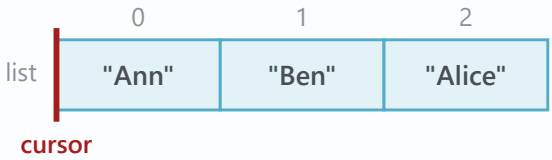
VARIABLES

`names = [Ann, Ben, Alice]`

`modCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 3 · expectedModCount = 3 ✓

`iterator()` : cursor 0; the iterator records the current count as **expectedModCount = 3**.

Step 4 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
▶  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
7    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);             // not through it!
10 }
```

VARIABLES

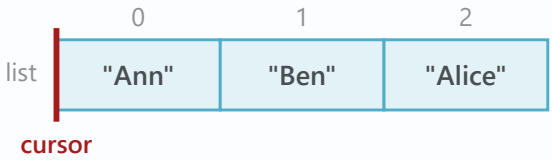
`names = [Ann, Ben, Alice]`

`cursor = 0` `s = –` `modCount = 3`

`expectedModCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 3 · expectedModCount = 3 ✓

`hasNext()` : cursor 0 ≠ size 3 → true.

Step 5 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
▶  while (it.hasNext()) {
7    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);             // not through it!
10 }
```

VARIABLES

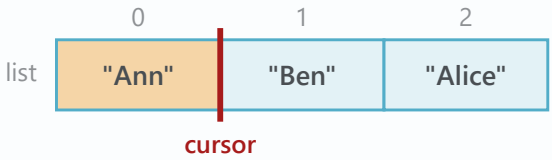
`names = [Ann, Ben, Alice]`

`cursor = 0` `s = –` `modCount = 3`

`expectedModCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 3 · expectedModCount = 3 ✓

`next()` first checks modCount == expectedModCount: 3 == 3 ✓ → **"Ann"**, cursor = 1.

Step 6 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
▶    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);             // not through it!
10 }
```

VARIABLES

`names = [Ann, Ben, Alice]`

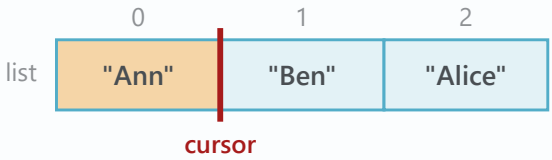
`cursor = 1` `s = "Ann"`

`modCount = 3`

`expectedModCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 3 · expectedModCount = 3 ✓

"Ann" starts with A → true.

Step 7 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
7    String s = it.next();
▶    if (s.startsWith("A"))
9      names.remove(s);             // not through it!
10 }
```

VARIABLES

`names = [Ann, Ben, Alice]`

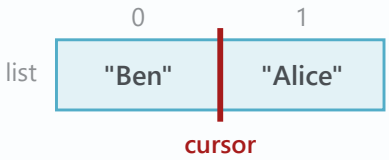
`cursor = 1` `s = "Ann"`

`modCount = 3`

`expectedModCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 4 · expectedModCount = 3 ✗

`names.remove(s)` removes **directly** from the list → elements shift left, modCount = 4. The iterator doesn't know (expected is still 3).

Step 8 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
7    String s = it.next();
8    if (s.startsWith("A"))
▶      names.remove(s);           // not through it!
10 }
```

VARIABLES

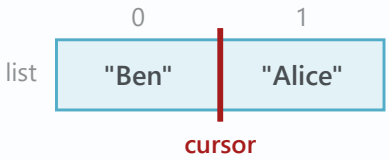
`names = [Ben, Alice]` `cursor = 1`

`s = "Ann"` `modCount = 4`

`expectedModCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 4 · expectedModCount = 3 ✗

`hasNext()` : cursor 1 ≠ size 2 → true. ("Ben" is now behind the cursor and would never be visited!)

Step 9 / 10

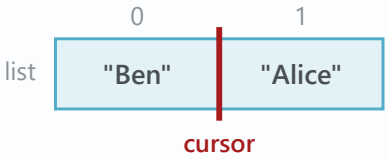
```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
▶  while (it.hasNext()) {
7    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);             // not through it!
10 }
```

VARIABLES

`names = [Ben, Alice]` `cursor = 1` `s = "Ann"` `modCount = 4` `expectedModCount = 3`

OUTPUT

Code Trace: `ConcurrentModificationException`



modCount = 4 · expectedModCount = 3 ✗

`next()` : modCount 4 ≠ expectedModCount 3 → **ConcurrentModificationException**. The iterator notices the change at once and stops (**fail-fast**).

Step 10 / 10

```
1  List<String> names = new ArrayList<>();
2  names.add("Ann"); names.add("Ben");
3  names.add("Alice");              // modCount = 3
4  // for (String s : names) ... compiled:
5  Iterator<String> it = names.iterator();
6  while (it.hasNext()) {
▶    String s = it.next();
8    if (s.startsWith("A"))
9      names.remove(s);           // not through it!
10 }
```

VARIABLES

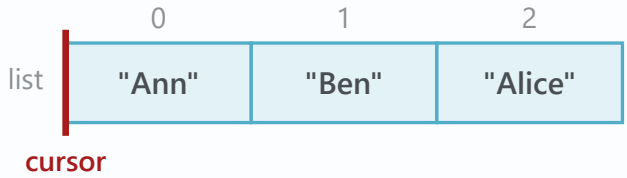
`names = [Ben, Alice]` `cursor = 1` `s = "Ann"` `modCount = 4` `expectedModCount = 3`

OUTPUT

```
Exception in thread "main"
java.util.
ConcurrentModificationException
```

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: Safe Removal with `it.remove()`



modCount = 3 · expectedModCount = 3 ✓

The same list [Ann, Ben, Alice], modCount = 3. Iterator: cursor = 0, **lastRet = −1** (nothing returned yet).

Step 1 / 14

```
▶ Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ann, Ben, Alice]`

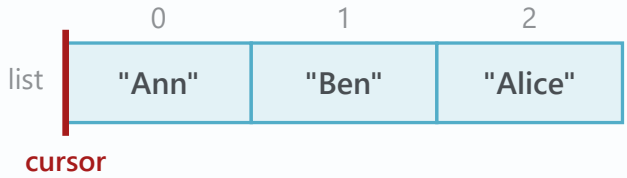
`cursor = 0` `lastRet = -1` `s = –`

`modCount = 3` `expected = 3`

OUTPUT

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: Safe Removal with `it.remove()`



modCount = 3 · expectedModCount = 3 ✓

`hasNext()` : cursor 0 ≠ size 3 → **true**.

Step 2 / 14

```
1 Iterator<String> it = names.iterator();
▶ while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

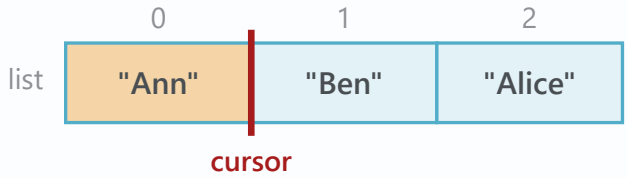
`names = [Ann, Ben, Alice]`

`cursor = 0` `lastRet = -1` `s = –`

`modCount = 3` `expected = 3`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 3 · expectedModCount = 3 ✓

`next()` : 3 == 3 ✓ → **"Ann"**; cursor = 1, lastRet = 0.

Step 3 / 14

```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
▶   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ann, Ben, Alice]`

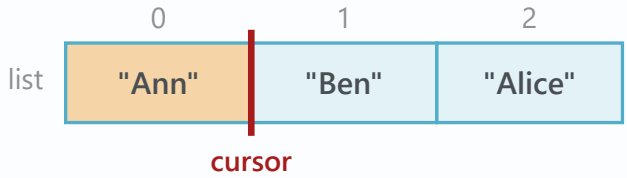
`cursor = 1` `lastRet = 0`

`s = "Ann"` `modCount = 3`

`expected = 3`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 3 · expectedModCount = 3 ✓

Does "Ann" start with A? **Yes**.

Step 4 / 14

```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
▶  if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ann, Ben, Alice]`

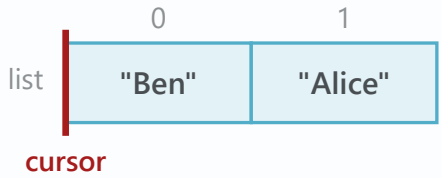
`cursor = 1` `lastRet = 0`

`s = "Ann"` `modCount = 3`

`expected = 3`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 4 · expectedModCount = 4 ✓

`it.remove()` removes the element at lastRet; **cursor = 0** (the shifted element is not skipped), lastRet = −1, expectedModCount = modCount = 4.

Step 5 / 14

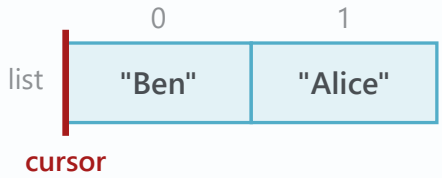
```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
▶     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ben, Alice]` `cursor = 0` `lastRet = -1` `s = "Ann"` `modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



`hasNext()` : cursor 0 ≠ size 2 → **true**.

modCount = 4 · expectedModCount = 4 ✓

Step 6 / 14

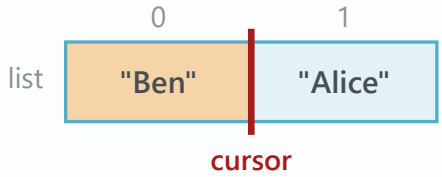
```
1 Iterator<String> it = names.iterator();
▶ while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ben, Alice]` `cursor = 0`
`lastRet = -1` `s = "Ann"`
`modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 4 · expectedModCount = 4 ✓

`next()` : 4 == 4 ✓ → **"Ben"**; cursor = 1, lastRet = 0.

Step 7 / 14

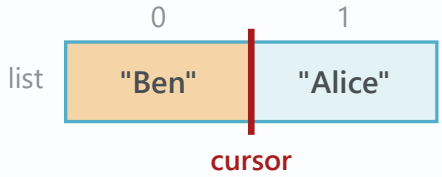
```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
▶   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ben, Alice]` `cursor = 1`
`lastRet = 0` `s = "Ben"`
`modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 4 · expectedModCount = 4 ✓

Does "Ben" start with A? **No**.

Step 8 / 14

```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
▶   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

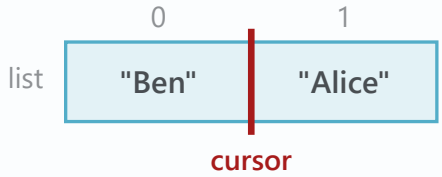
`names = [Ben, Alice]` `cursor = 1`

`lastRet = 0` `s = "Ben"`

`modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 4 · expectedModCount = 4 ✓

`hasNext()` : cursor 1 ≠ size 2 → **true**.

Step 9 / 14

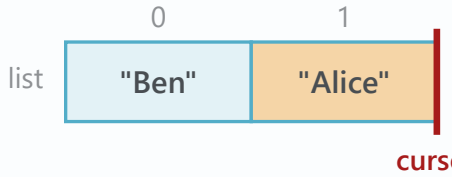
```
1 Iterator<String> it = names.iterator();
▶ while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ben, Alice]` `cursor = 1`
`lastRet = 0` `s = "Ben"`
`modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 4 · expectedModCount = 4 ✓

`next()` : 4 == 4 ✓ → **"Alice"**; cursor = 2, lastRet = 1.

Step 10 / 14

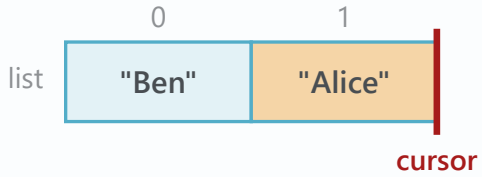
```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
▶   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ben, Alice]` `cursor = 2` `lastRet = 1` `s = "Alice"` `modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 4 · expectedModCount = 4 ✓

Does "Alice" start with A? **Yes**.

Step 11 / 14

```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
▶  if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

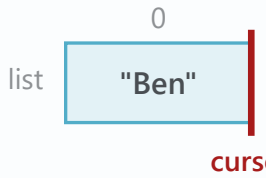
`names = [Ben, Alice]` `cursor = 2`

`lastRet = 1` `s = "Alice"`

`modCount = 4` `expected = 4`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 5 · expectedModCount = 5 ✓

`it.remove()` removes the element at lastRet; **cursor = 1** (the shifted element is not skipped), lastRet = −1, expectedModCount = modCount = 5.

Step 12 / 14

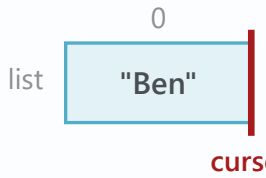
```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
▶     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

names = [Ben] | cursor = 1 | lastRet = -1 | s = "Alice" | modCount = 5 | expected = 5

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 5 · expectedModCount = 5 ✓

`hasNext()` : cursor 1 = size 1 → **false**. The loop ends with no error.

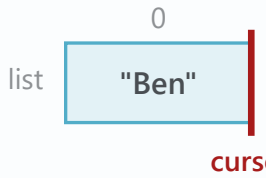

```
1 Iterator<String> it = names.iterator();
▶ while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
7 System.out.println(names);
```

VARIABLES

`names = [Ben]` `cursor = 1`
`lastRet = -1` `s = "Alice"`
`modCount = 5` `expected = 5`

OUTPUT

Code Trace: Safe Removal with `it.remove()`



modCount = 5 · expectedModCount = 5 ✓

The result is **[Ben]**: the names starting with A were removed safely. A second `it.remove()` with no `next()` in between → IllegalStateException.

Step 14 / 14

```
1 Iterator<String> it = names.iterator();
2 while (it.hasNext()) {
3   String s = it.next();
4   if (s.startsWith("A"))
5     it.remove();   // keeps the counts in sync
6 }
▶ System.out.println(names);
```

VARIABLES

`names = [Ben]` `cursor = 1`
`lastRet = -1` `s = "Alice"`
`modCount = 5` `expected = 5`

OUTPUT

```
[Ben]
```

TRAVERSAL

The ListIterator Interface

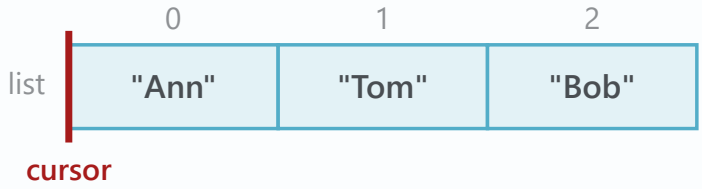
Available only on `List`s (`list.listIterator()`). Beyond what an Iterator does, it moves in **both directions** and can add, remove, and replace elements on the way. The cursor sits **between** elements.

Method	What it does
`hasNext()` / `next()`	is there an element ahead? / return the next element and move the cursor forward
`hasPrevious()` / `previous()`	is there an element behind? / return the previous element and move the cursor back
`nextIndex()` / `previousIndex()`	the index of the element `next()` / `previous()` would return (0 / −1 at the start)
`add(e)`	inserts the element **before the cursor** (the next `next()` is unaffected)
`remove()`	removes the element returned by the last `next()` / `previous()`
`set(e)`	replaces the element returned by the last `next()` / `previous()`

Calling `remove` / `set` right after `add` or `remove`, with no `next` / `previous` in between → `IllegalStateException`.

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: ListIterator, Step by Step



[Ann, Tom, Bob]

`listIterator()` : the cursor is at the start; nextIndex = 0, previousIndex = −1. Orange: the last returned element (lastRet).

Step 1 / 7

