Chapter 5: Collections

Object Oriented Programming with Java

The collections framework | Generic types | ArrayList and LinkedList | Vector and Stack | Queue and PriorityQueue | Deque and ArrayDeque | Set: HashSet, TreeSet | Map: HashMap, TreeMap | Iterator | Sorting and searching | Shuffling and composition | The Stream API

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_05_Collections.html

Learning Outcomes

Framework

Explain the interfaces, implementations and algorithms of the collections framework and its interface/class hierarchy.

Generic types

Write type-safe code with generic types such as `List<String>`; use wrapper classes and autoboxing.

Lists

Use `ArrayList`, `LinkedList`, `Vector` and `Stack`; explain their performance differences.

Queues

Build FIFO, priority and LIFO behavior with `Queue`, `PriorityQueue` and `ArrayDeque`.

Sets and maps

Choose correctly among `HashSet`, `TreeSet`, `HashMap`, `TreeMap` and their linked versions.

Iteration, algorithms, streams

Iterate safely with `Iterator`; sort, shuffle and search with `Collections`; build filter, map and reduce pipelines with `java.util.stream`.

Why Collections?

- **Collection**: a data structure that stores and manages several elements in a single object.
- **Java Collections Framework**: the ready-made interfaces, classes and algorithms in package `java.util`.
- Powerful, well-tested tools for storing, sorting, searching and processing data, optimized for performance and safety.
- Unlike an array, the size is **dynamic**, and methods such as `add`, `remove` and `contains` come built in.
- Picking the right structure matters: each collection type is designed for a different need.

ARRAY: SIZE FIXED UP FRONT

```
int[] arr = new int[3];      // size fixed: 3
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
// arr[3] = 40;  → ArrayIndexOutOfBoundsException
```

LIST: GROWS AS NEEDED

```
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);       // no problem: it grows
System.out.println(list);
System.out.println("size: " + list.size());
```

```
[10, 20, 30, 40]
size: 4
```

Analogy: an array is a bus with a fixed number of seats; an `ArrayList` is a bus that grows as passengers board.

Kinds of Collections

List

Ordered, indexed elements; duplicates allowed. `ArrayList`, `LinkedList`



Set

Unique elements only; the same element is not added twice. `HashSet`, `TreeSet`



Map

Key → value pairs; each key appears once. `HashMap`, `TreeMap`



Queue

Elements waiting to be processed; usually first in, first out (FIFO). `LinkedList`, `PriorityQueue`



Core Operations and Design Goals

Core operations

Add, remove, update and search; easy iteration with a `for-each` loop.

Ease of use

A simple, clear API: code is easier to read and maintain.

Performance

Memory-efficient structures optimized for data volume and access patterns.

Type safety

Generic types (`List<String>`) catch elements of the wrong type at compile time.

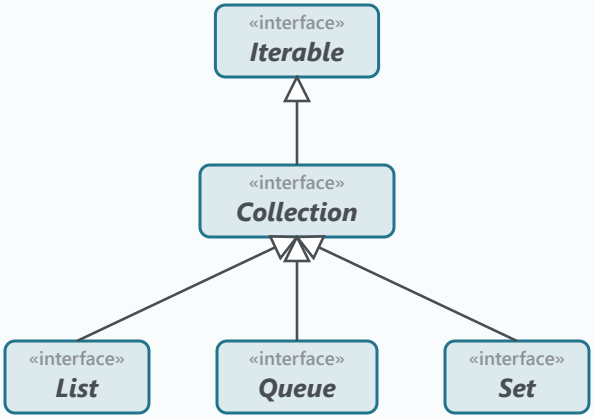
Rich hierarchy

A family of classes for different needs, working smoothly with the rest of the standard library.

Ready-made algorithms

Sorting, searching and shuffling are built in; misuse is reported with exceptions. They save time and make software more reliable.

Collection Interfaces



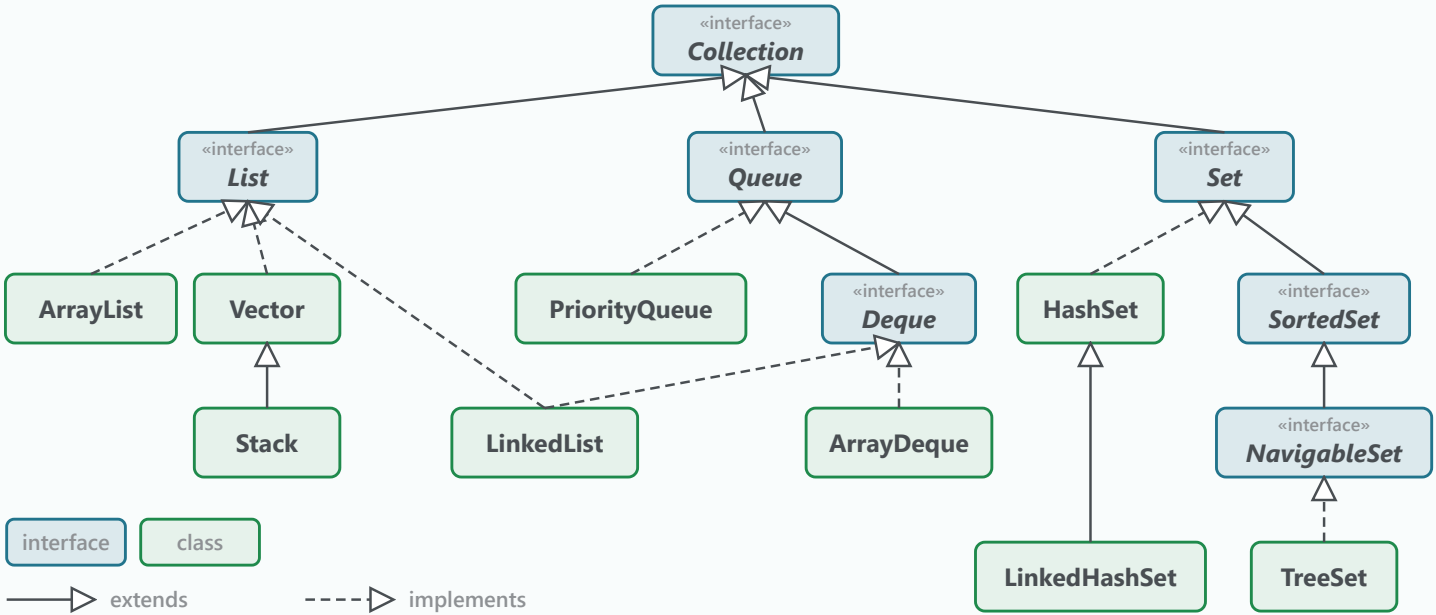
«interface»
Map

separate hierarchy
(not a Collection)

- `Iterable` : anything you can loop over with `for-each` (the `iterator()` method).
- `Collection` : methods common to all collections (`add` , `remove` , `size` ...).
- `List` , `Queue` , `Set` : subinterfaces that extend `Collection` .
- `Map` : a key–value structure. It is part of the framework but does **not extend** `Collection` .

The arrows follow UML: a hollow triangle points to the extended (parent) interface.

Collection Classes



`LinkedList` implements both `List` and `Deque`. Blue boxes are interfaces (behavior only); green boxes are classes you can instantiate with `new`.

Three Parts of the Framework

Interface

Defines **how a data structure should behave** (what it does). Example: `List` represents an ordered list.

Implementation

The actual class that implements the interface (how it does it). Example: `ArrayList` implements `List` with a dynamic array.

Algorithm

Ready-made methods for operations such as sorting and searching. Example: `Collections.sort()` sorts a list.

```
List<String> list = new ArrayList<>();  // interface ← class
list.add("Red");
list.add("Blue");
Collections.sort(list);                 // algorithm
System.out.println(list);
```

```
[Blue, Red]
```

Modular design: each component has a clear role and responsibility. **Flexibility**: different implementations and algorithms work together through the same interface.

Programming to an Interface

- An interface defines a **behavior**; classes implement it with their own methods.
- Different kinds of storage: `ArrayList` is a dynamic array, `LinkedList` a doubly linked list, `HashMap` a key–value table.
- Each class meets the needs of a particular data structure and does a particular job.
- Declare variables and parameters with the **interface type**: the implementation can be swapped in one line, while the rest of the code and the basic behavior stay the same.

```
public class PrintList {
    // interface parameter: works with any List
    static void print(List<String> list) {
        for (String s : list) {
            System.out.print(s + " ");
        }
        System.out.println("(" + list.size() + " items)");
    }

    public static void main(String[] args) {
        List<String> a = new ArrayList<>();
        a.add("Apple");
        a.add("Pear");
        List<String> b = new LinkedList<>(a);  // a copy
        b.add("Cherry");
        print(a);
        print(b);
    }
}
```

```
Apple Pear (2 items)
Apple Pear Cherry (3 items)
```

Generic Types and Type Safety

```
List<String> names = new ArrayList<>();
names.add("Ali");
// names.add(42);           // compile error
String first = names.get(0);  // no cast needed

List<Integer> grades = new ArrayList<>();  // no List<int>
grades.add(85);               // boxing: int → Integer
int n = grades.get(0);        // unboxing → int
System.out.println(first + " " + n);
```

```
Ali 85
```

- The **element type** goes inside `<>`: `List<String>` means "list of String".
- A wrong type is caught at compile time; the result of `get` needs no cast.
- The empty `<>` on the right is the diamond operator (Java 7+); the type is inferred from the left side.

COLLECTIONS HOLD OBJECTS ONLY

Primitive type	Wrapper class
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`long`	`Long`

Primitive types use wrapper classes. The conversion is automatic: **autoboxing** and **unboxing**.

First Examples

```
// list of names (ArrayList)
List<String> nameList = new ArrayList<>();
nameList.add("Ahmet");
nameList.add("Ayşe");
System.out.println(nameList);

// phone book (HashMap)
Map<String, String> phoneBook = new HashMap<>();
phoneBook.put("Ahmet", "555-1234");
phoneBook.put("Ayşe", "555-9876");
System.out.println(phoneBook.get("Ahmet"));
```

```
[Ahmet, Ayşe]
555-1234
```

- **List of names**: an `ArrayList` keeps elements in insertion order; `add` appends to the end.
- **Phone book**: a `HashMap` maps a key (the name) to a value (the number).
- Add with `put(key, value)`; look up the value with `get(key)`.
- Printing a list with `println` shows its elements inside `[ … ]`.

Methods of the Collection Interface

Method	Purpose	Returns
`add(E e)`	adds the element	`boolean`: `true` if added
`remove(Object o)`	removes the element	`boolean`
`size()`	gives the number of elements	`int`
`isEmpty()`	is the collection empty?	`boolean`
`contains(Object o)`	is the element in the collection?	`boolean`
`containsAll(Collection<?> c)`	are all elements of `c` in this collection?	`boolean`
`clear()`	removes all elements	—
`toArray()`	converts the collection to an array	`Object[]`
`iterator()`	gives an iterator for traversal	`Iterator<E>`
`forEach(Consumer<? super E> a)`	applies an action to each element (Java 8+)	—

`E`: the element type (a generic type parameter). `?` is a wildcard: "any type". Every `List`, `Set` and `Queue` has these methods.