```
1 ▶ ListIterator<String> it = names.listIterator();
2   it.next();          // "Ann"
3   it.next();          // "Tom"
4   it.set("TOM");    // [Ann, TOM, Bob]
5   it.add("Alice");     // [Ann, TOM, Alice, Bob]
6   it.previous();      // "Alice"
7   it.remove();        // [Ann, TOM, Bob]
```

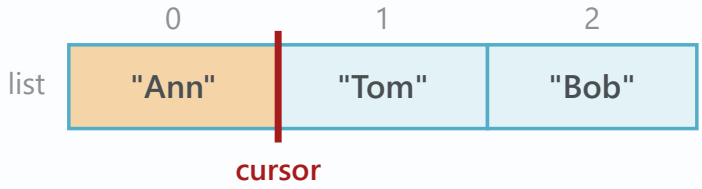
VARIABLES

`nextIndex = 0`

`previousIndex = -1` `lastRet = -1`

`returned = –`

Code Trace: ListIterator, Step by Step



[Ann, Tom, Bob]

`next()` → "Ann"; the cursor moves forward, lastRet = 0.

Step 2 / 7

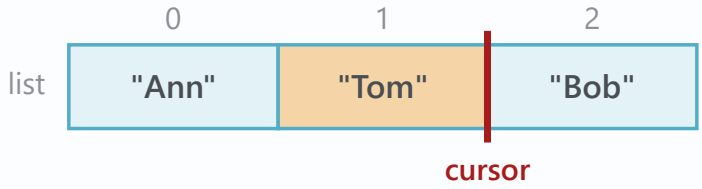
```
1 ListIterator<String> it = names.listIterator();
▶ it.next();          // "Ann"
3 it.next();          // "Tom"
4 it.set("TOM");    // [Ann, TOM, Bob]
5 it.add("Alice");     // [Ann, TOM, Alice, Bob]
6 it.previous();      // "Alice"
7 it.remove();        // [Ann, TOM, Bob]
```

VARIABLES

`nextIndex = 1` `previousIndex = 0`

`lastRet = 0` `returned = "Ann"`

Code Trace: ListIterator, Step by Step



[Ann, Tom, Bob]

`next()` → "Tom"; lastRet = 1.

Step 3 / 7

```
1 ListIterator<String> it = names.listIterator();
2 it.next();          // "Ann"
▶ it.next();          // "Tom"
4 it.set("TOM");    // [Ann, TOM, Bob]
5 it.add("Alice");     // [Ann, TOM, Alice, Bob]
6 it.previous();      // "Alice"
7 it.remove();        // [Ann, TOM, Bob]
```

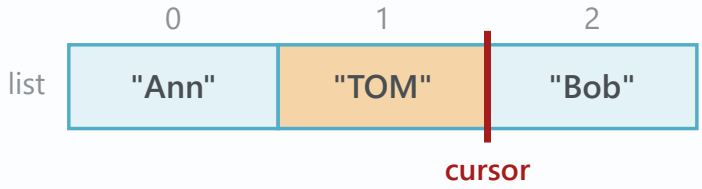
VARIABLES

`nextIndex = 2` `previousIndex = 1`

`lastRet = 1` `returned = "Tom"`

TRAVERSAL · STEP THROUGH THE CODE

Code Trace: ListIterator, Step by Step



[Ann, TOM, Bob]

`set("TOM")` replaces the element at **lastRet** ("Tom"); the cursor stays put.

Step 4 / 7

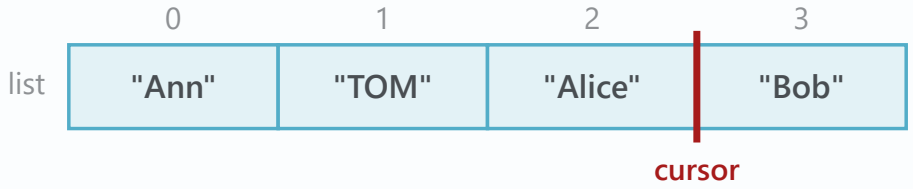
```
1 ListIterator<String> it = names.listIterator();
2 it.next();          // "Ann"
3 it.next();          // "Tom"
▶ it.set("TOM");    // [Ann, TOM, Bob]
5 it.add("Alice");     // [Ann, TOM, Alice, Bob]
6 it.previous();      // "Alice"
7 it.remove();        // [Ann, TOM, Bob]
```

VARIABLES

`nextIndex = 2` `previousIndex = 1`

`lastRet = 1` `returned = –`

Code Trace: ListIterator, Step by Step



[Ann, TOM, Alice, Bob]

`add("Alice")` inserts **before the cursor**, and the cursor moves past the new element (the next `next` still returns "Bob"); lastRet = −1.

Step 5 / 7

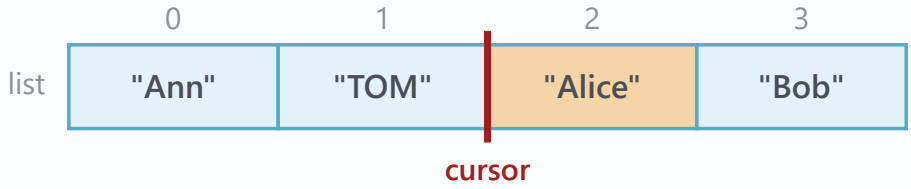
```
1 ListIterator<String> it = names.listIterator();
2 it.next();          // "Ann"
3 it.next();          // "Tom"
4 it.set("TOM");    // [Ann, TOM, Bob]
▶ it.add("Alice");     // [Ann, TOM, Alice, Bob]
6 it.previous();      // "Alice"
7 it.remove();        // [Ann, TOM, Bob]
```

VARIABLES

`nextIndex = 3` `previousIndex = 2`

`lastRet = -1` `returned = –`

Code Trace: ListIterator, Step by Step



[Ann, TOM, Alice, Bob]

`previous()` → "Alice", moving backward; the cursor steps back, lastRet = 2.

Step 6 / 7

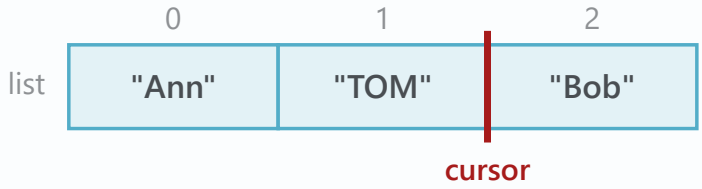
```
1 ListIterator<String> it = names.listIterator();
2 it.next();          // "Ann"
3 it.next();          // "Tom"
4 it.set("TOM");    // [Ann, TOM, Bob]
5 it.add("Alice");     // [Ann, TOM, Alice, Bob]
▶ it.previous();      // "Alice"
7 it.remove();        // [Ann, TOM, Bob]
```

VARIABLES

`nextIndex = 2` `previousIndex = 1`

`lastRet = 2` `returned = "Alice"`

Code Trace: ListIterator, Step by Step



[Ann, TOM, Bob]

`remove()` deletes "Alice" at lastRet; lastRet = −1 (an immediate second remove/set → IllegalStateException).

Step 7 / 7

```
1 ListIterator<String> it = names.listIterator();
2 it.next();          // "Ann"
3 it.next();          // "Tom"
4 it.set("TOM");    // [Ann, TOM, Bob]
5 it.add("Alice");     // [Ann, TOM, Alice, Bob]
6 it.previous();      // "Alice"
▶ it.remove();        // [Ann, TOM, Bob]
```

VARIABLES

`nextIndex = 2` `previousIndex = 1`

`lastRet = -1` `returned = –`

ListIterator Demo: Forward and Backward

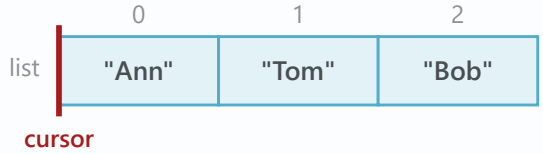
```
ListIterator<String> iterator =
        names.listIterator();
// traverse the list forward
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}
// cursor is now at the end: go backward
while (iterator.hasPrevious()) {
    String name = iterator.previous();
    System.out.println(name);
}
```

```
Ann
Tom
Bob
Bob
Tom
Ann
```

The second loop reuses the same iterator: when the first loop ends, the cursor is at the end of the list.

To start at the end directly: `names.listIterator(names.size())`.

Code Trace: Forward and Backward



`listIterator()` : the cursor is at the start.

Step 1 / 15

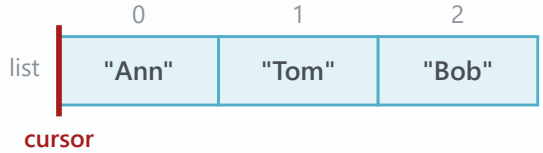
```
▶ ListIterator<String> iterator =
▶         names.listIterator();
3 // traverse the list forward
4 while (iterator.hasNext()) {
5     String name = iterator.next();
6     System.out.println(name);
7 }
8 // cursor is now at the end: go backward
9 while (iterator.hasPrevious()) {
10    String name = iterator.previous();
11    System.out.println(name);
12 }
```

VARIABLES

`cursor = 0` `name = –`

OUTPUT

Code Trace: Forward and Backward



`hasNext()` → true (cursor 0 < 3).

Step 2 / 15

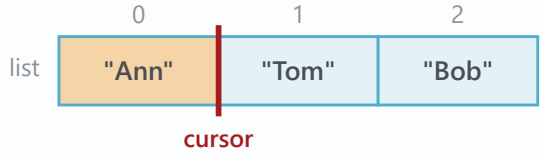
```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
▶  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

VARIABLES

`cursor = 0` `name = –`

OUTPUT

Code Trace: Forward and Backward



`next()` → "Ann", printed; the cursor moves forward.

Step 3 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
▶      String name = iterator.next();
▶      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

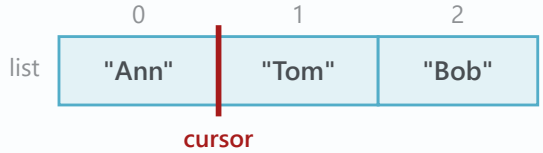
VARIABLES

`cursor = 1` `name = "Ann"`

OUTPUT

```
Ann
```

Code Trace: Forward and Backward



`hasNext()` → true (cursor 1 < 3).

Step 4 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
▶  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

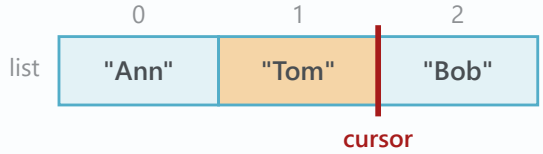
VARIABLES

`cursor = 1` `name = "Ann"`

OUTPUT

```
Ann
```

Code Trace: Forward and Backward



`next()` → "Tom", printed; the cursor moves forward.

Step 5 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
▶      String name = iterator.next();
▶      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

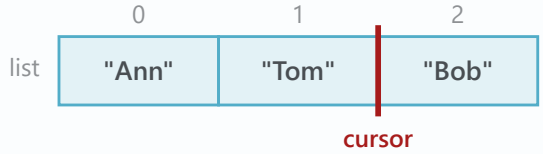
VARIABLES

`cursor = 2` `name = "Tom"`

OUTPUT

```
Ann
Tom
```

Code Trace: Forward and Backward



`hasNext()` → true (cursor 2 < 3).

Step 6 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
▶  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

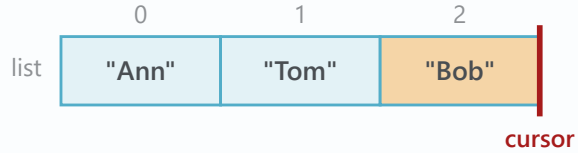
VARIABLES

`cursor = 2` `name = "Tom"`

OUTPUT

```
Ann
Tom
```

Code Trace: Forward and Backward



`next()` → "Bob", printed; the cursor moves forward.

Step 7 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
▶      String name = iterator.next();
▶      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

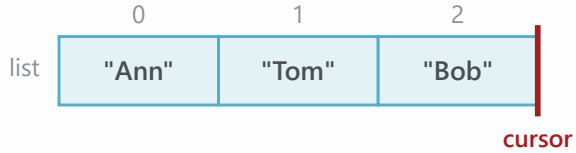
VARIABLES

`cursor = 3` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
```

Code Trace: Forward and Backward



`hasNext()` → false: the cursor is at the end. The first loop ends; we continue with the **same iterator**.

Step 8 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
▶ while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

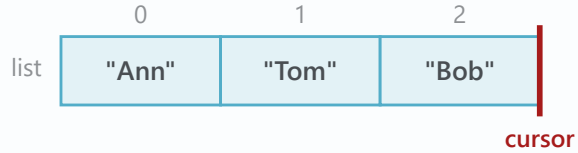
VARIABLES

`cursor = 3` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
```

Code Trace: Forward and Backward



`hasPrevious()` → true (cursor 3 > 0).

Step 9 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
▶ while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

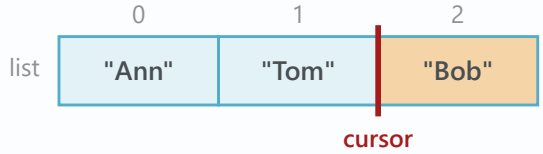
VARIABLES

`cursor = 3` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
```

Code Trace: Forward and Backward



`previous()` → "Bob", printed; the cursor moves back.

Step 10 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
▶      String name = iterator.previous();
▶      System.out.println(name);
12 }
```

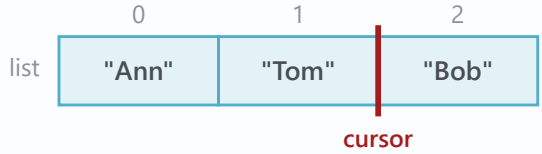
VARIABLES

`cursor = 2` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
Bob
```

Code Trace: Forward and Backward



`hasPrevious()` → true (cursor 2 > 0).

Step 11 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
▶  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

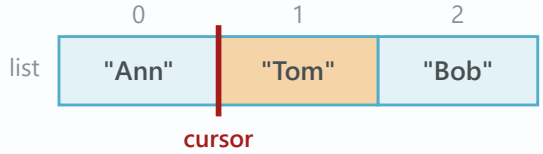
VARIABLES

`cursor = 2` `name = "Bob"`

OUTPUT

```
Ann
Tom
Bob
Bob
```

Code Trace: Forward and Backward



`previous()` → "Tom", printed; the cursor moves back.

Step 12 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
▶      String name = iterator.previous();
▶      System.out.println(name);
12 }
```

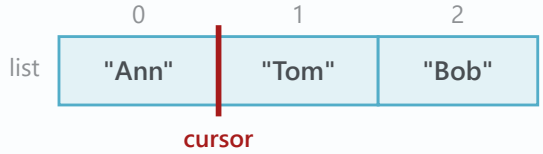
VARIABLES

`cursor = 1` `name = "Tom"`

OUTPUT

```
Ann
Tom
Bob
Bob
Tom
```

Code Trace: Forward and Backward



`hasPrevious()` → true (cursor 1 > 0).


```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
▶  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

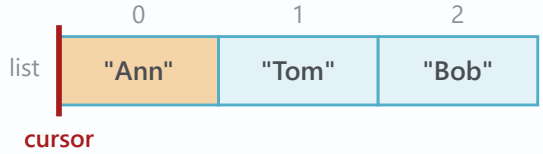
VARIABLES

`cursor = 1` `name = "Tom"`

OUTPUT

```
Ann
Tom
Bob
Bob
Tom
```

Code Trace: Forward and Backward



`previous()` → "Ann", printed; the cursor moves back.