Collection Methods: Example

```
Collection<String> colors = new ArrayList<>();
colors.add("red");
colors.add("green");
colors.add("blue");
System.out.println(colors.size());
System.out.println(colors.contains("green"));
colors.remove("green");
System.out.println(colors);
Collection<String> wanted = Arrays.asList("blue", "red");
System.out.println(colors.containsAll(wanted));
Object[] arr = colors.toArray();
System.out.println(arr.length);
colors.clear();
System.out.println(colors.isEmpty());
```

OUTPUT

```
3
true
[red, blue]
true
2
true
```

- The variable type is `Collection`, so only the common methods are available (no indexed `get`).
- `remove("green")` removes the element **by value**.
- `toArray()` returns a new, copied array.

The List Interface

- Elements are kept in **insertion order**: ordered, but not sorted.
- **Index**-based access: the first element is at 0, the last at `size() - 1`.
- The same element may appear more than once.
- `ArrayList`: a dynamic array with fast indexed access.
- `LinkedList`: a doubly linked list; fast insertion and removal at the ends.
- `Vector`: the old, synchronized (thread-safe) version; prefer `ArrayList`.

Method	Purpose
`add(E e)`	appends
`add(int i, E e)`	inserts at position `i`
`get(int i)`	gives the element at `i`
`set(int i, E e)`	replaces the element at `i`
`remove(int i)`	removes the element at `i`
`remove(Object o)`	removes `o`
`indexOf(Object o)`	first position; −1 if absent
`size()`	number of elements

The ArrayList Class

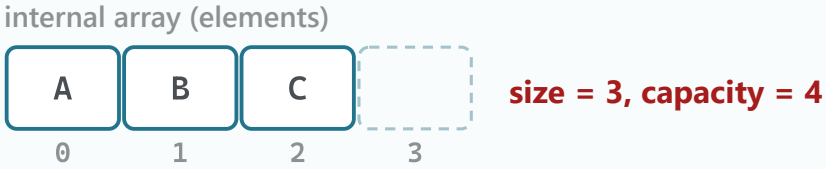
- **Dynamic array**: stores elements in an internal array, in insertion order.
- Extends `AbstractList` and implements the `List` interface.
- **Dynamic size**: the internal array grows automatically when it fills up.
- `get(i)` / `set(i, e)` are very fast: O(1).
- Appending is fast on average; inserting or removing **in the middle** shifts elements, so it is slow: O(n).

```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Pear");
System.out.println(fruits.get(1));  // by index
```

```
Pear
```

O(1): time independent of the number of elements (constant). O(n): time proportional to the number of elements *n*.

How Does an ArrayList Grow?

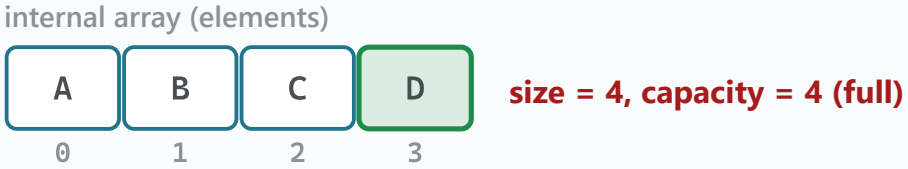


Take a list created with `new ArrayList<>(4)`, initial capacity 4: inside is a 4-cell array, 3 cells in use.

Step 1 / 4

The default capacity is 10; each growth makes it about 1.5 times larger. Since copying happens only now and then, appending is O(1) **on average** (amortized). Capacity is not `size()`: `size()` is the number of occupied cells.

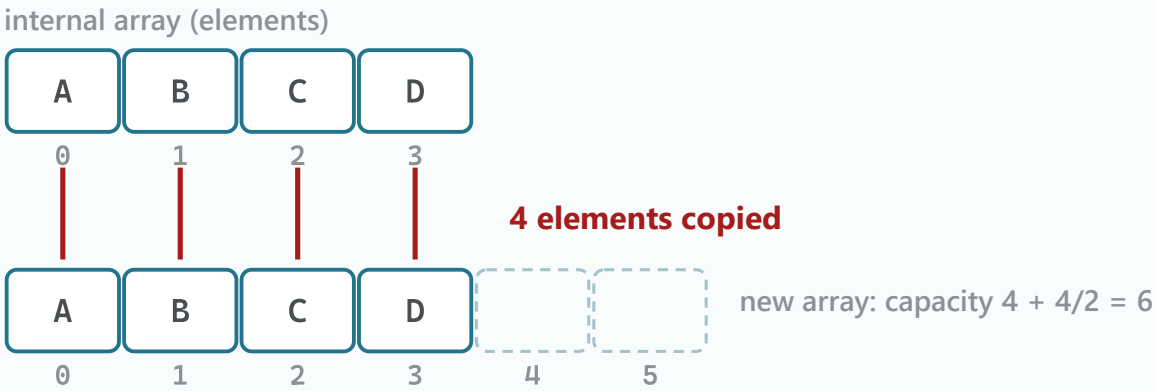
How Does an ArrayList Grow?



`add("D")` : there is a free cell; D goes straight to the end. Nothing moves: O(1).

The default capacity is 10; each growth makes it about 1.5 times larger. Since copying happens only now and then, appending is O(1) **on average** (amortized). Capacity is not `size()` : `size()` is the number of occupied cells.

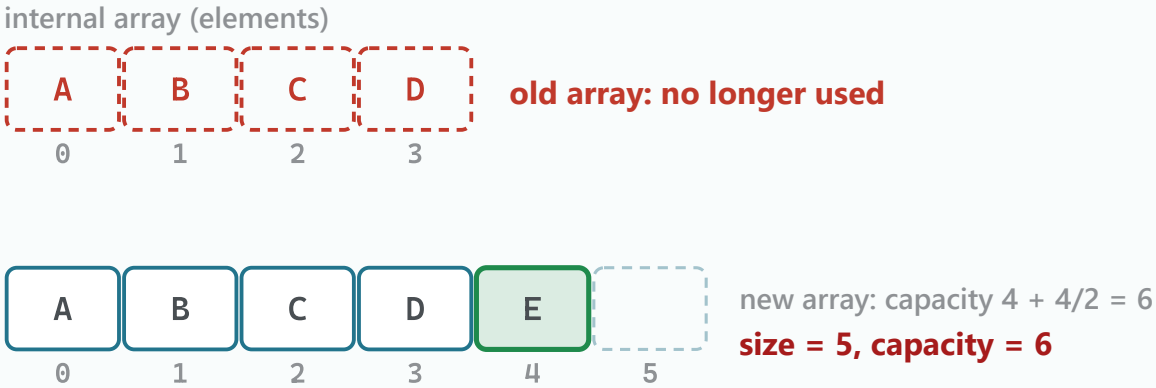
How Does an ArrayList Grow?



`add("E")` : the array is full! A larger array is allocated and the 4 old elements are copied one by one: O(n).

The default capacity is 10; each growth makes it about 1.5 times larger. Since copying happens only now and then, appending is O(1) **on average** (amortized). Capacity is not `size()` : `size()` is the number of occupied cells.

How Does an ArrayList Grow?

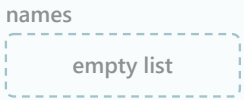


E is written at index 4 of the new array. The old array is left to the garbage collector.

The default capacity is 10; each growth makes it about 1.5 times larger. Since copying happens only now and then, appending is O(1) **on average** (amortized). Capacity is not `size()`: `size()` is the number of occupied cells.

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");       // replace index 3
9  System.out.println(names);
10 names.remove(4);           // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");     // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



size = 0

An empty `ArrayList` is created: `size()` is 0.

Step 1 / 10

VARIABLES

—

OUTPUT

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
▶  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");       // replace index 3
9  System.out.println(names);
10 names.remove(4);           // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");     // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



size = 1

`add` appends the element: Ali is at index 0.

Step 2 / 10

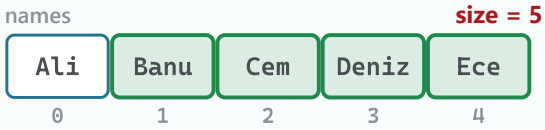
VARIABLES

—

OUTPUT

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
▶  names.add("Banu");
▶  names.add("Cem");
▶  names.add("Deniz");
▶  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");       // replace index 3
9  System.out.println(names);
10 names.remove(4);           // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");     // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



Four more elements are appended; indices 0…4.

Step 3 / 10

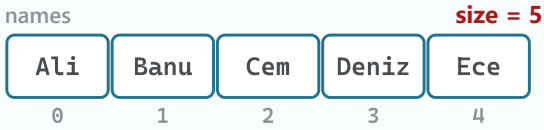
VARIABLES

—

OUTPUT

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
▶  System.out.println(names);
8  names.set(3, "Can");       // replace index 3
9  System.out.println(names);
10 names.remove(4);           // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");     // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



The list is printed: elements in insertion order.

Step 4 / 10

VARIABLES

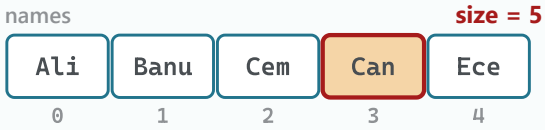
—

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
```

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
▶  names.set(3, "Can");        // replace index 3
▶  System.out.println(names);
10 names.remove(4);            // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");      // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



`set(3, "Can")` : Can replaces Deniz at index 3. The size does not change.

Step 5 / 10

VARIABLES

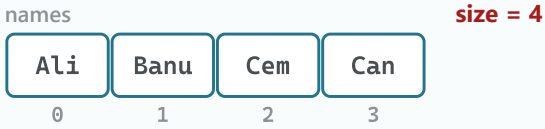
—

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
```

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");        // replace index 3
9  System.out.println(names);
▶ names.remove(4);            // remove index 4
▶ System.out.println(names);
12 names.add(2, "Fatma");      // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



`remove(4)` : Ece at index 4 is removed. It is the last element, so nothing shifts.

Step 6 / 10

VARIABLES

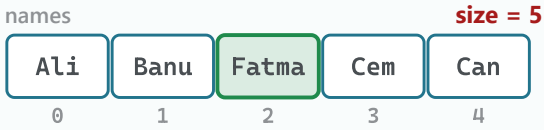
—

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
```

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");        // replace index 3
9  System.out.println(names);
10 names.remove(4);            // remove index 4
11 System.out.println(names);
▶ names.add(2, "Fatma");      // insert at index 2
▶ System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



`add(2, "Fatma")`: Cem and Can each shift one place right; Fatma goes in at index 2.

Step 7 / 10

VARIABLES

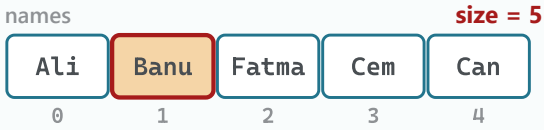
—

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
[Ali, Banu, Fatma, Cem, Can]
```

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");        // replace index 3
9  System.out.println(names);
10 names.remove(4);            // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");      // insert at index 2
13 System.out.println(names);
▶ int index = names.indexOf("Banu");
▶ System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



`indexOf("Banu")` searches from the start and returns the first position: 1.