```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
9  while (iterator.hasPrevious()) {
▶      String name = iterator.previous();
▶      System.out.println(name);
12 }
```

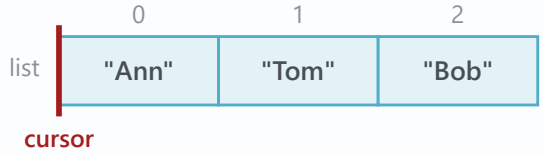
VARIABLES

`cursor = 0` `name = "Ann"`

OUTPUT

```
Ann
Tom
Bob
Bob
Tom
Ann
```

Code Trace: Forward and Backward



`hasPrevious()` → false: the cursor is back at the start. Output: 3 lines forward + 3 backward.

Step 15 / 15

```
1  ListIterator<String> iterator =
2          names.listIterator();
3  // traverse the list forward
4  while (iterator.hasNext()) {
5      String name = iterator.next();
6      System.out.println(name);
7  }
8  // cursor is now at the end: go backward
▶  while (iterator.hasPrevious()) {
10     String name = iterator.previous();
11     System.out.println(name);
12 }
```

VARIABLES

`cursor = 0` `name = "Ann"`

OUTPUT

```
Ann
Tom
Bob
Bob
Tom
Ann
```

LIST IMPLEMENTATIONS

Classes That Implement List

ArrayList

A growable array. Fast indexed access; the default general-purpose choice.

LinkedList

A doubly linked list. Fast insertion/removal at the ends; slow indexed access.

Vector

An older version of ArrayList whose methods are synchronized.

Stack

A stack that extends Vector: push, pop, peek (LIFO).

CopyOnWriteArrayList

Copies on every write; suits lists read by many threads.

Arrays.asList()

Turns an array into a fixed-size List view.

ArrayList: A Growable Array

- Keeps its elements in an internal array. **Capacity** (array length) ≥ **size** (number of elements).
- When the array fills up, a new one about 1.5 times larger is created and the elements are copied (O(n)). Since this is rare, appending is **amortized O(1)**.
- It does not shrink on removal; `trimToSize()` cuts the capacity down to the size.

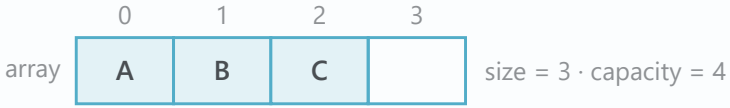
add(x) remove(size−1) trimToSize() reset



size = **3** · capacity = **4** · total copies = 0 (over 0 adds)

Real Java: an empty `new ArrayList<>()` gets capacity 10 on the first add; new capacity = old + old/2. The demo starts at 4.

Code Trace: ArrayList `add(e)` and Growth



Capacity 4, size 3 (a demo; real Java starts an empty list at 10 on the first add). We add D, E, F, G in turn.

Step 1 / 13

```
▶ public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

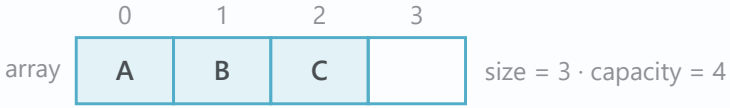
VARIABLES

`e = –` `size = 3` `data.length = 4` `copied = 0`

CALL STACK

`main()`

Code Trace: ArrayList `add(e)` and Growth



`add(D)` : size 3 == capacity 4? **No**.

Step 2 / 13

```
1 public boolean add(E e) {
▶   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

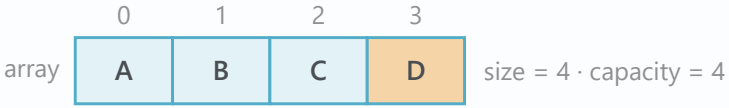
`e = D` `size = 3` `data.length = 4` `newCap = –` `copied = 0`

CALL STACK

`add(D)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`data[3] = D`, size = 4; returns true.

Step 3 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
    data[size++] = e;
    return true;
8 }
```

VARIABLES

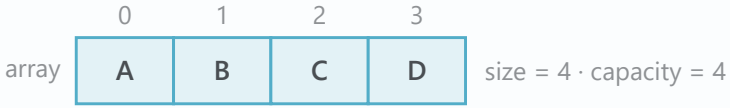
`e = D` `size = 4` `data.length = 4` `newCap = –` `copied = 0`

CALL STACK

`add(D)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`add(E)` : size 4 == capacity 4? **Yes** → **the array is full**.

Step 4 / 13

```
1 public boolean add(E e) {
▶   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

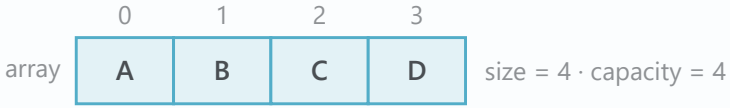
`e = E` `size = 4` `data.length = 4` `newCap = –` `copied = 0`

CALL STACK

`add(E)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



newCap = 4 + 4 / 2 = 6: about 1.5 times (Java: old + old >> 1).

Step 5 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
▶     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

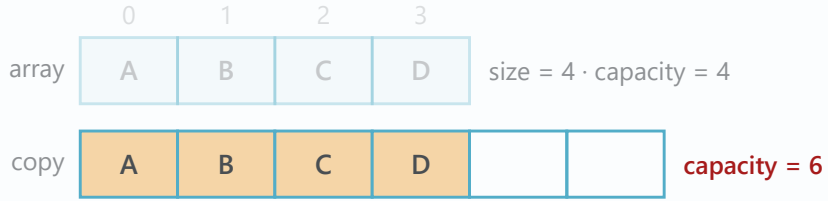
`e = E` `size = 4` `data.length = 4` `newCap = 6` `copied = 0`

CALL STACK

`add(E)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`Arrays.copyOf`: a new array with 6 slots; the 4 elements are copied one by one → this add is **O(n)**. The old array becomes garbage.

Step 6 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
▶     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

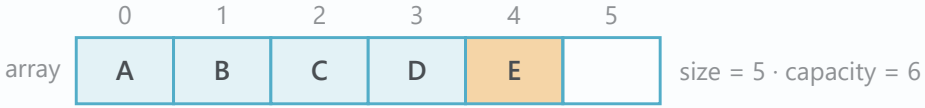
`e = E` `size = 4` `data.length = 4` `newCap = 6` `copied = 4`

CALL STACK

`add(E)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`data[4] = E`, size = 5; returns true.

Step 7 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
▶   data[size++] = e;
▶   return true;
8 }
```

VARIABLES

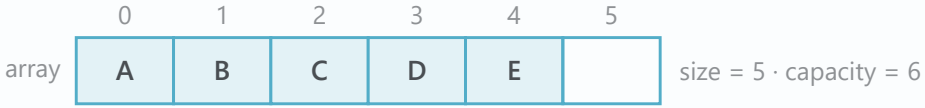
`e = E` `size = 5` `data.length = 6` `newCap = –` `copied = 4`

CALL STACK

`add(E)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`add(F)` : size 5 == capacity 6? **No**.

Step 8 / 13

```
1 public boolean add(E e) {
▶   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

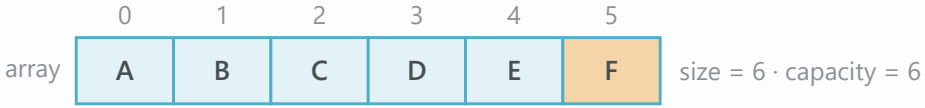
`e = F` `size = 5` `data.length = 6` `newCap = –` `copied = 4`

CALL STACK

`add(F)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`data[5] = F`, size = 6; returns true.

Step 9 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
    data[size++] = e;
    return true;
8 }
```

VARIABLES

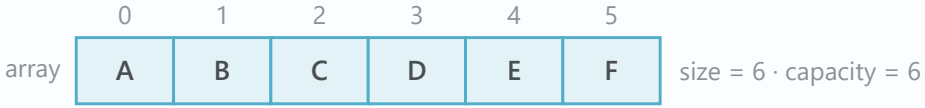
`e = F` `size = 6` `data.length = 6` `newCap = –` `copied = 4`

CALL STACK

`add(F)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`add(G)`: size 6 == capacity 6? **Yes** → **the array is full**.

Step 10 / 13

```
1 public boolean add(E e) {
▶   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

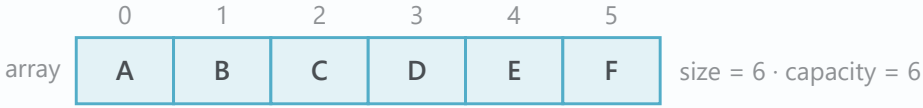
`e = G` `size = 6` `data.length = 6` `newCap = –` `copied = 4`

CALL STACK

`add(G)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



newCap = 6 + 6 / 2 = 9: about 1.5 times (Java: old + old >> 1).

Step 11 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
▶     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

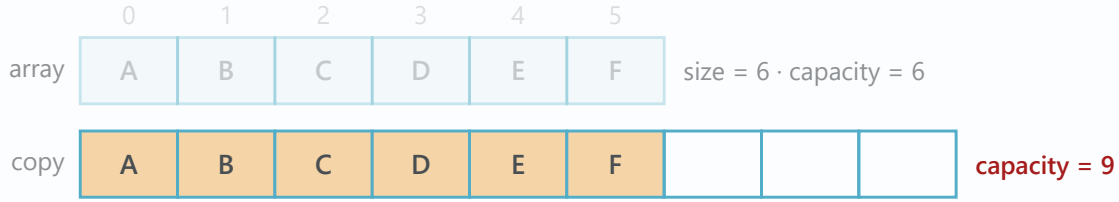
`e = G` `size = 6` `data.length = 6` `newCap = 9` `copied = 4`

CALL STACK

`add(G)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`Arrays.copyOf`: a new array with 9 slots; the 6 elements are copied one by one → this add is **O(n)**. The old array becomes garbage.

Step 12 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
▶     data = Arrays.copyOf(data, newCap);
5   }
6   data[size++] = e;
7   return true;
8 }
```

VARIABLES

`e = G` `size = 6` `data.length = 6` `newCap = 9` `copied = 10`

CALL STACK

`add(G)`

`main()`

Code Trace: ArrayList `add(e)` and Growth



`data[6] = G`, size = 7; returns true. Because the capacity grows by 1.5 times each time, resizes get rarer and rarer: n adds copy O(n) elements in total → an add is **amortized O(1)**.

Step 13 / 13

```
1 public boolean add(E e) {
2   if (size == data.length) {
3     int newCap = Math.max(data.length + 1,
  data.length + data.length / 2);
4     data = Arrays.copyOf(data, newCap);
5   }
▶   data[size++] = e;
▶   return true;
8 }
```

VARIABLES

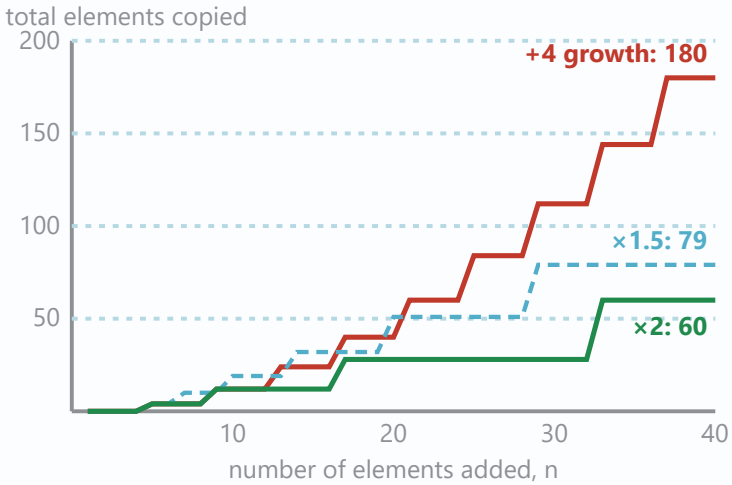
`e = G` `size = 7` `data.length = 9` `newCap = –` `copied = 10`

CALL STACK

`add(G)`

`main()`

Amortized Analysis: Why Grow Geometrically?



- **Doubling** (×2): n adds copy 4 + 8 + 16 + … < 2n elements → **amortized O(1)** per operation.
- Java's ×1.5 is geometric growth too: about 3n in total (integer division can shift this slightly), still amortized O(1).
- **Fixed increments** (+c): c + 2c + 3c + … ≈ $n^2/(2c)$ copies → **O(n)** per operation.
- Goodrich–Tamassia's "banker's" method: each cheap add deposits a few "tokens," and those tokens pay for the expensive resize.

The chart shows copies actually counted over 40 adds to a dynamic array that starts at capacity 4.

Using ArrayList: Operations and Costs

```
ArrayList<String> list =
        new ArrayList<>();
list.add("Ann");      // [Ann]
list.add("Dan");     // [Ann, Dan]
list.add(0, "Ben");  // [Ben, Ann, Dan]
list.get(1);         // "Ann"
list.remove(0);      // [Ann, Dan]
list.size();         // 2
```

Operation	Cost
`get(i)`, `set(i, e)`	O(1)
`add(e)` (at the end)	amortized O(1)
`add(i, e)`, `remove(i)`	O(n)
`contains`, `indexOf`	O(n)
`size()`	O(1)

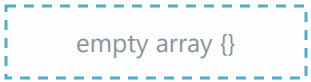
Pros over an array

Grows on its own; adding, removing, searching, and sorting come as ready-made methods.

Cons

Holds objects only (boxing costs); every access pays for a bounds check and a method call.

Code Trace: What ArrayList Operations Cost



An empty `new ArrayList<>()` : the internal array starts empty; the first add allocates one of capacity 10.

Step 1 / 9

```
▶ ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
4 list.add(0, "Ben");
5 String s = list.get(1);
6 list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = []` `size = 0`

`capacity = 0` `shifts = 0`

Code Trace: What ArrayList Operations Cost



The first `add` creates an array of 10 and puts Ann at index 0. Appending needs no shifting.

Step 2 / 9

```
1 ArrayList<String> list = new ArrayList<>();
▶ list.add("Ann");
3 list.add("Dan");
4 list.add(0, "Ben");
5 String s = list.get(1);
6 list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = [Ann]` `size = 1`

`capacity = 10` `shifts = 0`

Code Trace: What ArrayList Operations Cost



Dan goes to index 1 (amortized O(1)).

Step 3 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
▶ list.add("Dan");
4 list.add(0, "Ben");
5 String s = list.get(1);
6 list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = [Ann, Dan]` `size = 2`

`capacity = 10` `shifts = 0`

Code Trace: What ArrayList Operations Cost



`add(0, …)` : first the 2 elements from index 0 on shift one place right (starting from the right).

Step 4 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
▶ list.add(0, "Ben");
5 String s = list.get(1);
6 list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = [Ann, Dan]` `size = 2`

`capacity = 10` `shifts = 2`

Code Trace: What ArrayList Operations Cost



Ben goes into the freed index 0. Inserting at the front costs n shifts → **O(n)**.

Step 5 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
▶ list.add(0, "Ben");
5 String s = list.get(1);
6 list.remove(0);
7 int n = list.size();
```

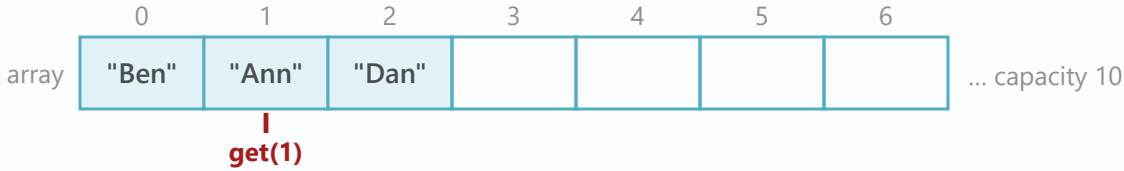
VARIABLES

`list = [Ben, Ann, Dan]`

`size = 3` `capacity = 10`

`shifts = 2`

Code Trace: What ArrayList Operations Cost



`get(1)` : direct access by address arithmetic → **O(1)**, no shifting.

Step 6 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
4 list.add(0, "Ben");
▶ String s = list.get(1);
6 list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = [Ben, Ann, Dan]`

`size = 3` `capacity = 10`

`shifts = 2` `s = "Ann"`

Code Trace: What ArrayList Operations Cost



`remove(0)` : Ben is about to go...

Step 7 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
4 list.add(0, "Ben");
5 String s = list.get(1);
▶ list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = [Ben, Ann, Dan]`

`size = 3` `capacity = 10`

`shifts = 2` `s = "Ann"`

Code Trace: What ArrayList Operations Cost



...the next 2 elements shift left and the last cell is set to null. The capacity does not shrink.

Step 8 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
4 list.add(0, "Ben");
5 String s = list.get(1);
▶ list.remove(0);
7 int n = list.size();
```

VARIABLES

`list = [Ann, Dan]` `size = 2`

`capacity = 10` `shifts = 4`

`s = "Ann"`

Code Trace: What ArrayList Operations Cost



`size()` = 2: it reads a counter field, O(1). All 4 shifts came from operations at the front.