Step 8 / 10

VARIABLES

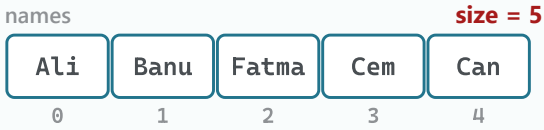
`index = 1`

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
[Ali, Banu, Fatma, Cem, Can]
index: 1
```

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");        // replace index 3
9  System.out.println(names);
10 names.remove(4);            // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");      // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
▶  boolean hasEce = names.contains("Ece");
▶  System.out.println("has Ece: " + hasEce);
18 System.out.println("size: " + names.size());
```



`contains("Ece")`: Ece was removed earlier, so the result is `false`.

Step 9 / 10

VARIABLES

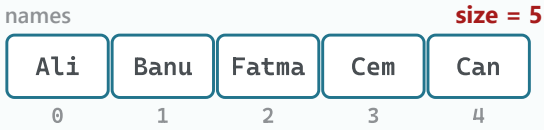
`index = 1` `hasEce = false`

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
[Ali, Banu, Fatma, Cem, Can]
index: 1
has Ece: false
```

ArrayList Methods: Trace

```
1  ArrayList<String> names = new ArrayList<>();
2  names.add("Ali");
3  names.add("Banu");
4  names.add("Cem");
5  names.add("Deniz");
6  names.add("Ece");
7  System.out.println(names);
8  names.set(3, "Can");        // replace index 3
9  System.out.println(names);
10 names.remove(4);            // remove index 4
11 System.out.println(names);
12 names.add(2, "Fatma");      // insert at index 2
13 System.out.println(names);
14 int index = names.indexOf("Banu");
15 System.out.println("index: " + index);
16 boolean hasEce = names.contains("Ece");
17 System.out.println("has Ece: " + hasEce);
▶  System.out.println("size: " + names.size());
```



Final size 5: 5 appends + 1 insertion − 1 removal.

Step 10 / 10

VARIABLES

`index = 1` `hasEce = false`

OUTPUT

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
[Ali, Banu, Fatma, Cem, Can]
index: 1
has Ece: false
size: 5
```

Inserting and Removing in the Middle

```
List<Integer> list2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
list2.add(1, 15);      // insert: 20, 30, 40 shift right
System.out.println(list2);
list2.remove(0);       // remove first: all shift left
System.out.println(list2);
```



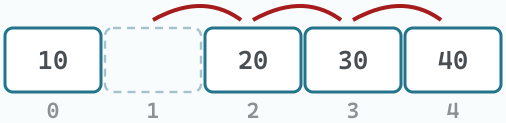
Start: `[10, 20, 30, 40]`. Now `add(1, 15)` runs.

Step 1 / 5

```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

Inserting and Removing in the Middle

```
List<Integer> list2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
list2.add(1, 15);      // insert: 20, 30, 40 shift right
System.out.println(list2);
list2.remove(0);       // remove first: all shift left
System.out.println(list2);
```



3 elements right

First the elements after index 1 shift one place right (starting from the end); cell 1 becomes free.

Step 2 / 5

```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

Inserting and Removing in the Middle

```
List<Integer> list2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
list2.add(1, 15);      // insert: 20, 30, 40 shift right
System.out.println(list2);
list2.remove(0);       // remove first: all shift left
System.out.println(list2);
```



[10, 15, 20, 30, 40]

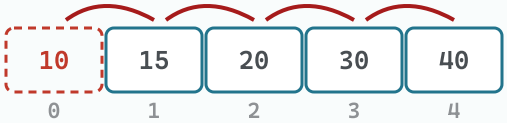
15 goes into the free cell. In a list of *n* elements the worst case is *n* shifts: O(n).

Step 3 / 5

```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

Inserting and Removing in the Middle

```
List<Integer> list2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
list2.add(1, 15);      // insert: 20, 30, 40 shift right
System.out.println(list2);
list2.remove(0);       // remove first: all shift left
System.out.println(list2);
```



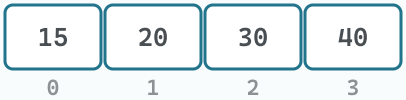
4 elements left

`remove(0)` : 10 is removed and the 4 elements behind it shift one place left.


```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

Inserting and Removing in the Middle

```
List<Integer> list2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
list2.add(1, 15);      // insert: 20, 30, 40 shift right
System.out.println(list2);
list2.remove(0);       // remove first: all shift left
System.out.println(list2);
```



[15, 20, 30, 40]

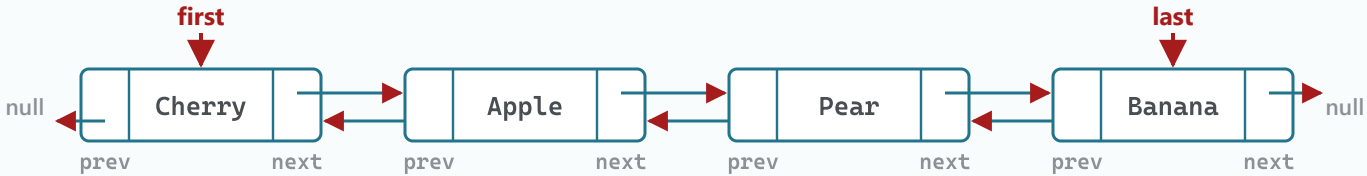
Result. Removing from the end needs no shifting: O(1). Removing from the front or middle is O(n).

Step 5 / 5

```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

The LinkedList Class

- **Doubly linked list**: elements live in **nodes**; each node holds its value and references to the previous (prev) and next (next) node.
- Extends `AbstractSequentialList`; implements the `List` and `Deque` interfaces.
- **Flexible size**: there is no capacity; each insertion creates a new node and nothing is shifted.
- Insertion and removal at the ends are O(1); but `get(i)` walks the nodes one by one: O(n).



Analogy: a line of people holding hands. If someone joins in the middle, only two people let go and regrab; but to find the 3rd person you must count from the start.

LinkedList Example

```
LinkedList<String> fruits = new LinkedList<>();
fruits.add("Apple");
fruits.add("Pear");
fruits.addFirst("Cherry");   // add to front
fruits.addLast("Banana");      // add to end
System.out.println(fruits);
System.out.println(fruits.getFirst());
System.out.println(fruits.getLast());
fruits.removeFirst();
System.out.println(fruits);
```

```
[Cherry, Apple, Pear, Banana]
Cherry
Banana
[Apple, Pear, Banana]
```

Method	Purpose
`addFirst(e)`	adds to the front
`addLast(e)`	appends
`getFirst()`	gives the first element
`getLast()`	gives the last element
`removeFirst()`	removes and returns the first element
`removeLast()`	removes and returns the last element

These methods come from the `Deque` interface; `add`, `get` and `remove(i)` come from `List`.

LinkedList: Nodes and Links

```
▶ LinkedList<String> fruits = new LinkedList<>();
2 fruits.add("Apple");
3 fruits.add("Pear");
4 fruits.addFirst("Cherry");
5 fruits.addLast("Banana");
6 System.out.println(fruits);
7 System.out.println(fruits.getFirst());
8 System.out.println(fruits.getLast());
9 fruits.removeFirst();
10 System.out.println(fruits);
```

```
first = null
last = null
```

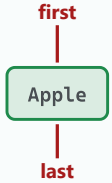
Empty list: the `first` and `last` references are `null`. There is no such thing as capacity.

Step 1 / 10

OUTPUT

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
▶  fruits.add("Apple");
3  fruits.add("Pear");
4  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
6  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```



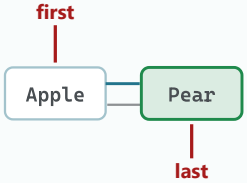
A node is created for Apple; the single node is both first and last.

Step 2 / 10

OUTPUT

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
▶  fruits.add("Pear");
4  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
6  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```



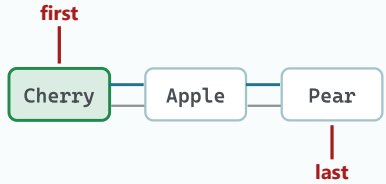
Pear is linked at the end: Apple.next → Pear, Pear.prev → Apple; `last` is updated.

Step 3 / 10

OUTPUT

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
3  fruits.add("Pear");
▶  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
6  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```



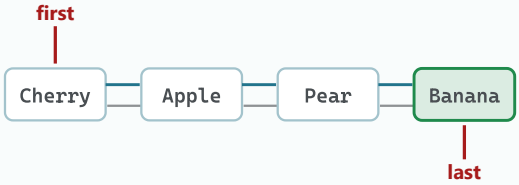
`addFirst` : Cherry goes to the front; only a few references change and nothing shifts: O(1).

Step 4 / 10

OUTPUT

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
3  fruits.add("Pear");
4  fruits.addFirst("Cherry");
▶  fruits.addLast("Banana");
6  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```

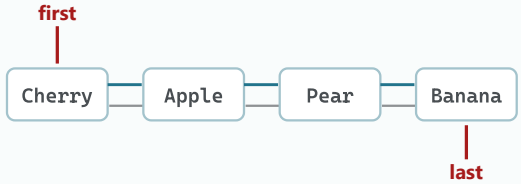


`addLast` : Banana goes to the end, again O(1).

OUTPUT

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
3  fruits.add("Pear");
4  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
▶  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```



println walks the list from front to back along the `next` links.

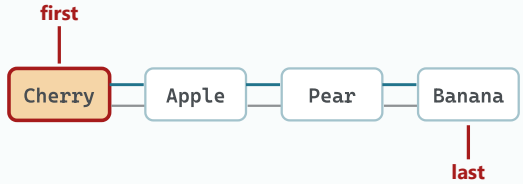
Step 6 / 10

OUTPUT

```
[Cherry, Apple, Pear, Banana]
```

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
3  fruits.add("Pear");
4  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
6  System.out.println(fruits);
▶  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```



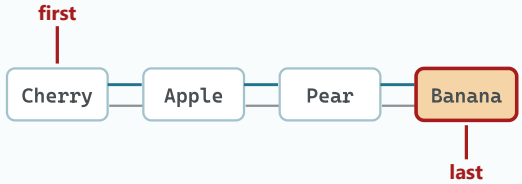
`getFirst()` : the node `first` points to, Cherry.

OUTPUT

```
[Cherry, Apple, Pear, Banana]
Cherry
```

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
3  fruits.add("Pear");
4  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
6  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
▶ 8  System.out.println(fruits.getLast());
9  fruits.removeFirst();
10 System.out.println(fruits);
```



`getLast()` : the node `last` points to, Banana. For an element in the middle (`get(i)`) you walk node by node.

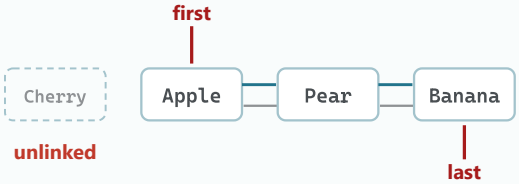
Step 8 / 10

OUTPUT

```
[Cherry, Apple, Pear, Banana]
Cherry
Banana
```

LinkedList: Nodes and Links

```
1  LinkedList<String> fruits = new LinkedList<>();
2  fruits.add("Apple");
3  fruits.add("Pear");
4  fruits.addFirst("Cherry");
5  fruits.addLast("Banana");
6  System.out.println(fruits);
7  System.out.println(fruits.getFirst());
8  System.out.println(fruits.getLast());
▶  fruits.removeFirst();
10 System.out.println(fruits);
```



`removeFirst()` : `first` moves to Apple and Apple.prev becomes null; the Cherry node drops out of the list.

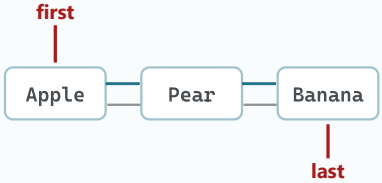
Step 9 / 10

OUTPUT

```
[Cherry, Apple, Pear, Banana]
Cherry
Banana
```

LinkedList: Nodes and Links

```
1 LinkedList<String> fruits = new LinkedList<>();
2 fruits.add("Apple");
3 fruits.add("Pear");
4 fruits.addFirst("Cherry");
5 fruits.addLast("Banana");
6 System.out.println(fruits);
7 System.out.println(fruits.getFirst());
8 System.out.println(fruits.getLast());
9 fruits.removeFirst();
▶ System.out.println(fruits);
```



`[Apple, Pear, Banana]` is printed.

Step 10 / 10

OUTPUT

```
[Cherry, Apple, Pear, Banana]
Cherry
Banana
[Apple, Pear, Banana]
```

ArrayList or LinkedList?

Operation	ArrayList	LinkedList
`get(i)`, `set(i, e)`	**O(1)**	O(n): walks node by node
Append `add(e)`	**O(1)** on average (occasional copy)	**O(1)**
Insert / remove at the front	O(n): every element shifts	**O(1)**
Insert / remove in the middle	O(n): shifting	O(n): find the position + O(1) relinking
Memory use	**low**: just the array	high: 2 extra references per node

In most cases **ArrayList** is the right choice. If insertions and removals are frequent, especially at the **ends** of the list, prefer `LinkedList` (or `ArrayDeque` for a queue or stack).

The Vector Class

- A growable dynamic array that implements the `List` interface.
- It starts with a given capacity (default 10) and **doubles** it when full.
- Its methods are synchronized: safe in multithreaded code, but every call pays for it.
- A leftover from Java 1.0; modern code prefers the lighter, faster `ArrayList`.

```
Vector<Integer> v = new Vector<>();   // capacity: 10
for (int i = 1; i <= 11; i++) {
    v.addElement(i * 10);
}
System.out.println(v.size() + " / " + v.capacity());
System.out.println(v.elementAt(0));
v.removeElement(110);    // value 110
System.out.println(v.size() + " / " + v.capacity());
```

```
11 / 20
10
10 / 20
```

Adding the 11th element raises the capacity from 10 to 20. Removal does not shrink it.

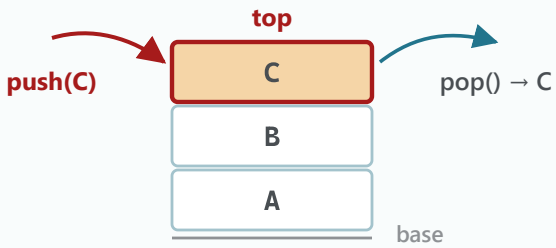
`addElement(E obj)`	`elementAt(int i)`	`removeElement(Object obj)`	`size()`	`capacity()`
adds an element	gives the element at `i`	removes the element	number of elements	current capacity

The Stack Class

- **LIFO** (Last In, First Out): the last element in is the first out.
- The newest element sits on top, the oldest at the bottom (base).
- It extends the `Vector` class.
- Uses: undo, parsing text and expressions (bracket matching), depth-first search, the method call stack.
- Modern code should use `ArrayDeque` (or `LinkedList`), which implements the `Deque` interface.

Analogy: a pile of plates; you always take from and put on the top.

Method	Purpose
`push(E item)`	pushes onto the top
`pop()`	removes and returns the top
`peek()`	returns the top without removing it
`empty()`	is the stack empty?
`search(Object o)`	1-based position from the top; −1 if absent



Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



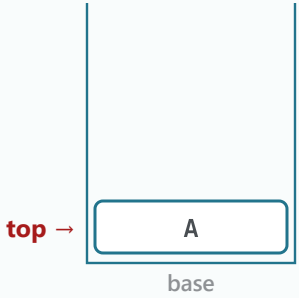
An empty stack is created.

Step 1 / 9

OUTPUT

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
▶ stack.push("A");
3 stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```

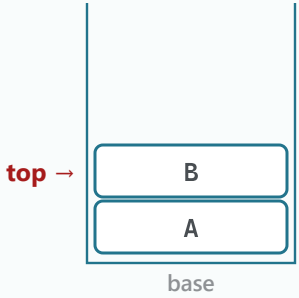


`push("A")` : A goes on top.

OUTPUT

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
▶ stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



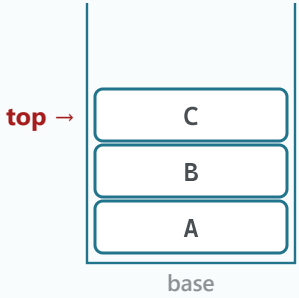
`push("B")` : B on top of A.

Step 3 / 9

OUTPUT

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
▶ stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



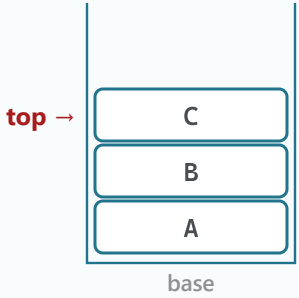
`push("C")` : now C is on top.

Step 4 / 9

OUTPUT

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
4 stack.push("C");
▶ System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



`println` prints the elements **bottom to top**: `[A, B, C]`.

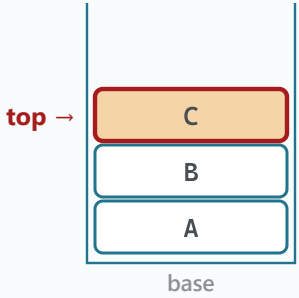
Step 5 / 9

OUTPUT

```
[A, B, C]
```

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
▶ System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



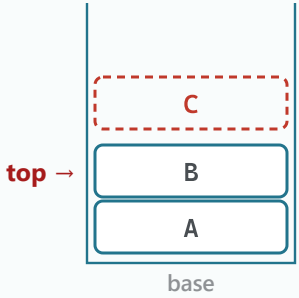
`peek()` returns the top element (C) without removing it.

OUTPUT

```
[A, B, C]
C
```

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
▶ System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



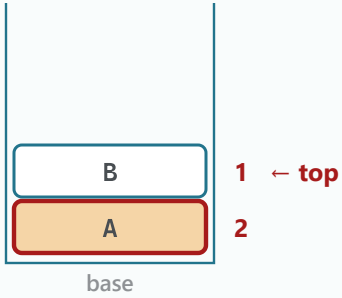
`pop()` removes and returns C: last in, first out.

OUTPUT

```
[A, B, C]
C
C
```

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
▶ System.out.println(stack.search("A"));
9 System.out.println(stack.empty());
```



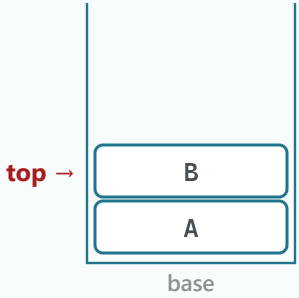
`search("A")` counts positions **from the top, starting at 1**: B is 1, A is 2.

OUTPUT

```
[A, B, C]
C
C
2
```

Stack: Trace

```
1 Stack<String> stack = new Stack<>();
2 stack.push("A");
3 stack.push("B");
4 stack.push("C");
5 System.out.println(stack);
6 System.out.println(stack.peek());
7 System.out.println(stack.pop());
8 System.out.println(stack.search("A"));
▶ System.out.println(stack.empty());
```



`empty()` : the stack holds 2 elements, so the result is `false` .

OUTPUT

```
[A, B, C]
C
C
2
false
```

Iterating Over a List

```
List<String> courses = new ArrayList<>(
        Arrays.asList("Physics", "Java", "Circuits"));
// 1) indexed for
for (int i = 0; i < courses.size(); i++) {
    System.out.print(courses.get(i) + " ");
}
System.out.println();
// 2) enhanced for (for-each)
for (String d : courses) {
    System.out.print(d + " ");
}
System.out.println();
// 3) forEach + lambda (Java 8+)
courses.forEach(d -> System.out.print(d + " "));
System.out.println();
```

OUTPUT

```
Physics Java Circuits
Physics Java Circuits
Physics Java Circuits
```

- **Indexed for**: when you need the index (lists only).
- **for-each**: the most readable way; works for every collection.
- **forEach + lambda**: concise Java 8+ form.
- If you must remove elements while iterating, use an `Iterator`.

The Queue Interface

- **FIFO** (First-In-First-Out): the first element in is the first out.
- Elements leave in the order they arrived; new elements join at the **tail**, and the next one to process is taken from the **head**.
- Elements are usually handled with `offer`, `poll` and `peek`.
- Implementations: `LinkedList` (doubly linked list), `PriorityQueue` (by priority, not FIFO), `ArrayDeque` (dynamic array).
- Uses: print queues, task queues, breadth-first search.



Analogy: a supermarket checkout. You join at the back; the person at the front pays.

Operation	Throws exception	Returns special value
Insert (at tail)	`add(e)`	`offer(e)` → `false`
Remove (from head)	`remove()`	`poll()` → `null`
Examine (head)	`element()`	`peek()` → `null`

On an empty queue `remove()` and `element()` throw `NoSuchElementException`, while `poll()` and `peek()` return `null`. `size()`: number of elements.

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



The variable has type `Queue`; the object is a `LinkedList`.

Step 1 / 11

OUTPUT

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
▶  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



`offer("Ali")` : Ali joins the queue; he is both head and tail.

Step 2 / 11

OUTPUT

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
▶  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



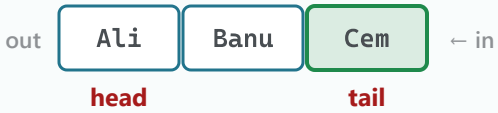
Banu joins at the tail.

Step 3 / 11

OUTPUT

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
▶  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



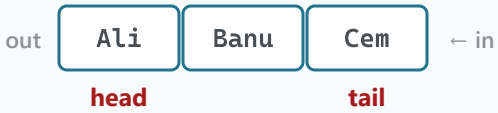
Cem joins at the tail.

Step 4 / 11

OUTPUT

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
▶  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



The queue is printed head to tail.

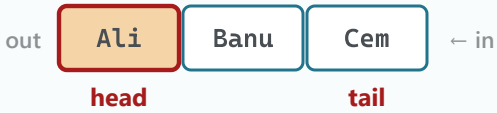
Step 5 / 11

OUTPUT

```
[Ali, Banu, Cem]
```

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
▶  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



`peek()` returns the head without removing it.

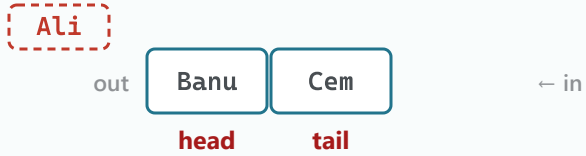
Step 6 / 11

OUTPUT

```
[Ali, Banu, Cem]
Ali
```

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
▶  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



`poll()` removes and returns the head: Ali, first in, is first out.

Step 7 / 11

OUTPUT

```
[Ali, Banu, Cem]
Ali
Ali
```

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
▶  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



Next, Banu leaves.

OUTPUT

```
[Ali, Banu, Cem]
Ali
Ali
Banu
```

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
▶  System.out.println(queue.size());
10 System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



`size()` : 1 element left in the queue.

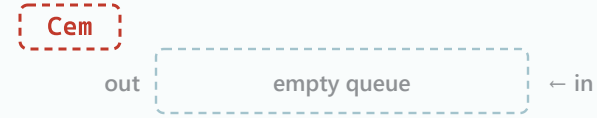
Step 9 / 11

OUTPUT

```
[Ali, Banu, Cem]
Ali
Ali
Banu
1
```

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
▶ System.out.println(queue.poll());
11 System.out.println(queue.poll());
```



Cem leaves too; the queue is empty.