Step 9 / 9

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Ann");
3 list.add("Dan");
4 list.add(0, "Ben");
5 String s = list.get(1);
6 list.remove(0);
▶ int n = list.size();
```

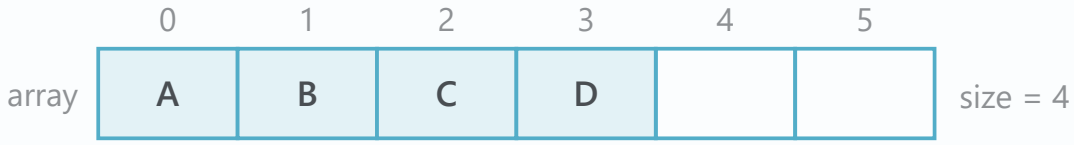
VARIABLES

`list = [Ann, Dan]` `size = 2`

`capacity = 10` `shifts = 4`

`s = "Ann"` `n = 2`

Code Trace: ArrayList `add(i, e)`



[A, B, C, D], capacity 6. `add(1, "X")`: X goes to index 1, so B, C, D must shift one place right.

Step 1 / 9

```
▶ public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
4   for (int k = size; k > i; k--)
5     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

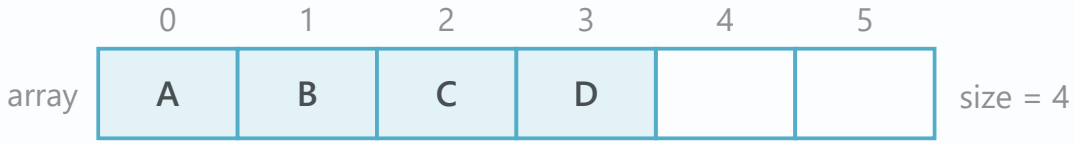
`i = 1` `e = X` `k = –` `size = 4`

`data = [A, B, C, D, null, null]`

CALL STACK

`main()`

Code Trace: ArrayList `add(i, e)`



0 ≤ 1 ≤ size (4) → a valid index (otherwise IndexOutOfBoundsException).

Step 2 / 9

```
1 public void add(int i, E e) {
▶   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
4   for (int k = size; k > i; k--)
5     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = –` `size = 4`

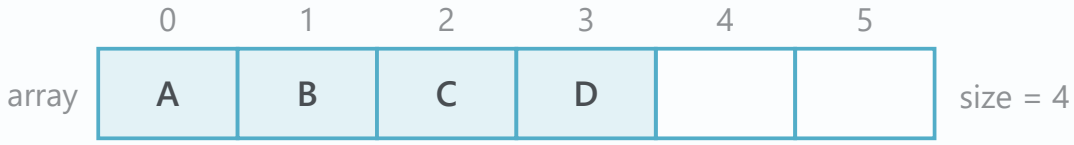
`data = [A, B, C, D, null, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



size 4 < capacity 6 → no need to grow.

Step 3 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
▶   if (size == data.length) resize();
4   for (int k = size; k > i; k--)
5     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = –` `size = 4`

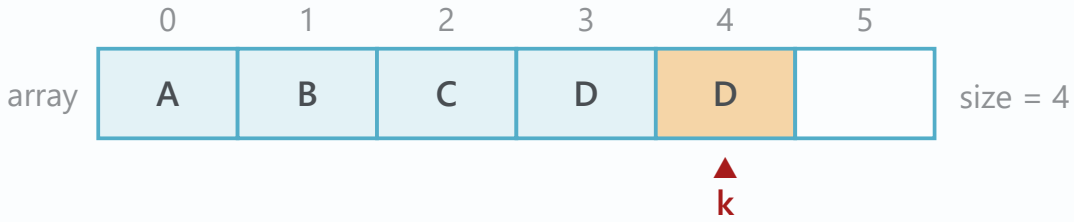
`data = [A, B, C, D, null, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



k = 4 > 1: data[4] = data[3] → D is copied right. We start **from the right** so nothing is overwritten.

Step 4 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
▶   for (int k = size; k > i; k--)
▶     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = 4` `size = 4`

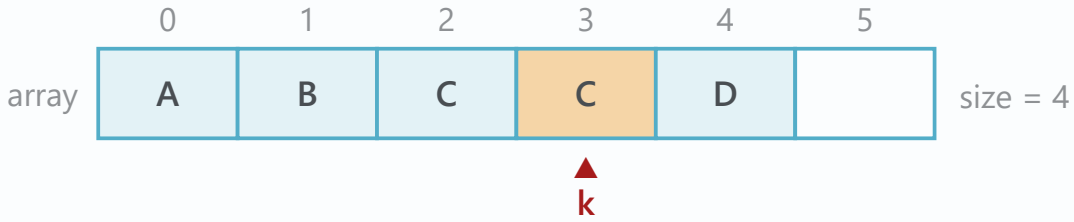
`data = [A, B, C, D, D, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



k = 3 > 1: data[3] = data[2] → C is copied right. We start **from the right** so nothing is overwritten.

Step 5 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
▶   for (int k = size; k > i; k--)
▶     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = 3` `size = 4`

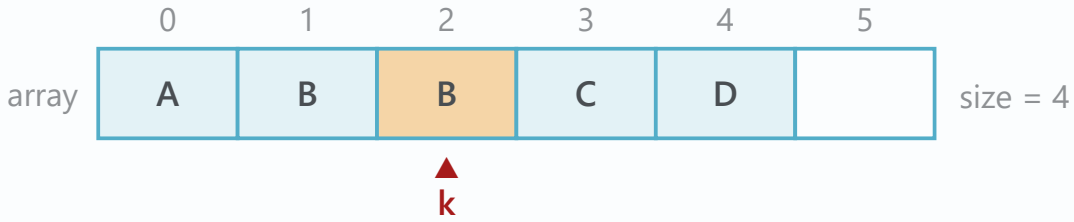
`data = [A, B, C, C, D, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



k = 2 > 1: data[2] = data[1] → B is copied right. We start **from the right** so nothing is overwritten.

Step 6 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
▶   for (int k = size; k > i; k--)
▶     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = 2` `size = 4`

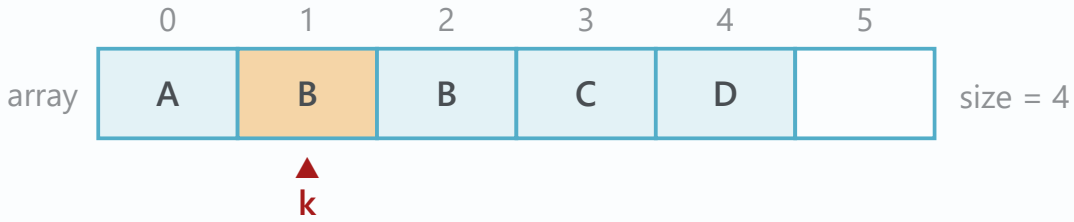
`data = [A, B, B, C, D, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



k = 1 > 1 is false → the loop ends. data[1] still holds the old copy of B.

Step 7 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
▶   for (int k = size; k > i; k--)
5     data[k] = data[k - 1];
6   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = 1` `size = 4`

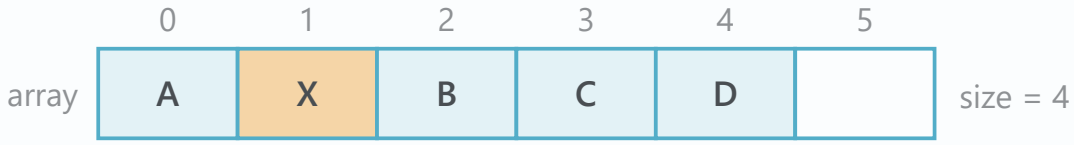
`data = [A, B, B, C, D, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



`data[1] = X` fills the gap.

Step 8 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
4   for (int k = size; k > i; k--)
5     data[k] = data[k - 1];
▶   data[i] = e;
7   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = 1` `size = 4`

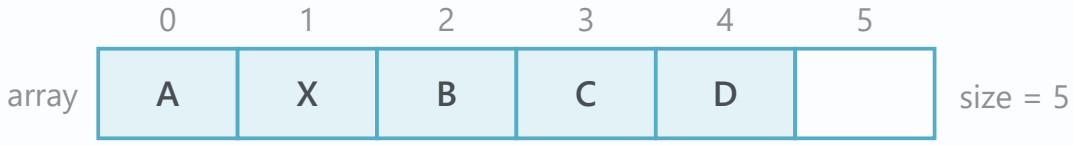
`data = [A, X, B, C, D, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `add(i, e)`



size = 5. n − i = 3 shifts → **O(n)**; inserting at the front shifts the whole list.

Step 9 / 9

```
1 public void add(int i, E e) {
2   Objects.checkIndex(i, size + 1);
3   if (size == data.length) resize();
4   for (int k = size; k > i; k--)
5     data[k] = data[k - 1];
6   data[i] = e;
▶   size++;
8 }
```

VARIABLES

`i = 1` `e = X` `k = 1` `size = 5`

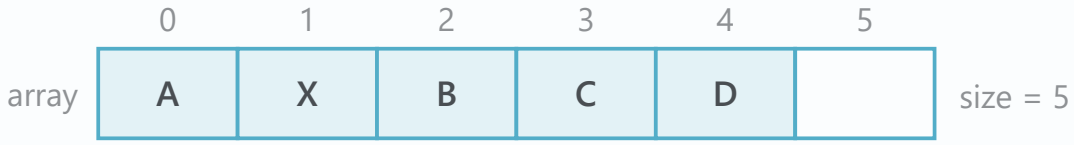
`data = [A, X, B, C, D, null]`

CALL STACK

`add(1, "X")`

`main()`

Code Trace: ArrayList `remove(i)`



The list is [A, X, B, C, D]. `remove(1)`: X goes, so B, C, D must shift left.

Step 1 / 9

```
▶ public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
4   for (int k = i; k < size - 1; k++)
5     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

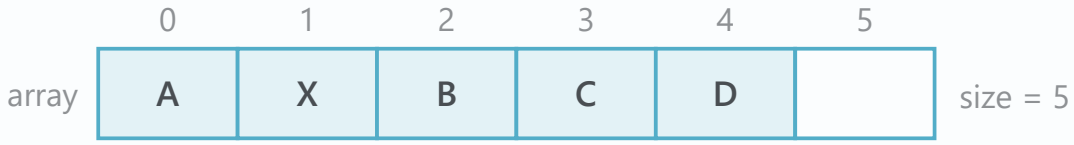
`i = 1` `old = –` `k = –` `size = 5`

`data = [A, X, B, C, D, null]`

CALL STACK

`main()`

Code Trace: ArrayList `remove(i)`



0 ≤ 1 < size (5) → a valid index.

Step 2 / 9

```
1 public E remove(int i) {
▶   Objects.checkIndex(i, size);
3   E old = data[i];
4   for (int k = i; k < size - 1; k++)
5     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = –` `k = –` `size = 5`

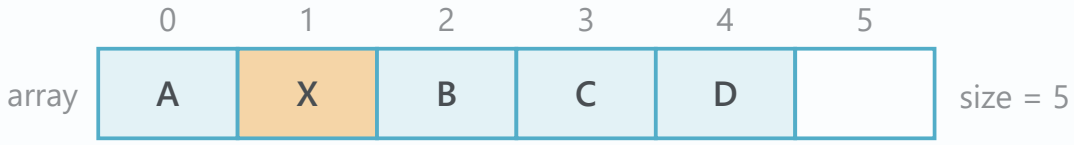
`data = [A, X, B, C, D, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



old = data[1] = X (to be returned).

Step 3 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
▶   E old = data[i];
4   for (int k = i; k < size - 1; k++)
5     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = –` `size = 5`

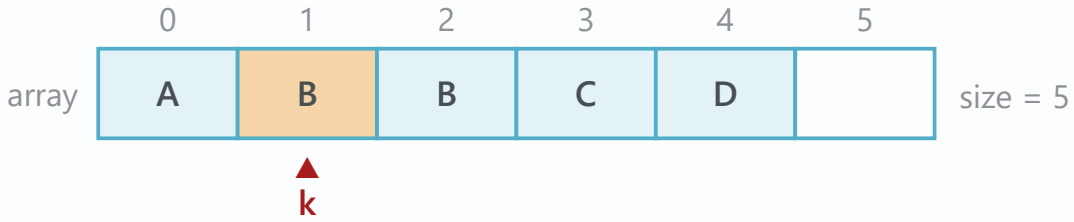
`data = [A, X, B, C, D, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



k = 1 < 4: data[1] = data[2] → B shifts left. This time we start **from the left**.

Step 4 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
▶   for (int k = i; k < size - 1; k++)
▶     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = 1` `size = 5`

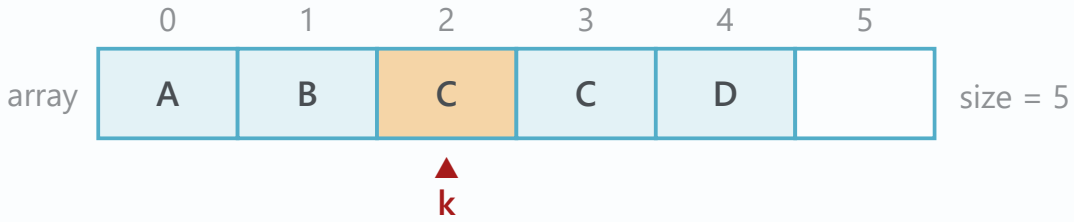
`data = [A, B, B, C, D, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



k = 2 < 4: data[2] = data[3] → C shifts left. This time we start **from the left**.

Step 5 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
▶   for (int k = i; k < size - 1; k++)
▶     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = 2` `size = 5`

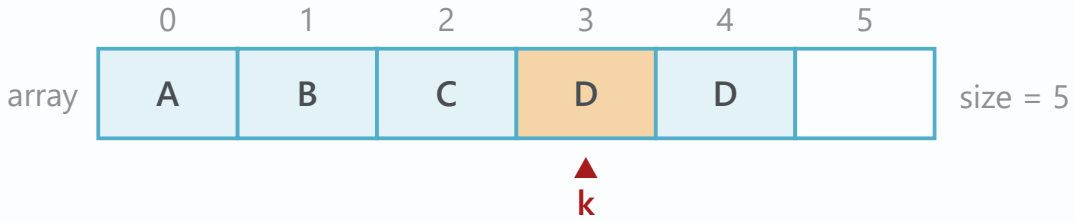
`data = [A, B, C, C, D, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



k = 3 < 4: data[3] = data[4] → D shifts left. This time we start **from the left**.

Step 6 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
▶   for (int k = i; k < size - 1; k++)
▶     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = 3` `size = 5`

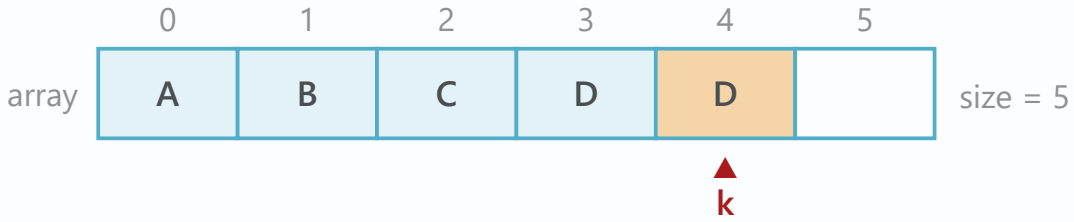
`data = [A, B, C, D, D, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



k = 4 < 4 is false → the loop ends. D now appears twice at the end.

Step 7 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
▶   for (int k = i; k < size - 1; k++)
5     data[k] = data[k + 1];
6   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = 4` `size = 5`

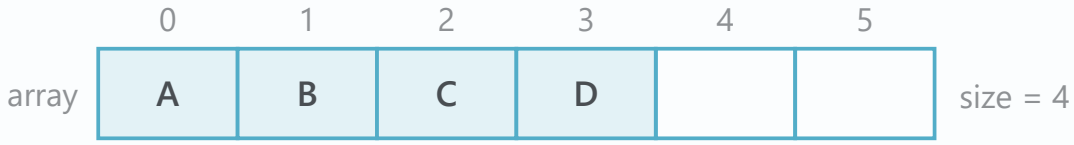
`data = [A, B, C, D, D, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



`data[--size] = null` : size 4; the extra reference is cleared (preventing a memory leak). The capacity does not shrink.

Step 8 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
4   for (int k = i; k < size - 1; k++)
5     data[k] = data[k + 1];
▶   data[--size] = null;
7   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = –` `size = 4`

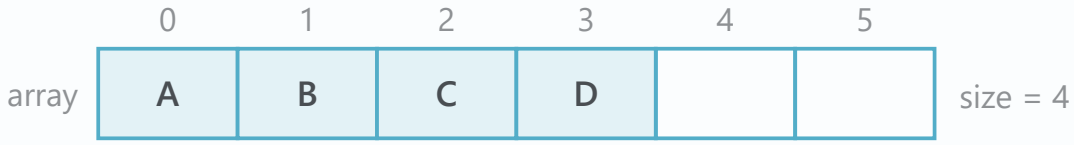
`data = [A, B, C, D, null, null]`

CALL STACK

`remove(1)`

`main()`

Code Trace: ArrayList `remove(i)`



return X. n − i − 1 = 3 shifts → **O(n)**. Removing the last element (i = size − 1) is O(1).

Step 9 / 9