Step 10 / 11

OUTPUT

```
[Ali, Banu, Cem]
Ali
Ali
Banu
1
Cem
```

Queue: Trace

```
1  Queue<String> queue = new LinkedList<>();
2  queue.offer("Ali");
3  queue.offer("Banu");
4  queue.offer("Cem");
5  System.out.println(queue);
6  System.out.println(queue.peek());
7  System.out.println(queue.poll());
8  System.out.println(queue.poll());
9  System.out.println(queue.size());
10 System.out.println(queue.poll());
▶  System.out.println(queue.poll());
```



On an empty queue `poll()` does not fail; it returns `null`.

Step 11 / 11

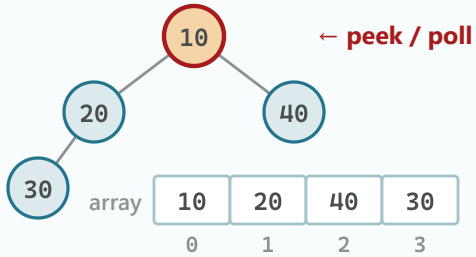
OUTPUT

```
[Ali, Banu, Cem]
Ali
Ali
Banu
1
Cem
null
```

The PriorityQueue Class

- **Priority queue**: the element that leaves is always the **highest-priority** one, not the earliest added.
- The default priority is natural order: the smallest element leaves first. Pass a comparator for the reverse.
- Elements must be comparable (`Comparable`), or a `Comparator` must be supplied.
- Internally a **binary heap**: `add` / `poll` O(log n), `peek` O(1); grows dynamically.
- Uses: task scheduling, emergency-room triage, shortest-path algorithms.

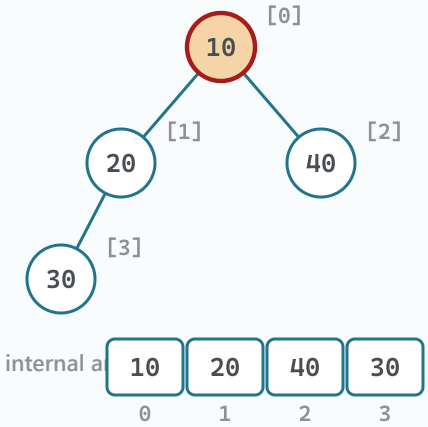
Method	Purpose
`add(E e)`	adds an element
`poll()`	removes and returns the highest-priority element
`peek()`	returns the highest-priority element without removing it
`remove(Object o)`	removes the given element
`size()`	number of elements



PriorityQueue Example

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(40);
pq.add(20);
System.out.println(pq.peek());   // smallest
System.out.println(pq);          // heap order
while (!pq.isEmpty()) {
    System.out.print(pq.poll() + " ");
}
System.out.println();
```

```
10
[10, 20, 40, 30]
10 20 30 40
```



`println(pq)` is **not** sorted: the internal array only obeys "every parent is smaller than its children". To get sorted output, remove elements one by one with `poll`.

PriorityQueue: The Heap Rule Step by Step

```
▶ PriorityQueue<Integer> pq = new PriorityQueue<>();
2 pq.add(30);
3 pq.add(10);
4 pq.add(40);
5 pq.add(20);
6 System.out.println(pq.peek());
7 System.out.println(pq);
8 while (!pq.isEmpty()) {
9     System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```

(empty)



Inside, an empty array: a **binary heap**. Rule: every parent is less than or equal to its children.

Step 1 / 12

VARIABLES

—

OUTPUT

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
▶  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```





30 goes to the root (index 0).

Step 2 / 12

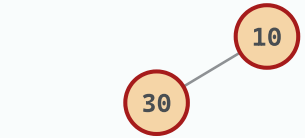
VARIABLES

—

OUTPUT

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
▶  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```





10 is appended (index 1); it is smaller than its parent 30, so it **moves up** and they swap. The root is now 10.

Step 3 / 12

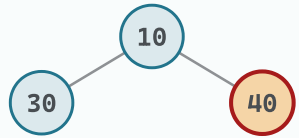
VARIABLES

—

OUTPUT

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
▶  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```





40 goes to index 2; it is larger than its parent 10 and stays.

Step 4 / 12

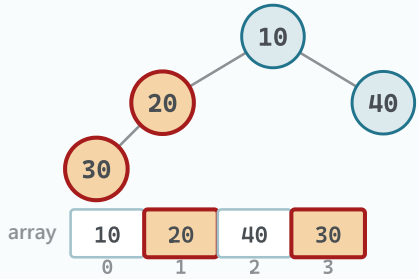
VARIABLES

—

OUTPUT

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
▶  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```



20 goes to index 3; smaller than its parent 30 → swap; larger than 10 → stop.

Step 5 / 12

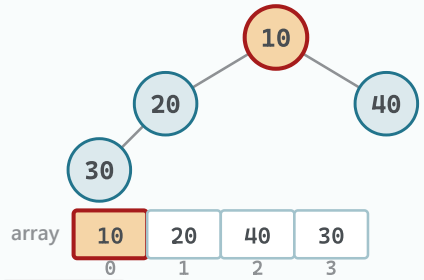
VARIABLES

—

OUTPUT

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
▶  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```



`peek()` : the root, i.e. the smallest element: 10. The queue is unchanged; O(1).

Step 6 / 12

VARIABLES

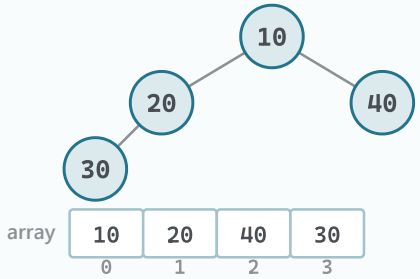
—

OUTPUT

```
10
```

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
▶  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```



println prints the internal array as is: `[10, 20, 40, 30]` is not sorted.

Step 7 / 12

VARIABLES

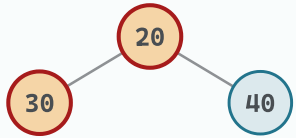
—

OUTPUT

```
10
[10, 20, 40, 30]
```

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
▶      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```





`poll()` : the root 10 leaves; the last element, 30, moves to the root and swaps with its smaller child 20, **sinking down**.

Step 8 / 12

VARIABLES

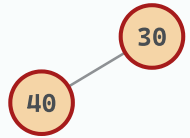
`x = 10`

OUTPUT

```
10
[10, 20, 40, 30]
10
```

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
▶      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```





out: 20

20 leaves; 40 moves to the root and swaps with 30.

Step 9 / 12

VARIABLES

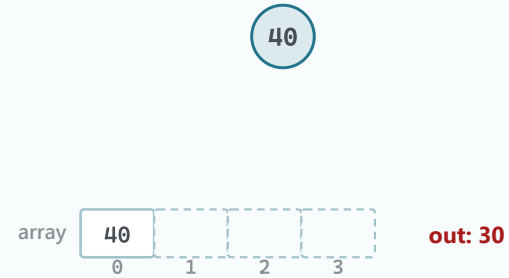
`x = 20`

OUTPUT

```
10
[10, 20, 40, 30]
10 20
```

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
▶      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```



Step 10 / 12

VARIABLES

`x = 30`

OUTPUT

```
10
[10, 20, 40, 30]
10 20 30
```

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
▶      System.out.print(pq.poll() + " ");
10 }
11 System.out.println();
```

(empty)



out: 40

40 leaves; the queue is empty and the loop ends. They came out sorted: 10 20 30 40.

Step 11 / 12

VARIABLES

`x = 40`

OUTPUT

```
10
[10, 20, 40, 30]
10 20 30 40
```

PriorityQueue: The Heap Rule Step by Step

```
1  PriorityQueue<Integer> pq = new PriorityQueue<>();
2  pq.add(30);
3  pq.add(10);
4  pq.add(40);
5  pq.add(20);
6  System.out.println(pq.peek());
7  System.out.println(pq);
8  while (!pq.isEmpty()) {
9      System.out.print(pq.poll() + " ");
10 }
▶  System.out.println();
```

(empty)



A newline is printed. Each `poll` follows only one branch: O(log n).

Step 12 / 12

VARIABLES

—

OUTPUT

```
10
[10, 20, 40, 30]
10 20 30 40
```

Setting the Priority

```
class Task implements Comparable<Task> {
    String name;
    int priority;          // smaller = more urgent

    Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    @Override
    public int compareTo(Task o) {
        return Integer.compare(priority, o.priority);
    }

    @Override
    public String toString() {
        return name + "(" + priority + ")";
    }
}
```

If `compareTo` returns a negative number, `this` comes first. `Collections.reverseOrder()` reverses natural order: the largest leaves first.

```
PriorityQueue<Task> tasks =
        new PriorityQueue<>();
tasks.add(new Task("Report", 3));
tasks.add(new Task("Outage", 1));
tasks.add(new Task("Meeting", 2));
while (!tasks.isEmpty()) {
    System.out.println(tasks.poll());
}
```

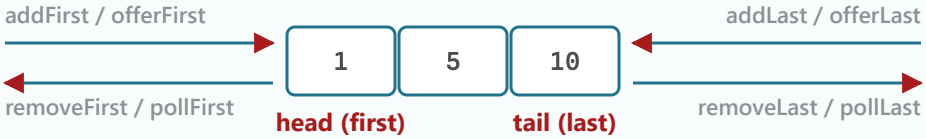
```
Outage(1)
Meeting(2)
Report(3)
```

```
PriorityQueue<Integer> maxFirst =
        new PriorityQueue<>(
        Collections.reverseOrder());
maxFirst.add(30);
maxFirst.add(10);
maxFirst.add(40);
System.out.println(maxFirst.poll());
```

```
40
```

The Deque Interface

- **Deque** (double-ended queue, pronounced “deck”): a queue where elements can be added and removed at both ends.
- It combines `Queue` (FIFO) and `Stack` (LIFO) behavior.
- Implementations: `LinkedList` (doubly linked list) and `ArrayDeque` (dynamic array).



Head	Purpose	Tail	Purpose
`addFirst(e)`	adds to the front	`addLast(e)`	appends
`removeFirst()`	removes and returns the head	`removeLast()`	removes and returns the tail
`getFirst()`	returns the head	`getLast()`	returns the tail

`remove…` / `get…` throw on an empty deque; `poll…` / `peek…` return `null`. `size()`: number of elements.

The ArrayDeque Class

- Implements a double-ended queue with a **dynamic (circular) array**.
- Supports both queue and stack operations.
- Usually faster and lighter than `LinkedList` and `Stack`; the recommended class for stacks and queues.
- Does not accept `null` elements.

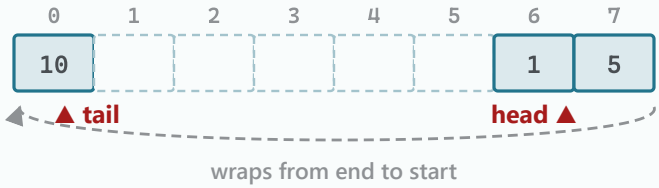
Method	Purpose
`pollFirst()` / `pollLast()`	removes and returns the head / tail
`peekFirst()` / `peekLast()`	returns the head / tail without removing it

AS A STACK (INSTEAD OF STACK)

```
Deque<String> stack = new ArrayDeque<>();
stack.push("A");     // adds at the front
stack.push("B");
System.out.println(stack.pop());
```

```
B
```

`push` / `pop` / `peek` work at the head. As a queue: `offer` adds at the tail, `poll` takes from the head.



ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



An empty `ArrayDeque`.

Step 1 / 9

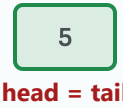
VARIABLES

—

OUTPUT

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
▶ numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



`addFirst(5)` : 5 goes to the front (the only element).

Step 2 / 9

VARIABLES

—

OUTPUT

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
▶ numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



`addLast(10)` : 10 goes to the end.

Step 3 / 9

VARIABLES

—

OUTPUT

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
▶ numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



`addFirst(1)` : 1 goes to the front; 5 and 10 stay in place.

Step 4 / 9

VARIABLES

—

OUTPUT

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
▶ System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



Printed from head to tail.

Step 5 / 9

VARIABLES

—

OUTPUT

```
[1, 5, 10]
```

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
▶ int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



`pollFirst()` removes the 1 at the head; it is unboxed into an `int`.

Step 6 / 9

VARIABLES

`firstNum = 1`

OUTPUT

```
[1, 5, 10]
```

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
▶ int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



`pollLast()` removes the 10 at the tail.

Step 7 / 9

VARIABLES

`firstNum = 1` `lastNum = 10`

OUTPUT

```
[1, 5, 10]
```

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
▶ System.out.println(firstNum + " " + lastNum);
9 System.out.println(numbers.peekFirst());
```



head = tail

Both values are printed.

Step 8 / 9

VARIABLES

`firstNum = 1` `lastNum = 10`

OUTPUT

```
[1, 5, 10]
1 10
```

ArrayDeque: Trace

```
1 ArrayDeque<Integer> numbers = new ArrayDeque<>();
2 numbers.addFirst(5);
3 numbers.addLast(10);
4 numbers.addFirst(1);
5 System.out.println(numbers);
6 int firstNum = numbers.pollFirst();
7 int lastNum = numbers.pollLast();
8 System.out.println(firstNum + " " + lastNum);
▶ System.out.println(numbers.peekFirst());
```



`peekFirst()` returns the 5 at the head without removing it.

Step 9 / 9

VARIABLES

`firstNum = 1` `lastNum = 10`

OUTPUT

```
[1, 5, 10]
1 10
5
```

The Set Interface

- Stores **unique** elements; like a set in mathematics, and suited to set operations.
- Each element appears only once: `add` does not add an existing element and returns `false`.
- In general there is no order guarantee (the order depends on the implementation).
- Insertion, removal and lookup are fast.
- `HashSet`: no order. `LinkedHashSet`: insertion order. `TreeSet`: sorted (natural order or a `Comparator`).

Method	Purpose
`add(E e)`	adds to the set
`remove(Object o)`	removes from the set
`contains(Object o)`	is it in the set?
`size()`	number of elements

```
Set<String> fruits = new HashSet<>();
System.out.println(fruits.add("Apple"));
System.out.println(fruits.add("Pear"));
System.out.println(fruits.add("Apple"));
System.out.println(fruits.size());
System.out.println(fruits.contains("Pear"));
fruits.remove("Pear");
System.out.println(fruits);
```

```
true
true
false
2
true
[Apple]
```

HashSet and Hashing

- Stores unique elements **without order**; extends `AbstractSet` and implements `Set`.
- A **bucket** number is computed from the element's `hashCode()`, and the element goes straight into that bucket: insertion and lookup are O(1) on average.
- Storage order is not guaranteed: iteration follows the buckets.



buckets: 0 … 15

Inside is a table of 16 buckets (it grows as it fills).

HashSet and Hashing

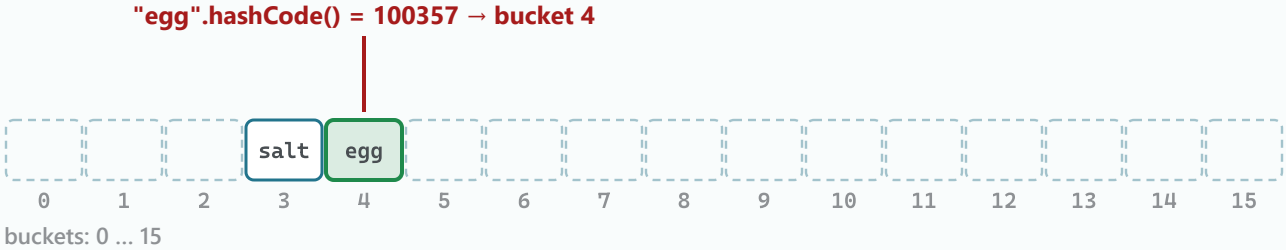
- Stores unique elements **without order**; extends `AbstractSet` and implements `Set`.
- A **bucket** number is computed from the element's `hashCode()`, and the element goes straight into that bucket: insertion and lookup are O(1) on average.
- Storage order is not guaranteed: iteration follows the buckets.



`add("salt")`: bucket 3 is computed from the hash, and salt goes there.

HashSet and Hashing

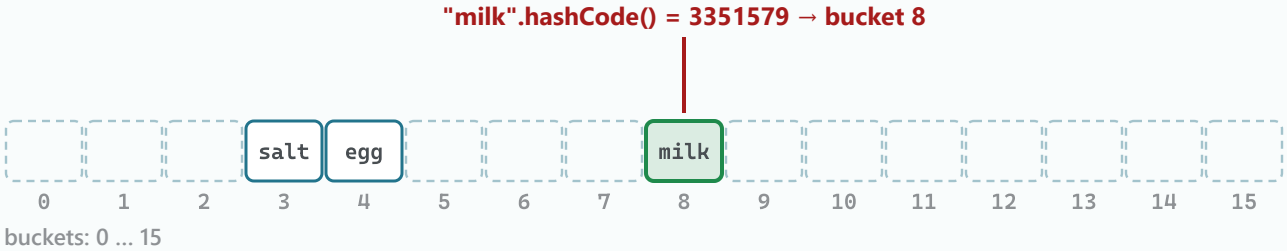
- Stores unique elements **without order**; extends `AbstractSet` and implements `Set`.
- A **bucket** number is computed from the element's `hashCode()`, and the element goes straight into that bucket: insertion and lookup are O(1) on average.
- Storage order is not guaranteed: iteration follows the buckets.



`add("egg")`: bucket 4.

HashSet and Hashing

- Stores unique elements **without order**; extends `AbstractSet` and implements `Set`.
- A **bucket** number is computed from the element's `hashCode()`, and the element goes straight into that bucket: insertion and lookup are O(1) on average.
- Storage order is not guaranteed: iteration follows the buckets.



`add("milk")`: bucket 8.

HashSet and Hashing

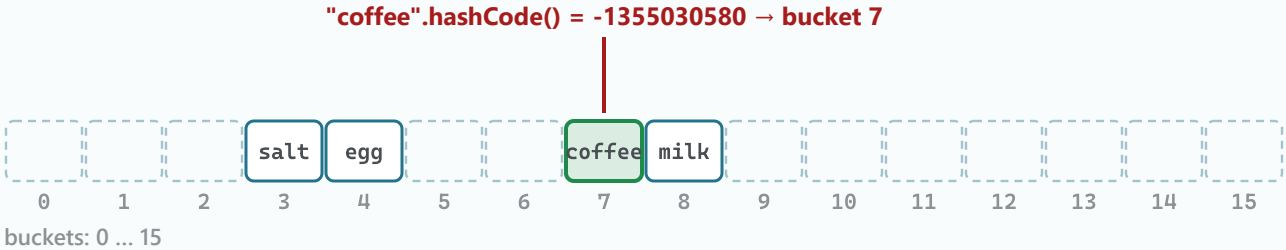
- Stores unique elements **without order**; extends `AbstractSet` and implements `Set`.
- A **bucket** number is computed from the element's `hashCode()`, and the element goes straight into that bucket: insertion and lookup are O(1) on average.
- Storage order is not guaranteed: iteration follows the buckets.



The second `add("egg")`: bucket 4 already holds an element that is `equals` to it. It is not added; `add` returns `false`.

HashSet and Hashing

- Stores unique elements **without order**; extends `AbstractSet` and implements `Set`.
- A **bucket** number is computed from the element's `hashCode()`, and the element goes straight into that bucket: insertion and lookup are O(1) on average.
- Storage order is not guaranteed: iteration follows the buckets.



`add("coffee")`: bucket 7. Iteration order follows the buckets: `[salt, egg, coffee, milk]`.

The LinkedHashSet Class

- A set of unique elements; extends `HashSet`.
- It also links the elements in a linked list: **insertion order is preserved**.
- It gives fast access and guarantees that elements are unique.
- Choose it when both performance and order matter; if order does not matter and performance is critical, use `HashSet`.

```
LinkedHashSet<Integer> numbers =
        new LinkedHashSet<>();
numbers.add(5);
numbers.add(10);
numbers.add(5);   // duplicate: not added
numbers.add(1);
System.out.println(numbers);
```

```
[5, 10, 1]
```

Method	Purpose
`add(E e)`	adds an element
`remove(Object o)`	removes the element
`contains(Object o)`	is the element present?
`size()`	number of elements
`clear()`	removes all elements

Order in Three Sets

```
String[] words = {"salt", "egg", "milk", "egg", "coffee"};
Set<String> h = new HashSet<>();
Set<String> l = new LinkedHashSet<>();
Set<String> t = new TreeSet<>();
for (String k : words) {
    h.add(k);
    l.add(k);
    t.add(k);
}
System.out.println("HashSet:       " + h);
System.out.println("LinkedHashSet: " + l);
System.out.println("TreeSet:       " + t);
```

```
HashSet:       [salt, egg, coffee, milk]
LinkedHashSet: [salt, egg, milk, coffee]
TreeSet:       [coffee, egg, milk, salt]
```

Class	Iteration order	Underlying structure	add / contains
`HashSet`	no guarantee (bucket order)	hash table	O(1) average
`LinkedHashSet`	insertion order	hash table + linked list	O(1) average
`TreeSet`	sorted (ascending)	red-black tree	O(log n)

Our Own Classes in a HashSet

```java
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point n = (Point) o;
        return x == n.x && y == n.y;
    }
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

POINTA: THE SAME CLASS WITHOUT EQUALS/HASHCODE

```java
Set<PointA> s1 = new HashSet<>();
s1.add(new PointA(1, 2));
s1.add(new PointA(1, 2));
System.out.println(s1.size());

Set<Point> s2 = new HashSet<>();
s2.add(new Point(1, 2));
s2.add(new Point(1, 2));
System.out.println(s2.size());
```

```
2
1
```

- `PointA` : the two objects count as **different** (Object's `equals` compares references) → size 2.
- `Point` : `equals` and `hashCode` are overridden → the same point appears once → size 1.

Rule: equal objects must have equal `hashCode` values: the bucket is found with `hashCode`, then compared with `equals`.

Changing an Object Inside a Set: The Lost Element

```
▶ Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```

buckets



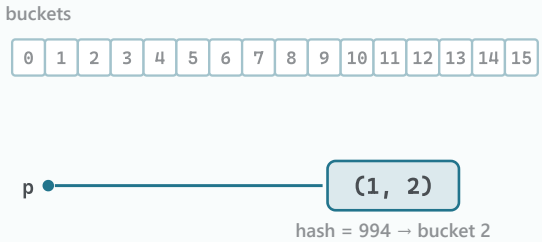
`HashSet` uses a 16-bucket hash table internally. `Point`'s `equals` and `hashCode` are computed from `x` and `y`.