```
1 public E remove(int i) {
2   Objects.checkIndex(i, size);
3   E old = data[i];
4   for (int k = i; k < size - 1; k++)
5     data[k] = data[k + 1];
6   data[--size] = null;
▶   return old;
8 }
```

VARIABLES

`i = 1` `old = X` `k = –` `size = 4`

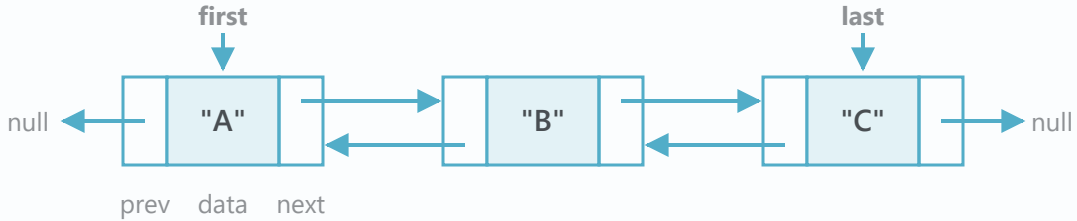
`data = [A, B, C, D, null, null]`

CALL STACK

`remove(1)`

`main()`

LinkedList: A Doubly Linked List



- Elements are not contiguous in memory: each one lives in its own **node** object.
- Each node holds its data and references to the **previous / next** nodes; the list knows `first` and `last`.
- Besides `List`, it implements `Deque`, so it can serve as a stack or a queue.
- Random access is slow: `get(i)` walks node by node (from the nearer end) → O(n).

Using LinkedList: Operations and Costs

```
LinkedList<String> list =
        new LinkedList<>();
list.add("B");          // [B]
list.addFirst("A");     // [A, B]
list.addLast("C");      // [A, B, C]
list.get(1);            // "B" (by walking)
list.removeFirst();     // [B, C]
list.size();            // 2
```

Operation	Cost
add / remove at either end	O(1)
`get(i)`, `set(i, e)`	O(n)
`add(i, e)`, `remove(i)`	O(n)*
add / remove through an iterator	O(1)

* Relinking is O(1), but first we must **walk to node i**, which takes O(n). "Insertion is fast in a LinkedList" holds only when we already have the position (an end, or a ListIterator).

Code Trace: What LinkedList Operations Cost

first = null · last = null

An empty LinkedList: no nodes; `first` and `last` are null.

Step 1 / 9

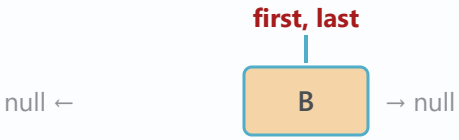
```
▶ LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
5 String s = list.get(1);
6 list.removeFirst();
7 int n = list.size();
```

VARIABLES

`list = []` `size = 0`

`first = null` `last = null`

Code Trace: What LinkedList Operations Cost



`add` = `addLast`: a single node, and both first and last point to it.

Step 2 / 9

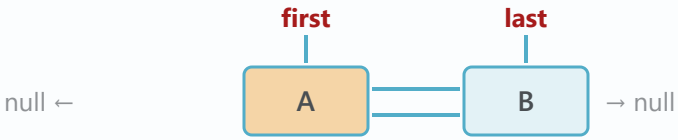
```
1 LinkedList<String> list = new LinkedList<>();
▶ list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
5 String s = list.get(1);
6 list.removeFirst();
7 int n = list.size();
```

VARIABLES

`list = [B]` `size = 1`

`first = node(B)` `last = node(B)`

Code Trace: What LinkedList Operations Cost



`addFirst` : a new node; A.next = B, B.prev = A, first = A → **O(1)**, no shifting.

Step 3 / 9

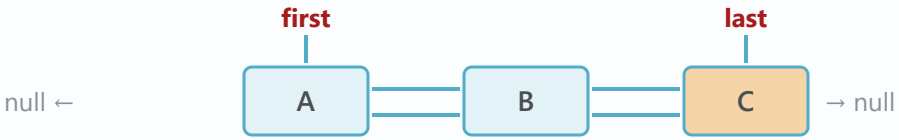
```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
▶ list.addFirst("A");
4 list.addLast("C");
5 String s = list.get(1);
6 list.removeFirst();
7 int n = list.size();
```

VARIABLES

`list = [A, B]` `size = 2`

`first = node(A)` `last = node(B)`

Code Trace: What LinkedList Operations Cost



`addLast` : thanks to last, no walk to the end is needed → O(1).

Step 4 / 9

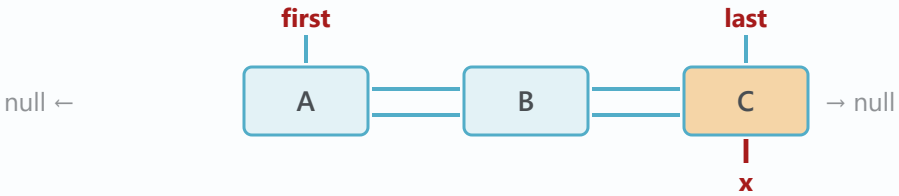
```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
▶ list.addLast("C");
5 String s = list.get(1);
6 list.removeFirst();
7 int n = list.size();
```

VARIABLES

`list = [A, B, C]` `size = 3`

`first = node(A)` `last = node(C)`

Code Trace: What LinkedList Operations Cost



`get(1)` → `node(1)` : 1 < size/2 = 1 is false → start **from the end**: x = last (C).

Step 5 / 9

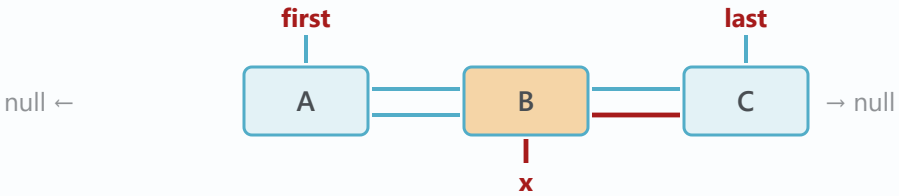
```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
▶ String s = list.get(1);
6 list.removeFirst();
7 int n = list.size();
```

VARIABLES

`list = [A, B, C]` `size = 3`

`first = node(A)` `last = node(C)`

Code Trace: What LinkedList Operations Cost



x = x.prev → B, the node we want, after walking 1 link. O(n) in general.

Step 6 / 9

```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
▶ String s = list.get(1);
6 list.removeFirst();
7 int n = list.size();
```

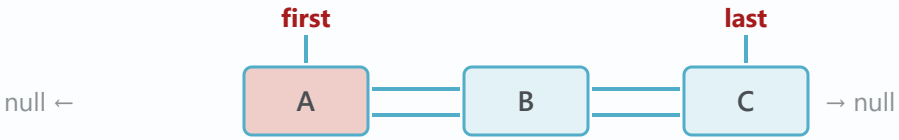
VARIABLES

`list = [A, B, C]` `size = 3`

`first = node(A)` `last = node(C)`

`s = "B"`

Code Trace: What LinkedList Operations Cost



`removeFirst` : A is about to be unlinked.

Step 7 / 9

```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
5 String s = list.get(1);
▶ list.removeFirst();
7 int n = list.size();
```

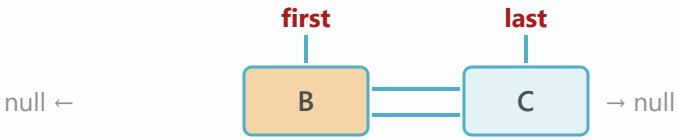
VARIABLES

`list = [A, B, C]` `size = 3`

`first = node(A)` `last = node(C)`

`s = "B"`

Code Trace: What LinkedList Operations Cost



first = B, B.prev = null; nothing refers to A any more, so the garbage collector reclaims it → O(1).

Step 8 / 9

```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
5 String s = list.get(1);
▶ list.removeFirst();
7 int n = list.size();
```

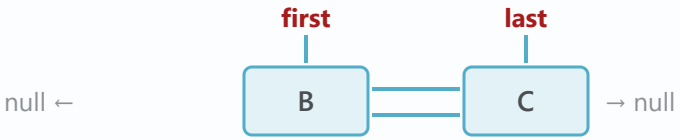
VARIABLES

`list = [B, C]` `size = 2`

`first = node(B)` `last = node(C)`

`s = "B"`

Code Trace: What LinkedList Operations Cost



`size()` reads a counter field: O(1).

Step 9 / 9

```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("B");
3 list.addFirst("A");
4 list.addLast("C");
5 String s = list.get(1);
6 list.removeFirst();
▶ int n = list.size();
```

VARIABLES

`list = [B, C]` `size = 2`

`first = node(B)` `last = node(C)`

`s = "B"` `n = 2`

Code Trace: LinkedList `addLast(e)`

first = null · last = null · size = 0

An empty LinkedList. `addLast("A")`, then `addLast("B")` (the same logic as Java's `linkLast`).

Step 1 / 12

```
▶ void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

`e = –` `l = –` `n = –` `first = null` `last = null` `size = 0`

CALL STACK

`main()`

Code Trace: LinkedList `addLast(e)`

first = null · last = null · size = 0

l = last = null: remember the old last node.

Step 2 / 12

```
1 void addLast(E e) {
▶   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

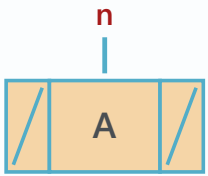
e = A | l = null | n = – | first = null | last = null | size = 0

CALL STACK

addLast("A")

main()

Code Trace: LinkedList `addLast(e)`



first = null · last = null · size = 0

A new node: prev = l (null), next = null.

Step 3 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
▶   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

`e = A` `l = null` `n = node(A)`

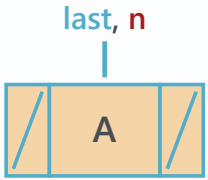
`first = null` `last = null` `size = 0`

CALL STACK

`addLast("A")`

`main()`

Code Trace: LinkedList `addLast(e)`



first = null · size = 0

`last = n` .

Step 4 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
▶   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

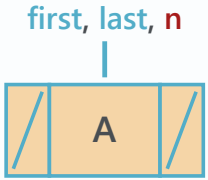
VARIABLES

`e = A` `l = null` `n = node(A)`
`first = null` `last = node(A)` `size = 0`

CALL STACK

`addLast("A")`
`main()`

Code Trace: LinkedList `addLast(e)`



size = 0

l == null (the list was empty) → `first = n` : the only node is both first and last.

Step 5 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
▶   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

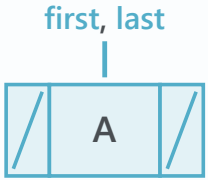
VARIABLES

`e = A` `l = null` `n = node(A)`
`first = node(A)` `last = node(A)`
`size = 0`

CALL STACK

`addLast("A")`
`main()`

Code Trace: LinkedList `addLast(e)`



size = 1

size = 1.

Step 6 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
▶   size++;
8 }
```

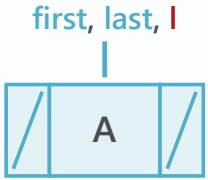
VARIABLES

`e = A` `l = null` `n = node(A)`
`first = node(A)` `last = node(A)`
`size = 1`

CALL STACK

`addLast("A")`
`main()`

Code Trace: LinkedList `addLast(e)`



size = 1

l = last = node(A): remember the old last node.


```
1 void addLast(E e) {
▶   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

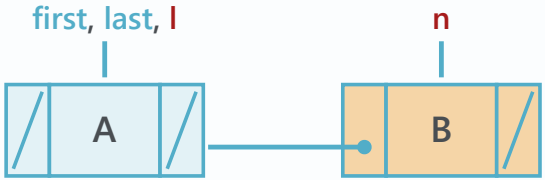
e = B | l = node(A) | n = – | first = node(A) | last = node(A) | size = 1

CALL STACK

addLast("B")

main()

Code Trace: LinkedList `addLast(e)`



size = 1

A new node: prev = l (node(A)), next = null. The back link is set at construction.

Step 8 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
▶  Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

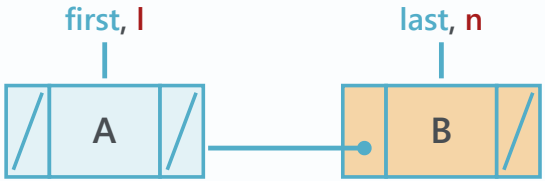
`e = B` `l = node(A)` `n = node(B)` `first = node(A)` `last = node(A)` `size = 1`

CALL STACK

`addLast("B")`

`main()`

Code Trace: LinkedList `addLast(e)`



size = 1

`last = n` .

Step 9 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
▶   last = n;
5   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

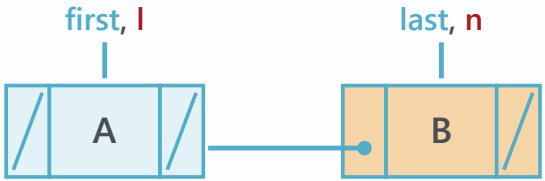
`e = B` `l = node(A)` `n = node(B)` `first = node(A)` `last = node(B)` `size = 1`

CALL STACK

`addLast("B")`

`main()`

Code Trace: LinkedList `addLast(e)`



size = 1

l is not null → else.

Step 10 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
▶   if (l == null) first = n;
6   else l.next = n;
7   size++;
8 }
```

VARIABLES

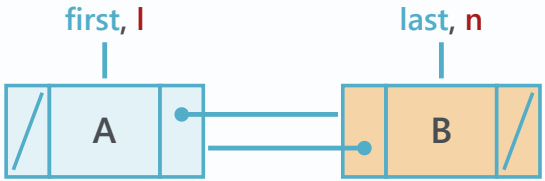
`e = B` `l = node(A)` `n = node(B)` `first = node(A)` `last = node(B)` `size = 1`

CALL STACK

`addLast("B")`

`main()`

Code Trace: LinkedList `addLast(e)`



size = 1

`l.next = n` : the forward link A → B.

Step 11 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
▶   else l.next = n;
7   size++;
8 }
```

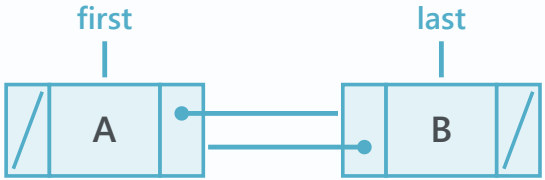
VARIABLES

`e = B` `l = node(A)` `n = node(B)`
`first = node(A)` `last = node(B)`
`size = 1`

CALL STACK

`addLast("B")`
`main()`

Code Trace: LinkedList `addLast(e)`



size = 2

size = 2. No traversal: appending is **O(1)**.

Step 12 / 12

```
1 void addLast(E e) {
2   Node<E> l = last;
3   Node<E> n = new Node<>(l, e, null);
4   last = n;
5   if (l == null) first = n;
6   else l.next = n;
▶   size++;
8 }
```

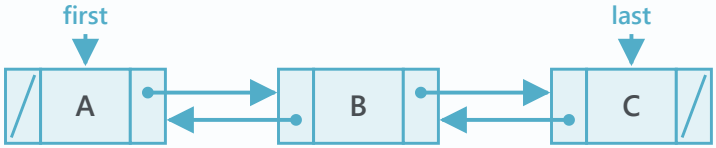
VARIABLES

`e = B` `l = node(A)` `n = node(B)`
`first = node(A)` `last = node(B)`
`size = 2`

CALL STACK

`addLast("B")`
`main()`

Code Trace: LinkedList `remove(i)`



size = 3

[A, B, C]. `remove(1)` : to find B, we first have to **walk**.

Step 1 / 10

```
E remove(int i) {
2   Objects.checkIndex(i, size);
3   Node<E> x = first;
4   for (int k = 0; k < i; k++) x = x.next;
5   Node<E> p = x.prev, s = x.next;
6   if (p == null) first = s; else p.next = s;
7   if (s == null) last = p; else s.prev = p;
8   size--;
9   return x.data;
10 }
```

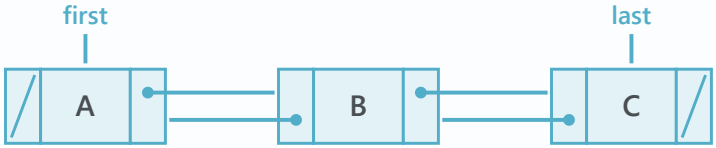
VARIABLES

`i = 1` `k = –` `x = –` `p = –` `s = –` `size = 3`

CALL STACK

`main()`

Code Trace: LinkedList `remove(i)`



size = 3

0 ≤ 1 < 3 → a valid index.

Step 2 / 10

```
1  E remove(int i) {
▶     Objects.checkIndex(i, size);
3     Node<E> x = first;
4     for (int k = 0; k < i; k++) x = x.next;
5     Node<E> p = x.prev, s = x.next;
6     if (p == null) first = s; else p.next = s;
7     if (s == null) last = p; else s.prev = p;
8     size--;
9     return x.data;
10 }
```

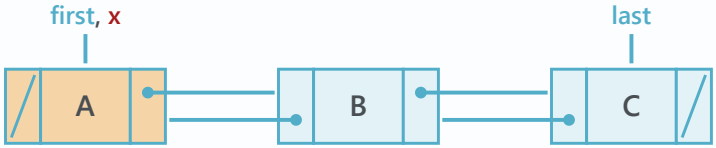
VARIABLES

`i = 1` `k = –` `x = –` `p = –` `s = –` `size = 3`

CALL STACK

- **`remove(1)`**
- `main()`

Code Trace: LinkedList `remove(i)`



size = 3

x = first (A). There is no direct indexed access as in an array.

Step 3 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
▶    Node<E> x = first;
4    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
8    size--;
9    return x.data;
10 }
```

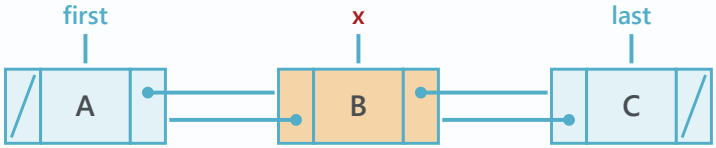
VARIABLES

`i = 1` `k = –` `x = node(A)` `p = –` `s = –` `size = 3`

CALL STACK

- **remove(1)**
- main()

Code Trace: LinkedList `remove(i)`



size = 3

k = 0 < 1: **x = x.next** → B.

Step 4 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
▶    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
8    size--;
9    return x.data;
10 }
```

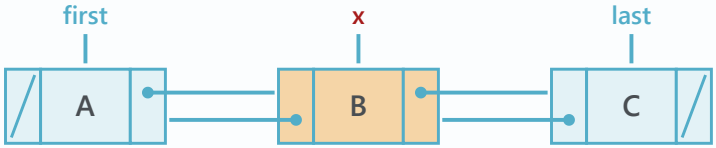
VARIABLES

`i = 1` `k = 0` `x = node(B)` `p = –` `s = –` `size = 3`

CALL STACK

- **remove(1)**
- main()

Code Trace: LinkedList `remove(i)`



size = 3

k = 1 < 1 is false → the walk ends after i steps: **O(i)**. (If i ≥ size/2, Java walks backward from last.)

Step 5 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
▶    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
8    size--;
9    return x.data;
10 }
```

VARIABLES

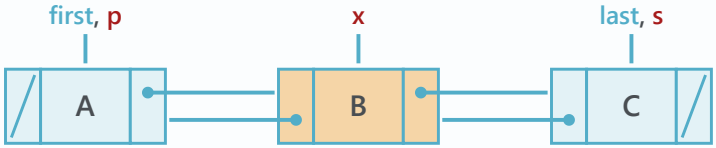
`i = 1` `k = 1` `x = node(B)` `p = –` `s = –` `size = 3`

CALL STACK

`remove(1)`

`main()`

Code Trace: LinkedList `remove(i)`



size = 3

p = x.prev = A, **s = x.next** = C: the two neighbors of the node to remove.

Step 6 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
4    for (int k = 0; k < i; k++) x = x.next;
▶    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
8    size--;
9    return x.data;
10 }
```

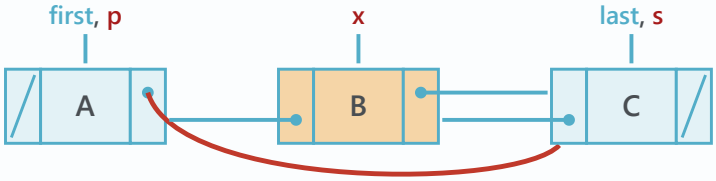
VARIABLES

`i = 1` `k = 1` `x = node(B)` `p = node(A)` `s = node(C)` `size = 3`

CALL STACK

- **remove(1)**
- main()

Code Trace: LinkedList `remove(i)`



size = 3

p is not null → `p.next = s` : A now points straight to C (B is bypassed).

Step 7 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
4    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
▶    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
8    size--;
9    return x.data;
10 }
```

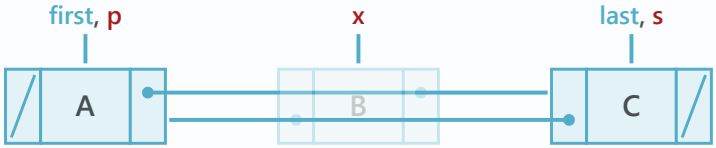
VARIABLES

`i = 1` `k = 1` `x = node(B)` `p = node(A)` `s = node(C)` `size = 3`

CALL STACK

- **remove(1)**
- main()

Code Trace: LinkedList `remove(i)`



size = 3

s is not null → `s.prev = p` : C points back to A. B is cut off in both directions.

Step 8 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
4    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
▶    if (s == null) last = p; else s.prev = p;
8    size--;
9    return x.data;
10 }
```

VARIABLES

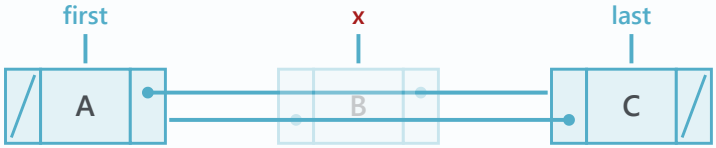
`i = 1` `k = 1` `x = node(B)` `p = node(A)` `s = node(C)` `size = 3`

CALL STACK

`remove(1)`

`main()`

Code Trace: LinkedList `remove(i)`



size = 2

size = 2.

Step 9 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
4    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
▶    size--;
9    return x.data;
10 }
```

VARIABLES

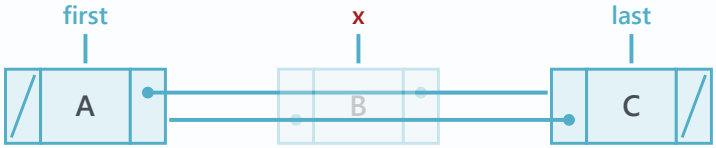
`i = 1` `k = 1` `x = node(B)` `p = node(A)` `s = node(C)` `size = 2`

CALL STACK

`remove(1)`

`main()`

Code Trace: LinkedList `remove(i)`



size = 2

return "B". Relinking is O(1), but walking is O(n) → `remove(i)` is **O(n)** overall.

Step 10 / 10

```
1  E remove(int i) {
2    Objects.checkIndex(i, size);
3    Node<E> x = first;
4    for (int k = 0; k < i; k++) x = x.next;
5    Node<E> p = x.prev, s = x.next;
6    if (p == null) first = s; else p.next = s;
7    if (s == null) last = p; else s.prev = p;
8    size--;
▶    return x.data;
10 }
```

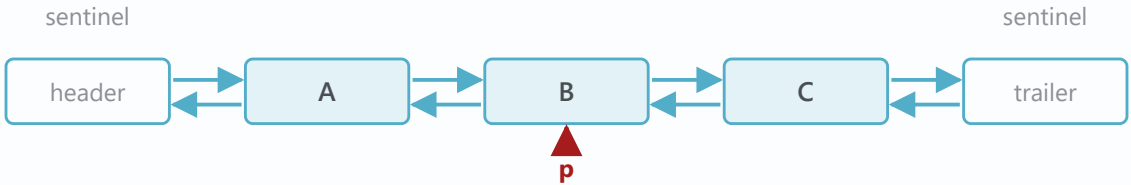
VARIABLES

`i = 1` `k = 1` `x = node(B)` `p = node(A)` `s = node(C)` `size = 2`

CALL STACK

- `remove(1)`
- `main()`

The Positional List ADT



after(p) = position of C · before(p) = position of A

- Indices **shift** on insertion and removal; a **position** stays with its element, like the cursor in a text editor.
- Goodrich–Tamassia: `Position<E>` offers only `getElement()`; in a linked implementation, a position is a node.
- Thanks to the **sentinel** nodes header and trailer, every insertion happens between two real nodes: no null checks.

Method	Cost
`first()`, `last()`, `before(p)`, `after(p)`	O(1)
`addFirst(e)`, `addLast(e)`	O(1)
`addBefore(p, e)`, `addAfter(p, e)`	O(1)
`set(p, e)`, `remove(p)`	O(1)

Java has no direct equivalent; the closest is `ListIterator` (O(1) insertion/removal next to the cursor).

Code Trace: Positional List `addAfter(p, e)`



size = 2

header ⇄ A ⇄ C ⇄ trailer; p is the position of A. We call `addAfter(p, "B")`.

Step 1 / 8

```
▶ Position<E> addAfter(Position<E> p, E e) {
2   Node<E> n = validate(p);
3   return addBetween(e, n, n.next);
4 }
5 Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6   Node<E> newest = new Node<>(pred, e, succ);
7   pred.next = newest;
8   succ.prev = newest;
9   size++;
10  return newest;
11 }
```

VARIABLES

`p = node(A)` `e = "B"` `size = 2`

CALL STACK

`addAfter(p, "B")`

`main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



size = 2

`validate(p)` : is p a valid node of this list? Yes → it is cast to a Node.

Step 2 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
▶    Node<E> n = validate(p);
3    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6    Node<E> newest = new Node<>(pred, e, succ);
7    pred.next = newest;
8    succ.prev = newest;
9    size++;
10   return newest;
11 }
```

VARIABLES

`p = node(A)` `e = "B"` `size = 2`

`n = node(A)`

CALL STACK

`addAfter(p, "B")`

`main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



The neighbors are known: pred = A, succ = A.next = C. The work is handed to `addBetween`.

Step 3 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
2    Node<E> n = validate(p);
▶    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6    Node<E> newest = new Node<>(pred, e, succ);
7    pred.next = newest;
8    succ.prev = newest;
9    size++;
10   return newest;
11 }
```

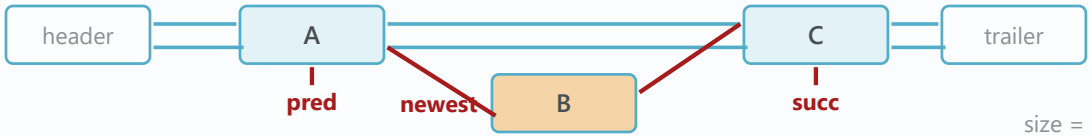
VARIABLES

`p = node(A)` `e = "B"` `size = 2`
`n = node(A)` `pred = node(A)`
`succ = node(C)`

CALL STACK

`addBetween("B", A, C)`
`addAfter(p, "B")`
`main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



The new node points to both neighbors, but nothing points to it yet.

Step 4 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
2    Node<E> n = validate(p);
3    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
▶    Node<E> newest = new Node<>(pred, e, succ);
7    pred.next = newest;
8    succ.prev = newest;
9    size++;
10   return newest;
11 }
```

VARIABLES

`p = node(A)` `e = "B"` `size = 2`

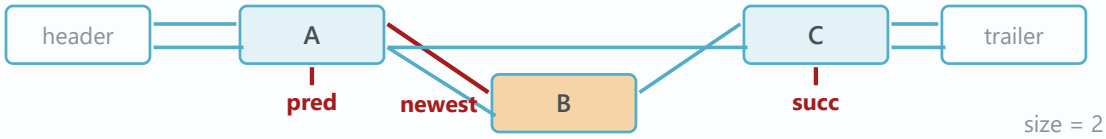
`pred = node(A)` `succ = node(C)`

`newest = node(B)`

CALL STACK

- `addBetween("B", A, C)`
- `addAfter(p, "B")`
- `main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



pred.next = newest: forward, A → B → C.

Step 5 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
2    Node<E> n = validate(p);
3    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6    Node<E> newest = new Node<>(pred, e, succ);
▶    pred.next = newest;
8    succ.prev = newest;
9    size++;
10   return newest;
11 }
```

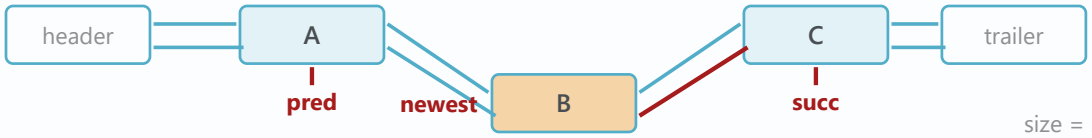
VARIABLES

`p = node(A)` `e = "B"` `size = 2`
`pred = node(A)` `succ = node(C)`
`newest = node(B)`

CALL STACK

`addBetween("B", A, C)`
`addAfter(p, "B")`
`main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



succ.prev = newest: backward, C → B → A. The old A ⇄ C links are gone.

Step 6 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
2    Node<E> n = validate(p);
3    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6    Node<E> newest = new Node<>(pred, e, succ);
7    pred.next = newest;
▶    succ.prev = newest;
9    size++;
10   return newest;
11 }
```

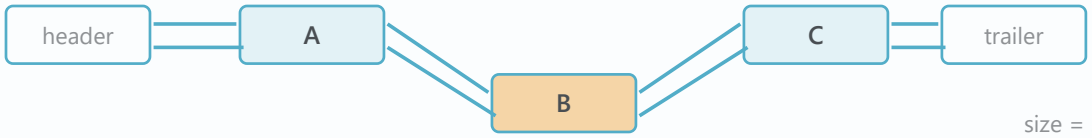
VARIABLES

- `p = node(A)`
- `e = "B"`
- `size = 2`
- `pred = node(A)`
- `succ = node(C)`
- `newest = node(B)`

CALL STACK

- `addBetween("B", A, C)`
- `addAfter(p, "B")`
- `main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



size++. No walking, no shifting: 4 assignments → **O(1)**.

Step 7 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
2    Node<E> n = validate(p);
3    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6    Node<E> newest = new Node<>(pred, e, succ);
7    pred.next = newest;
8    succ.prev = newest;
▶    size++;
10   return newest;
11 }
```

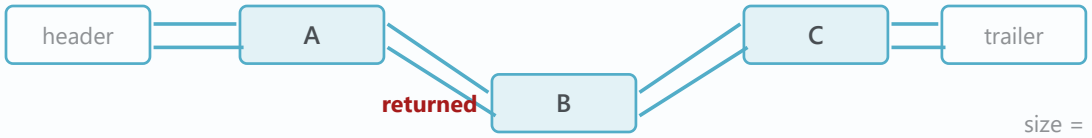
VARIABLES

`p = node(A)` `e = "B"` `size = 3`
`pred = node(A)` `succ = node(C)`
`newest = node(B)`

CALL STACK

- `addBetween("B", A, C)`
- `addAfter(p, "B")`
- `main(): list.addAfter(p, "B")`

Code Trace: Positional List `addAfter(p, e)`



It returns the new element's **position**, ready for later calls such as `addBefore` or `remove`.