Step 1 / 9

OUTPUT

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
▶ Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```



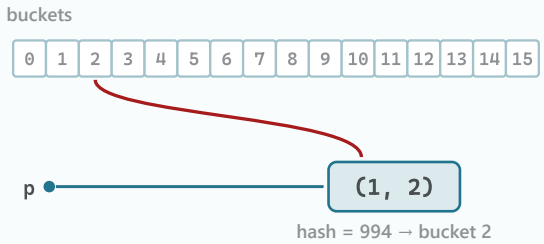
`Objects.hash(1, 2)` = 994. The bucket number is the last 4 bits of the hash: 994 → bucket 2.

Step 2 / 9

OUTPUT

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
▶ set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```



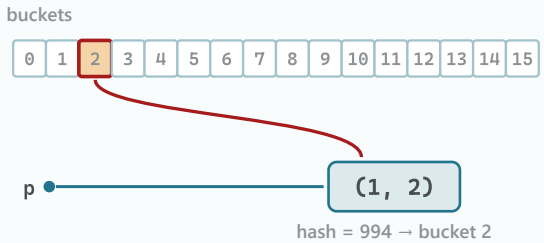
`add(p)` : the object lands in bucket 2.

Step 3 / 9

OUTPUT

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
▶ System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```



`contains(p)` : bucket 2 is checked again and `equals` is true → `true` .

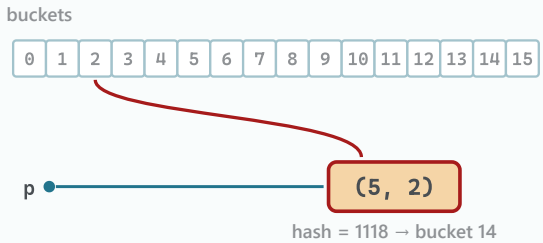
Step 4 / 9

OUTPUT

```
true
```

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
▶ p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```



A field of the object in the set has changed! Its hashCode is now 1118 → bucket 14. But the object still sits **in bucket 2**.

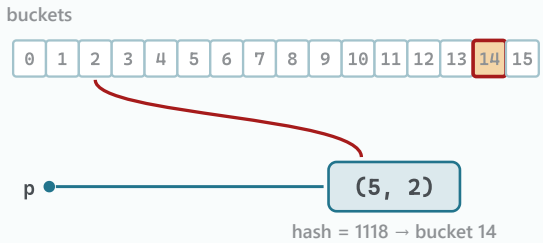
Step 5 / 9

OUTPUT

```
true
```

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
▶ System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```



`contains(p)` : bucket 14 is checked; it is empty → `false` . The object is in the set but cannot be found.

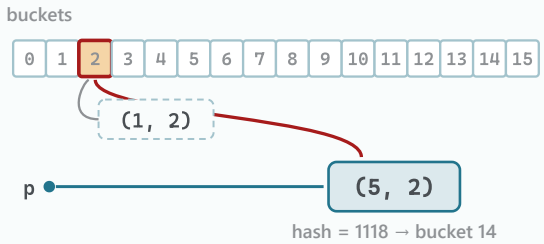
Step 6 / 9

OUTPUT

```
true
false
```

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
▶ System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
9 System.out.println(set.size());
```



The new `Point(1, 2)` goes to bucket 2; the object there is now (5, 2), so `equals` is false → `false`.

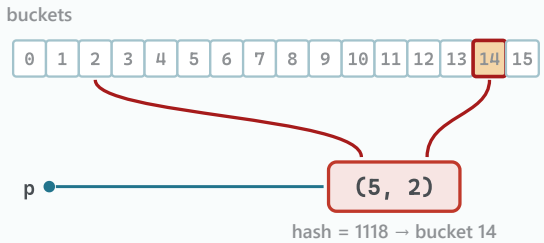
Step 7 / 9

OUTPUT

```
true
false
false
```

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
▶ set.add(p);
9 System.out.println(set.size());
```



`add(p)` : bucket 14 looks empty, so **the same object is added a second time**.

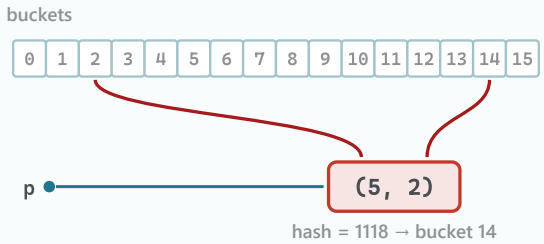
Step 8 / 9

OUTPUT

```
true
false
false
```

Changing an Object Inside a Set: The Lost Element

```
1 Set<Point> set = new HashSet<>();
2 Point p = new Point(1, 2);
3 set.add(p);
4 System.out.println(set.contains(p));
5 p.x = 5;              // field changed
6 System.out.println(set.contains(p));
7 System.out.println(set.contains(new Point(1, 2)));
8 set.add(p);
▶ System.out.println(set.size());
```



`size()` = 2: the set is corrupted. Never change the fields used by equals/hashCode of an object in a hashed collection; choose immutable keys such as `String` or `Integer`.

Step 9 / 9

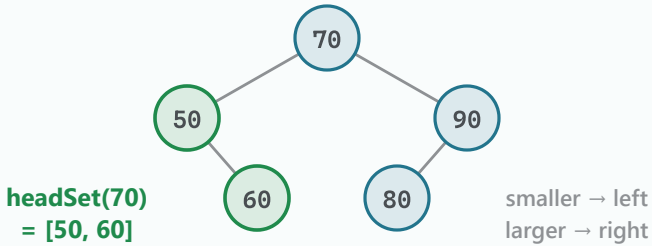
OUTPUT

```
true
false
false
2
```

SortedSet and TreeSet

- `SortedSet` : the interface of a set that keeps its unique elements **sorted**.
- Sorting follows natural order (`Comparable`) or a supplied `Comparator` .
- It offers subset and superset operations.
- `TreeSet` : extends `AbstractSet` and implements `NavigableSet` (and therefore `SortedSet`).
- Internally a red-black tree: insertion, removal and lookup are O(log n).

Method	Purpose
`first()` / `last()`	smallest / largest element
`headSet(to)`	elements less than `to` (`to` excluded)
`tailSet(from)`	`from` and greater (`from` included)
`subSet(from, to)`	`from` included, `to` excluded
`comparator()`	the sorting `Comparator` ; `null` for natural order
`add` , `remove`	insertion, removal



TreeSet Example

```
TreeSet<Integer> grades = new TreeSet<>();
grades.add(70);
grades.add(50);
grades.add(90);
grades.add(60);
grades.add(80);
System.out.println(grades);
System.out.println(grades.first() + " " + grades.last());
System.out.println(grades.headSet(70));     // 70 excluded
System.out.println(grades.tailSet(70));     // 70 included
System.out.println(grades.subSet(60, 90));  // [60, 90)
System.out.println(grades.comparator());    // natural order
```

```
TreeSet<String> fruits = new TreeSet<>();
fruits.add("Pear");
fruits.add("Apple");
System.out.println(fruits);
```

OUTPUT

```
[50, 60, 70, 80, 90]
50 90
[50, 60]
[70, 80, 90]
[60, 70, 80]
null
[Apple, Pear]
```

Elements are kept sorted whatever order they are added in. Strings are ordered alphabetically (by character code).

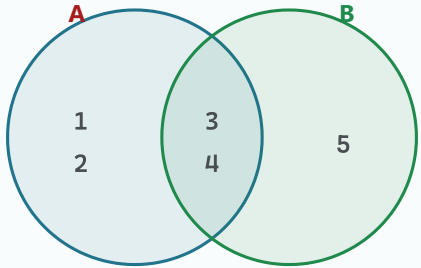
Set Operations

```
Set<Integer> a = new TreeSet<>(Arrays.asList(1, 2, 3, 4));
Set<Integer> b = new TreeSet<>(Arrays.asList(3, 4, 5));

Set<Integer> union = new TreeSet<>(a);
union.addAll(b);
Set<Integer> inter = new TreeSet<>(a);
inter.retainAll(b);
Set<Integer> diff = new TreeSet<>(a);
diff.removeAll(b);

System.out.println("A ∪ B = " + union);
System.out.println("A ∩ B = " + inter);
System.out.println("A − B = " + diff);
System.out.println(a.containsAll(inter));
```

```
A ∪ B = [1, 2, 3, 4, 5]
A ∩ B = [3, 4]
A − B = [1, 2]
true
```



A ∩ B

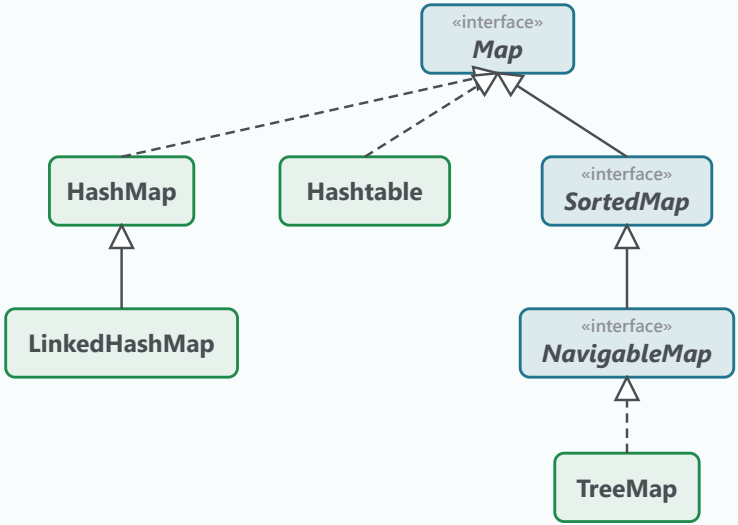
Operation	Method
union ∪	`addAll`
intersection ∩	`retainAll`
difference −	`removeAll`
subset ⊆	`containsAll`

These methods **modify** the set they are called on, so make a copy first.

The Map Interface

- Stores **key–value** pairs: each key maps to one value.
- Keys are unique; values may repeat.
- Gives fast access by key and lets you work on the pairs.
- `HashMap` : no order guarantee.
  `LinkedHashMap` : insertion order.
  `TreeMap` : sorted keys (natural order or a `Comparator`).

Analogy: a dictionary. You find the meaning (value) by the word (key); the same word does not appear twice.