Step 8 / 8

```
1  Position<E> addAfter(Position<E> p, E e) {
2    Node<E> n = validate(p);
3    return addBetween(e, n, n.next);
4  }
5  Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
6    Node<E> newest = new Node<>(pred, e, succ);
7    pred.next = newest;
8    succ.prev = newest;
9    size++;
▶    return newest;
11 }
```

VARIABLES

`p = node(A)` `e = "B"` `size = 3`

`pred = –` `succ = –`

`newest = node(B)`

CALL STACK

`main(): list.addAfter(p, "B")`

LIST IMPLEMENTATIONS

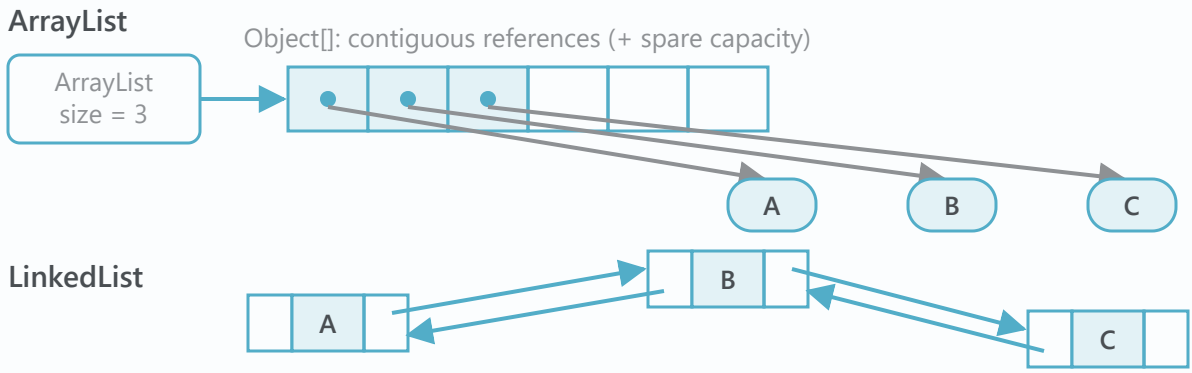
ArrayList or LinkedList?

	ArrayList	LinkedList
Internals	growable array	doubly linked nodes
get(i)	O(1)	O(n)
Append	amortized O(1)	O(1)
Add / remove at the front	O(n)	O(1)
Memory per element	1 reference (+ spare capacity)	a node object (3 references)
Cache behavior	contiguous, fast	scattered, slow

Default: ArrayList
Faster in most scenarios (read-heavy, appending at the end).

LinkedList / ArrayDeque
When you add and remove at both ends often (queue, deque).

ArrayList and LinkedList in Memory



ArrayList

About 1 reference per element (4–8 bytes) + spare capacity. The array is contiguous, so the CPU cache prefetches the next elements.

LinkedList

One node object per element: an object header + 3 references (~24–40 bytes). Following each `next` may cause a cache miss.

LIST IMPLEMENTATIONS

Vector

- A growable array like ArrayList: indexed access, grows as needed (by default the capacity **doubles**).
- Its methods are `synchronized`, so individual operations are thread-safe.
- The price: every call pays for a lock, even with a single thread.
- **Compound** operations such as "check, then act" are still not atomic.

```
Vector<Integer> v = new Vector<>();
v.add(10);              // [10]
v.add(20);              // [10, 20]
v.get(0);               // 10
v.remove(0);            // [20]
v.size();               // 1
v.capacity();           // 10

// compound action: needs an external lock
synchronized (v) {
    if (!v.isEmpty()) v.remove(0);
}
```

Vector is a legacy class from Java 1.0. New code should use `ArrayList` for a single thread, and `Collections.synchronizedList` or `CopyOnWriteArrayList` for many.

Vector: The "Check, Then Act" Race

Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:



Start: v = [20]. Both threads will run `if (!v.isEmpty()) v.remove(0);`.

Vector: The "Check, Then Act" Race

Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:



T1: `isEmpty()` takes the lock, returns false, and **releases** the lock.

Vector: The "Check, Then Act" Race

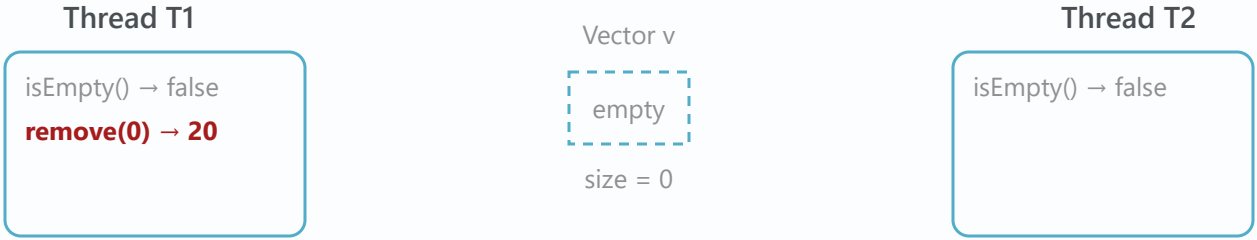
Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:



T2 slips in: v still has 1 element, so it sees false too. Both think they can remove.

Vector: The "Check, Then Act" Race

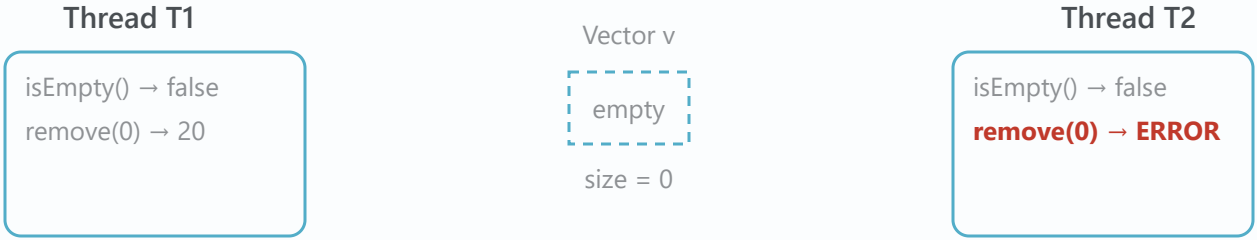
Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:



T1: `remove(0)` → 20. v is now empty.

Vector: The "Check, Then Act" Race

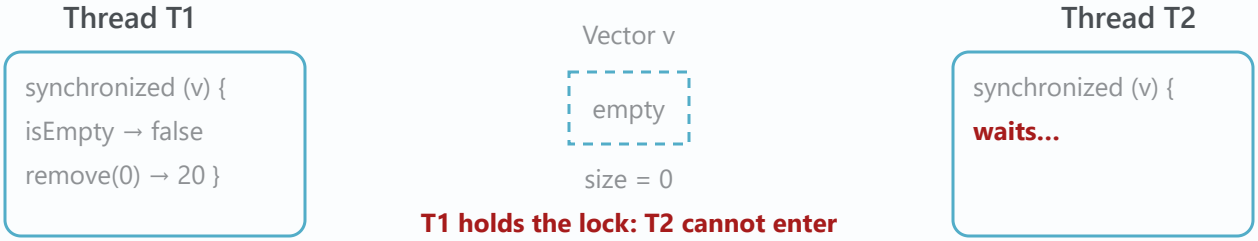
Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:



T2: `remove(0)` on an empty vector → `ArrayIndexOutOfBoundsException`. Each method is synchronized on its own, but the **pair** is not atomic.

Vector: The "Check, Then Act" Race

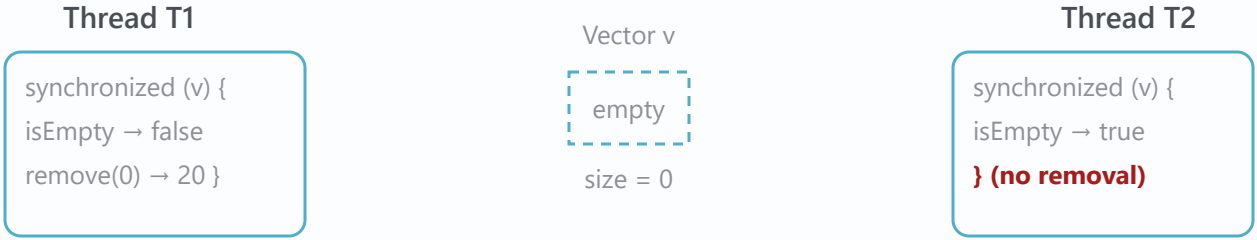
Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:



The fix: put the compound action under one lock with `synchronized (v) { … }`. T2 waits until T1 leaves the block.

Vector: The "Check, Then Act" Race

Every Vector method is `synchronized`, but the gap **between** two calls is unprotected. One possible interleaving of two threads:

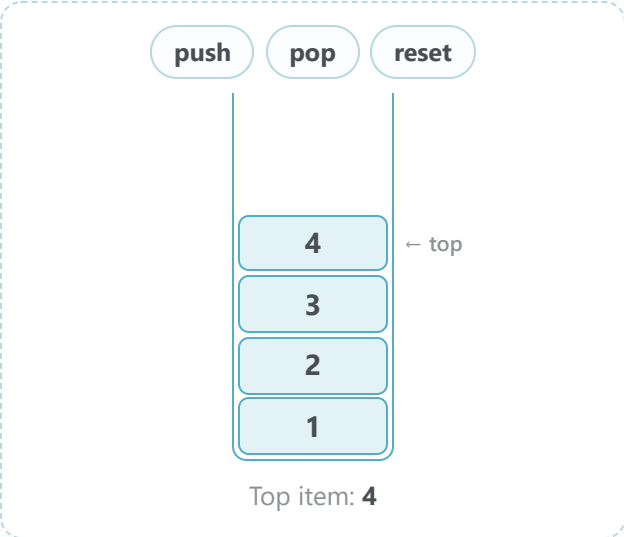


When T2 gets the lock, it sees v is empty and does not remove: "check, then act" is now safe.

The Stack Class

- Extends Vector and implements a **last-in, first-out** (LIFO) stack.
- Operations act on the **top** element; handy for undo/redo.

`push(e)`	adds on top
`pop()`	removes and returns the top
`peek()`	returns the top without removing it
`empty()`	is the stack empty?
`search(o)`	distance from the top, **starting at 1**; −1 if absent

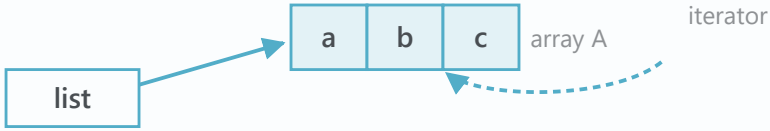


`pop` / `peek` on an empty stack → `EmptyStackException`.

Because Stack extends Vector, it also carries methods that break LIFO, such as `get(i)` and `add(i, e)`, and it pays for synchronization. The Java documentation recommends `Deque<E> s = new ArrayDeque<>();` for a stack.

CopyOnWriteArrayList: Copy on Write

- On every change (add, set, remove), the internal array is **copied**, the change is made on the copy, and the list then switches to the new array.
- Reads take no lock and are very fast; writes queue up behind a short lock and pay an **O(n)** copy.



Readers read the internal array without locking; each iterator walks the array as it was when the iterator was created (a snapshot).

Step 1 / 3

Good for

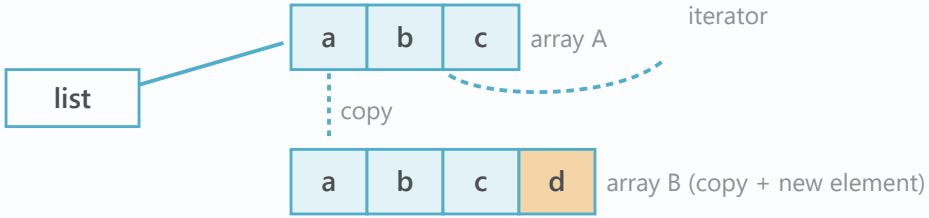
Lists that are traversed often and rarely change: event listeners, configuration.

Bad for

Large lists that change often: every write copies the whole array. `it.remove()` is not supported.

CopyOnWriteArrayList: Copy on Write

- On every change (add, set, remove), the internal array is **copied**, the change is made on the copy, and the list then switches to the new array.
- Reads take no lock and are very fast; writes queue up behind a short lock and pay an **O(n)** copy.



`add("d")` : the writer takes the lock, **copies** the array, and appends the new element to the copy → O(n)

Good for

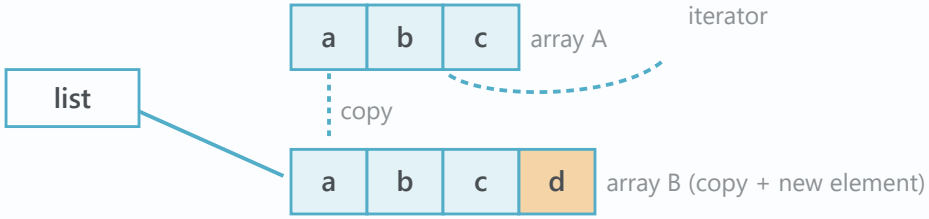
Lists that are traversed often and rarely change: event listeners, configuration.

Bad for

Large lists that change often: every write copies the whole array. `it.remove()` is not supported.

CopyOnWriteArrayList: Copy on Write

- On every change (add, set, remove), the internal array is **copied**, the change is made on the copy, and the list then switches to the new array.
- Reads take no lock and are very fast; writes queue up behind a short lock and pay an **O(n)** copy.



The list reference switches to the new array. The old iterator keeps walking A: no `ConcurrentModificationException`.

Step 3 / 3

Good for

Lists that are traversed often and rarely change: event listeners, configuration.

Bad for

Large lists that change often: every write copies the whole array. `it.remove()` is not supported.

Code Trace: CopyOnWriteArrayList `add(e)`



The list points to A = [a, b, c]. An iterator has walked over A and read "a" (cursor 1). Now `add("d")`.

Step 1 / 10

```
▶ public boolean add(E e) {
2   synchronized (lock) {
3     Object[] old = array;
4     int n = old.length;
5     Object[] copy = Arrays.copyOf(old, n + 1);
6     copy[n] = e;
7     array = copy;       // one reference assignment
8     return true;
9   }
10 }
```

VARIABLES

`e = –` `old = –` `n = –` `copy = –` `list.array = A`

CALL STACK

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`

lock: held by writer



Writers take the lock, so two writers cannot overwrite each other's copies. **Readers never wait for the lock.**

Step 2 / 10

```
1  public boolean add(E e) {
▶    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;       // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

`e = d` `old = –` `n = –` `copy = –` `list.array = A`

CALL STACK

`add("d")`

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`

lock: held by writer



old = array: the current array, A.

Step 3 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
▶      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;       // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

`e = d` `old = A` `n = –` `copy = –` `list.array = A`

CALL STACK

`add("d")`

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`

lock: held by writer



n = 3.

Step 4 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
▶      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;       // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

`e = d` `old = A` `n = 3` `copy = –` `list.array = A`

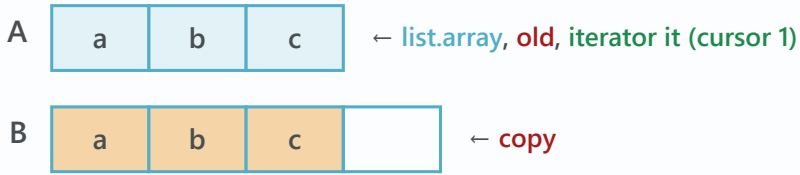
CALL STACK

`add("d")`

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`

lock: held by writer



`Arrays.copyOf(old, 4)` : every element is copied into the **copy** array → O(n). A is left untouched.

Step 5 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
▶      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;        // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

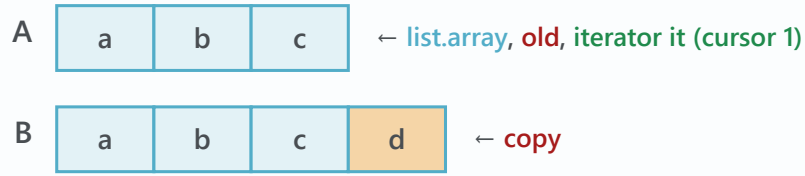
`e = d` `old = A` `n = 3` `copy = B` `list.array = A`

CALL STACK

- `add("d")`
- `main()`

Code Trace: CopyOnWriteArrayList `add(e)`

lock: held by writer



`copy[3] = "d"` : the change happens only in the copy.

Step 6 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
▶      copy[n] = e;
7      array = copy;        // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

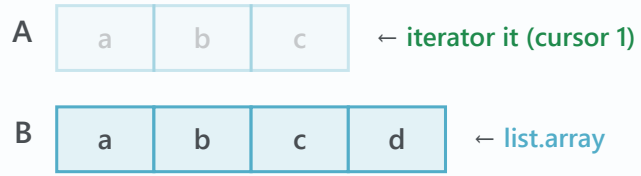
`e = d` `old = A` `n = 3` `copy = B` `list.array = A`

CALL STACK

- `add("d")`
- `main()`

Code Trace: CopyOnWriteArrayList `add(e)`

lock: held by writer



`array = copy` : a single reference assignment; the list now points to B. The iterator is still on A.

Step 7 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
▶      array = copy;        // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

`e = d` `old = A` `n = 3` `copy = B`

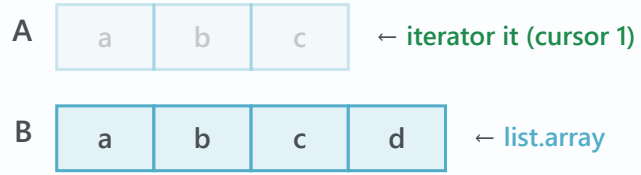
`list.array = B`

CALL STACK

`add("d")`

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`



Returns true and releases the lock.

Step 8 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;       // one reference assignment
▶      return true;
9    }
10 }
```

VARIABLES

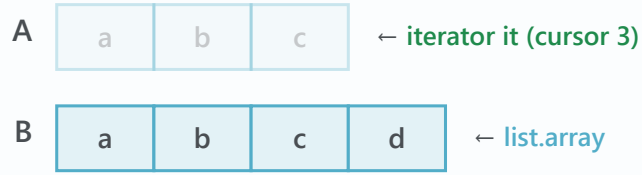
`e = d` `old = A` `n = 3` `copy = B` `list.array = B`

CALL STACK

`add("d")`

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`



The iterator's `it.next()` → "b", then "c": it walks its **snapshot** (A), never sees "d", and throws no `ConcurrentModificationException`.

Step 9 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;        // one reference assignment
8      return true;
9    }
10 }
```

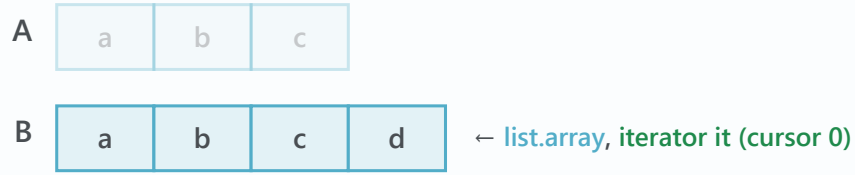
VARIABLES

`list.array = B` `reader = A`

CALL STACK

`main()`

Code Trace: CopyOnWriteArrayList `add(e)`



A new `iterator()` sees B: [a, b, c, d]. Once the old iterator is done, nothing refers to A, and the garbage collector reclaims it.

Step 10 / 10

```
1  public boolean add(E e) {
2    synchronized (lock) {
3      Object[] old = array;
4      int n = old.length;
5      Object[] copy = Arrays.copyOf(old, n + 1);
6      copy[n] = e;
7      array = copy;       // one reference assignment
8      return true;
9    }
10 }
```

VARIABLES

`list.array = B` `reader = A`

CALL STACK

`main()`

LIST IMPLEMENTATIONS

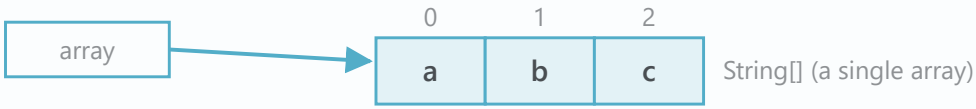
`Arrays.asList()`: A List View of an Array

- Lets us use an array through the `List` interface; nothing is **copied**, and the array and the list share the same memory.
- The list is **fixed-size** but **not immutable**: `set` works (and shows up in the array); `add` / `remove` do not.

```
String[] array = {"a", "b", "c"};
List<String> list = Arrays.asList(array);
list.set(0, "x");     // works: array[0] is now "x" too
array[2] = "z";          // the list changes too → [x, b, z]
list.add("d");         // UnsupportedOperationException (fixed size)
List<String> copy = new ArrayList<>(Arrays.asList(array));   // independent copy
```

	set	add / remove	Link to the array
`Arrays.asList(array)`	✓	✗	shared
`List.of(...)`	✗	✗	none (immutable)
`new ArrayList<>(...)`	✓	✓	none (copy)

Code Trace: The `Arrays.asList` View



An array of 3 elements.

Step 1 / 8

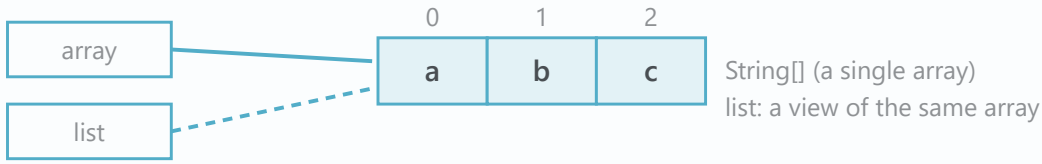
```
▶ String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
3 list.set(0, "x");
4 array[2] = "z";
5 // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
7 copy.add("d");
8 System.out.println(list + " " + copy);
```

VARIABLES

`array = [a, b, c]` `list = –` `copy = –`

OUTPUT

Code Trace: The `Arrays.asList` View



`Arrays.asList` copies nothing: the returned list is a view wrapping **the same array** → O(1).

Step 2 / 8

```
1 String[] array = {"a", "b", "c"};
▶ List<String> list = Arrays.asList(array);
3 list.set(0, "x");
4 array[2] = "z";
5 // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
7 copy.add("d");
8 System.out.println(list + " " + copy);
```

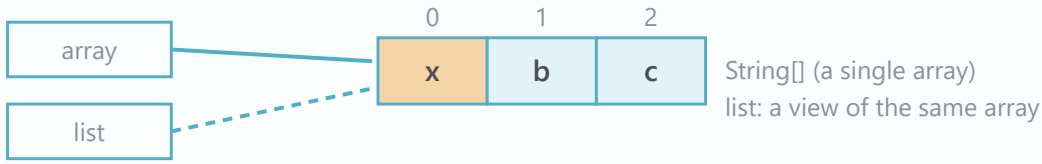
VARIABLES

`array = [a, b, c]`

`list = [a, b, c]` `copy = –`

OUTPUT

Code Trace: The `Arrays.asList` View



`list.set(0, "x")` writes into the array: `array[0]` is x too.

Step 3 / 8

```
1 String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
▶ list.set(0, "x");
4 array[2] = "z";
5 // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
7 copy.add("d");
8 System.out.println(list + " " + copy);
```

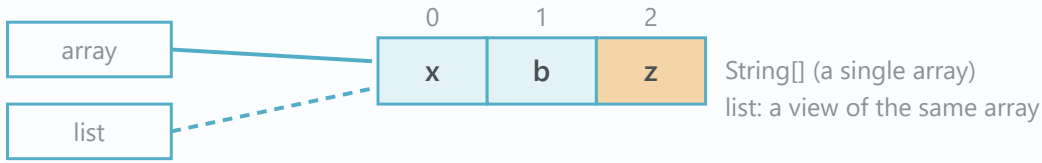
VARIABLES

`array = [x, b, c]`

`list = [x, b, c]` `copy = –`

OUTPUT

Code Trace: The `Arrays.asList` View



It works the other way too: writing to the array changes the list.

Step 4 / 8

```
1 String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
3 list.set(0, "x");
▶ array[2] = "z";
5 // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
7 copy.add("d");
8 System.out.println(list + " " + copy);
```

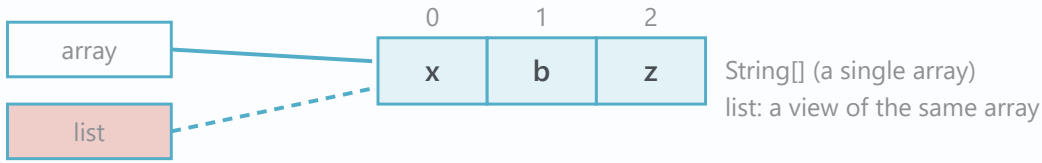
VARIABLES

`array = [x, b, z]`

`list = [x, b, z]` `copy = –`

OUTPUT

Code Trace: The `Arrays.asList` View



If uncommented: the array's length cannot change, so `add` / `remove` → `UnsupportedOperationException`.

Step 5 / 8

```
1 String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
3 list.set(0, "x");
4 array[2] = "z";
▶ // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
7 copy.add("d");
8 System.out.println(list + " " + copy);
```

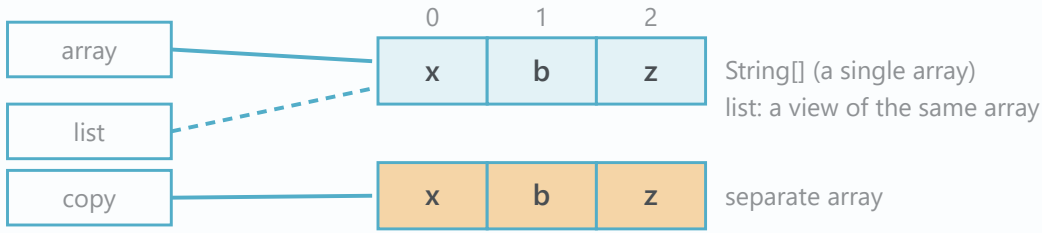
VARIABLES

`array = [x, b, z]`

`list = [x, b, z]` `copy = –`

OUTPUT

Code Trace: The `Arrays.asList` View



For an independent, growable list, copy it: `new ArrayList<>(list)` allocates a new array.

Step 6 / 8

```
1 String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
3 list.set(0, "x");
4 array[2] = "z";
5 // list.add("d");  → error (fixed size)
▶ List<String> copy = new ArrayList<>(list);
7 copy.add("d");
8 System.out.println(list + " " + copy);
```

VARIABLES

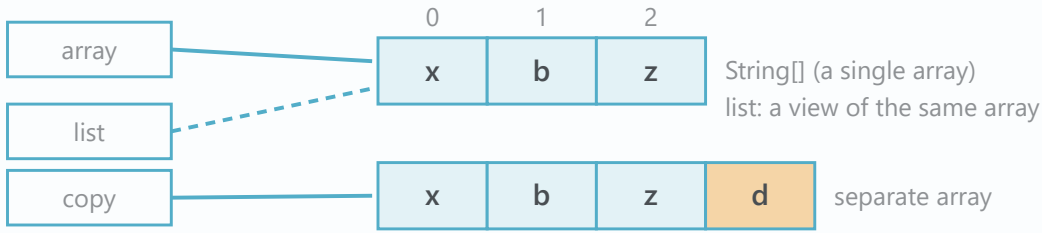
`array = [x, b, z]`

`list = [x, b, z]`

`copy = [x, b, z]`

OUTPUT

Code Trace: The `Arrays.asList` View



Adding to the copy is fine; `array` and `list` are unaffected.

Step 7 / 8

```
1 String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
3 list.set(0, "x");
4 array[2] = "z";
5 // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
▶ copy.add("d");
8 System.out.println(list + " " + copy);
```

VARIABLES

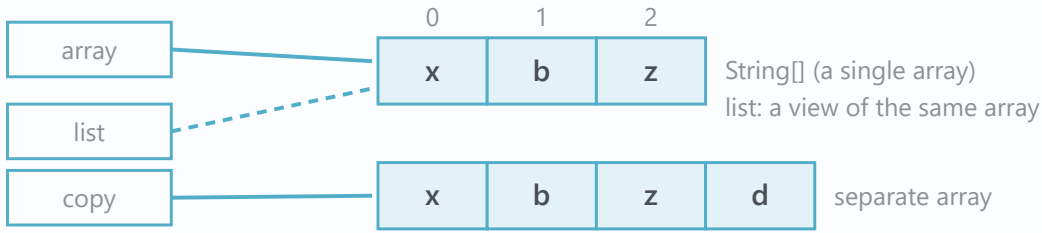
`array = [x, b, z]`

`list = [x, b, z]`

`copy = [x, b, z, d]`

OUTPUT

Code Trace: The `Arrays.asList` View



The result: two separate lists.

Step 8 / 8

```
1 String[] array = {"a", "b", "c"};
2 List<String> list = Arrays.asList(array);
3 list.set(0, "x");
4 array[2] = "z";
5 // list.add("d");  → error (fixed size)
6 List<String> copy = new ArrayList<>(list);
7 copy.add("d");
▶ System.out.println(list + " " + copy);
```

VARIABLES

`array = [x, b, z]`

`list = [x, b, z]`

`copy = [x, b, z, d]`

OUTPUT

```
[x, b, z] [x, b, z, d]
```

SUMMARY

Summary: List Implementations

Class	get(i)	append	front / middle	Threads
ArrayList	O(1)	amortized O(1)	O(n)	not safe
LinkedList	O(n)	O(1)	O(1) / O(n)	not safe
Vector	O(1)	amortized O(1)	O(n)	synchronized
Stack	O(1)	push O(1)	—	synchronized
CopyOnWriteArrayList	O(1)	O(n)	O(n)	safe (by copying)
Arrays.asList	O(1)	✗	✗	not safe

Traversal

for-each / Iterator is O(n) on every list. To remove while traversing, use `it.remove()` or `removeIf`.

Interactive simulator

List ADT operations, step by step: List ADT Simulator.

SUMMARY

Sample Problems

1. What is the list after `List<Integer> l = new ArrayList<>(List.of(5, 6, 7)); l.remove(1);` ?
2. What does it cost to traverse a `LinkedList` with `for (int i…) get(i)` ? What is better?
3. What happens when we add an element to an ArrayList with capacity 10 that holds 10 elements?
4. Why does `Arrays.asList(array).add(x)` fail? Why doesn't `set` ?
5. Which class suits a listener list that is read often, rarely changes, and is shared by many threads?
6. On [A, B, C], after `it = listIterator(); it.next(); it.add("X");` , what are the list and `it.next()` ?

▶ **Show answers**

SUMMARY

Exercises

1. `l = [A, B, C, D]`; what is `l` after `l.subList(1, 3).clear();`? And what if we then call `l.add("E")` and read the old view?
2. A dynamic array **grows by 2** every time it fills up. How many copies do n adds make in total?
3. Which contract would break if `Range.next()` skipped the `k >= end` check?
4. Why is `remove(p)` O(1) in a positional list but `LinkedList.remove(int)` O(n)?
5. `l = new ArrayList<>(List.of(3, 1, 2)); l.remove(Integer.valueOf(1)); l.remove(1);` → `l`?

▶ **Answers / hints**

SUMMARY

Further Reading

- Oracle, Java SE 17 API documentation for `Collection`, `List`, `Iterator`, `ListIterator`, and `Spliterator`.
- Bloch, *Effective Java*, 3rd ed., 2018.
- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.

The End

Chapter 6: The List ADT and Java Collections

Next deck: Chapter 7: Stacks · Study questions · Simulator: List ADT Simulator · Glossary