Map Methods

Method	Purpose
`put(K key, V value)`	maps the key to the value; if the key exists, replaces the value and returns the old one
`get(Object key)`	returns the key's value; `null` if the key is absent
`remove(Object key)`	removes the key and its value
`containsKey(Object key)`	is the key in the map?
`containsValue(Object v)`	is the value in the map?
`keySet()`	returns all keys as a `Set`
`values()`	returns all values as a `Collection`
`entrySet()`	returns all key–value pairs as a `Set<Map.Entry<K, V>>`
`size()`, `isEmpty()`	number of pairs, is it empty?
`getOrDefault(key, v)`	returns `v` if the key is absent (Java 8+)

`K`: key type, `V`: value type. Example: `Map<String, Integer>` maps names to grades.

The HashMap Class

```
HashMap<String, Double> prices = new HashMap<>();
prices.put("Tablet", 2000.0);
prices.put("Headphones", 350.0);
prices.put("Case", 120.0);
Double old = prices.put("Tablet", 1800.0);
System.out.println(old);
System.out.println(prices.get("Tablet"));
System.out.println(prices.get("Laptop"));
System.out.println(prices.containsKey("Case"));
prices.remove("Case");
System.out.println(prices.size());
```

```
2000.0
1800.0
null
true
2
```

```
prices.put(null, 0.0);         // a single null key
prices.put("Pen", null);     // null value
System.out.println(prices.size());
```

```
4
```

- Stores key–value pairs in a **hash table**: `get` / `put` are O(1) on average.
- Extends `AbstractMap` and implements `Map`.
- **Unique keys**: `put` with an existing key replaces the value.
- **Null**: allows one `null` key and any number of `null` values.
- Elements are stored without order.

Iterating Over a Map

```
Map<String, Integer> grades = new TreeMap<>();
grades.put("Cem", 72);
grades.put("Ayşe", 95);
grades.put("Banu", 60);

for (String name : grades.keySet()) {
    System.out.print(name + " ");
}
System.out.println();
for (Integer g : grades.values()) {
    System.out.print(g + " ");
}
System.out.println();
for (Map.Entry<String, Integer> g : grades.entrySet()) {
    System.out.println(g.getKey() + " → " + g.getValue());
}
```

```
Ayşe Banu Cem
95 60 72
Ayşe → 95
Banu → 60
Cem → 72
```

- `keySet()` : keys only.
- `values()` : values only.
- `entrySet()` : each pair is a `Map.Entry`, read with `getKey()` and `getValue()`. The most efficient way when you need both key and value.
- A `Map` cannot be given directly to `for-each` : it is not `Iterable`.

Counting Words: Trace

```
String text = "egg coffee egg milk coffee egg";
Map<String, Integer> counter = new TreeMap<>();
for (String k : text.split(" ")) {
    if (counter.containsKey(k)) {
        counter.put(k, counter.get(k) + 1);
    } else {
        counter.put(k, 1);
    }
}
System.out.println(counter);
```



The text and an empty `TreeMap` are set up. `split(" ")` breaks the text into words.

Step 1 / 8

VARIABLES

—

OUTPUT

Counting Words: Trace

```
1  String text = "egg coffee egg milk coffee egg";
2  Map<String, Integer> counter = new TreeMap<>();
▶  for (String k : text.split(" ")) {
▶      if (counter.containsKey(k)) {
5          counter.put(k, counter.get(k) + 1);
6      } else {
▶          counter.put(k, 1);
8      }
9  }
10 System.out.println(counter);
```

key	value
"egg"	1

`"egg"` is seen for the first time: `put(k, 1)`.

Step 2 / 8

VARIABLES

`k = "egg"`

OUTPUT

Counting Words: Trace

```
1  String text = "egg coffee egg milk coffee egg";
2  Map<String, Integer> counter = new TreeMap<>();
▶  for (String k : text.split(" ")) {
▶      if (counter.containsKey(k)) {
5          counter.put(k, counter.get(k) + 1);
6      } else {
▶          counter.put(k, 1);
8      }
9  }
10 System.out.println(counter);
```

key	value
"coffee"	1
"egg"	1

`"coffee"` is seen for the first time: `put(k, 1)`.

Step 3 / 8

VARIABLES

`k = "coffee"`

OUTPUT

Counting Words: Trace

```
1  String text = "egg coffee egg milk coffee egg";
2  Map<String, Integer> counter = new TreeMap<>();
▶  for (String k : text.split(" ")) {
▶      if (counter.containsKey(k)) {
▶          counter.put(k, counter.get(k) + 1);
6      } else {
7          counter.put(k, 1);
8      }
9  }
10 System.out.println(counter);
```

key	value
"coffee"	1
"egg"	2

`"egg"` is already there: the old value 1 is read and 2 is stored.

Step 4 / 8

VARIABLES

`k = "egg"`

OUTPUT

Counting Words: Trace

```
1  String text = "egg coffee egg milk coffee egg";
2  Map<String, Integer> counter = new TreeMap<>();
▶  for (String k : text.split(" ")) {
▶      if (counter.containsKey(k)) {
5          counter.put(k, counter.get(k) + 1);
6      } else {
▶          counter.put(k, 1);
8      }
9  }
10 System.out.println(counter);
```

key	value
"coffee"	1
"egg"	2
"milk"	1

`"milk"` is seen for the first time: `put(k, 1)`.

Step 5 / 8

VARIABLES

`k = "milk"`

OUTPUT

Counting Words: Trace

```
1  String text = "egg coffee egg milk coffee egg";
2  Map<String, Integer> counter = new TreeMap<>();
▶  for (String k : text.split(" ")) {
▶      if (counter.containsKey(k)) {
▶          counter.put(k, counter.get(k) + 1);
6      } else {
7          counter.put(k, 1);
8      }
9  }
10 System.out.println(counter);
```

key	value
"coffee"	2
"egg"	2
"milk"	1

`"coffee"` is already there: the old value 1 is read and 2 is stored.

Step 6 / 8

VARIABLES

`k = "coffee"`

OUTPUT

Counting Words: Trace

```
1  String text = "egg coffee egg milk coffee egg";
2  Map<String, Integer> counter = new TreeMap<>();
▶  for (String k : text.split(" ")) {
▶      if (counter.containsKey(k)) {
▶          counter.put(k, counter.get(k) + 1);
6      } else {
7          counter.put(k, 1);
8      }
9  }
10 System.out.println(counter);
```

key	value
"coffee"	2
"egg"	3
"milk"	1

`"egg"` is already there: the old value 2 is read and 3 is stored.

Step 7 / 8

VARIABLES

`k = "egg"`

OUTPUT

Counting Words: Trace

```
1 String text = "egg coffee egg milk coffee egg";
2 Map<String, Integer> counter = new TreeMap<>();
3 for (String k : text.split(" ")) {
4     if (counter.containsKey(k)) {
5         counter.put(k, counter.get(k) + 1);
6     } else {
7         counter.put(k, 1);
8     }
9 }
▶ System.out.println(counter);
```

key	value
"coffee"	2
"egg"	3
"milk"	1

The loop is done. A `TreeMap` prints its keys in alphabetical order.

Step 8 / 8

VARIABLES

`k = "egg"`

OUTPUT

```
{coffee=2, egg=3, milk=1}
```

The LinkedHashMap Class

- A map of unique key–value pairs; extends `HashMap`.
- Each entry is a key and a value; keys are unique.
- It links the pairs in a linked list: **insertion order is preserved**.
- Choose it when performance and order both matter; if order does not matter, `HashMap` is enough.

```
Map<String, String> contacts = new LinkedHashMap<>();
contacts.put("Zeynep", "555-1111");
contacts.put("Ahmet", "555-1234");
contacts.put("Mert", "555-2222");
Map<String, String> h = new HashMap<>(contacts);
Map<String, String> t = new TreeMap<>(contacts);
System.out.println("LinkedHashMap: " + contacts);
System.out.println("HashMap:       " + h);
System.out.println("TreeMap:       " + t);
```

```
LinkedHashMap: {Zeynep=555-1111, Ahmet=555-1234, Mert=555-2222}
HashMap:       {Ahmet=555-1234, Mert=555-2222, Zeynep=555-1111}
TreeMap:       {Ahmet=555-1234, Mert=555-2222, Zeynep=555-1111}
```

The core methods are the same as in `Map`: `put`, `remove`, `get`, `containsKey`, `size`.

The Hashtable Class

- An old, **synchronized** (thread-safe) map class for key–value pairs (Java 1.0).
- Difference from `HashMap`: `HashMap` is **not** synchronized; every `Hashtable` method takes a lock.
- Neither keys nor values may be `null`.
- Locking makes it slower than modern alternatives; use `HashMap` in single-threaded code and `ConcurrentHashMap` in multithreaded code.
- Methods: `put`, `remove`, `get`, `containsKey`, `size`.

```
Hashtable<String, String> phoneBook = new Hashtable<>();
phoneBook.put("Ahmet", "555-1234");
System.out.println(phoneBook.get("Ahmet"));
System.out.println(phoneBook.containsKey("Ayşe"));
// phoneBook.put(null, "x");  → NullPointerException
```

```
555-1234
false
```

```
Exception in thread "main"
java.lang.NullPointerException:
Cannot invoke "Object.hashCode()"
because "key" is null
```

SortedMap and TreeMap

- `SortedMap` : the interface of a map that keeps its keys **sorted**; each key appears once.
- It offers submap operations such as head and tail maps.
- `TreeMap` : extends `AbstractMap` and implements `NavigableMap` .
- Keys are ordered by natural order or a supplied `Comparator` .
- Internally a **red-black tree**: `get` / `put` O(log n).

Method	Purpose
`firstKey()` / `lastKey()`	smallest / largest key
`headMap(to)`	keys less than `to` (excluded)
`tailMap(from)`	`from` and greater (included)
`subMap(from, to)`	`from` included, `to` excluded
`comparator()`	the sorting `Comparator` (`null` for natural order)
`put` , `get` , `remove`	same as `Map`

TreeMap Example

```
TreeMap<String, Double> prices = new TreeMap<>();
prices.put("Tablet", 2000.0);
prices.put("Pen", 15.0);
prices.put("Notebook", 40.0);
prices.put("Ruler", 5.0);
System.out.println(prices);
System.out.println(prices.firstKey());
System.out.println(prices.lastKey());
System.out.println(prices.headMap("Pen"));
System.out.println(prices.tailMap("Pen"));
System.out.println(prices.subMap("Notebook", "Ruler"));
```

OUTPUT

```
{Notebook=40.0, Pen=15.0, Ruler=5.0, Tablet=2000.0}
Notebook
Tablet
{Notebook=40.0}
{Pen=15.0, Ruler=5.0, Tablet=2000.0}
{Notebook=40.0, Pen=15.0}
```

Comparing Map Classes

	HashMap	LinkedHashMap	TreeMap	Hashtable
Iteration order	no guarantee	insertion order	sorted keys	no guarantee
`get` / `put`	O(1) average	O(1) average	O(log n)	O(1) average, locked
`null` key	one	one	no	no
`null` value	yes	yes	yes	no
Synchronized	no	no	no	yes (legacy)
When?	the default choice	order must be kept	sorted keys, range queries	legacy code

The same logic applies to sets: `HashSet`, `LinkedHashSet` and `TreeSet` use a `HashMap`, `LinkedHashMap` and `TreeMap` internally.

Iterator

- **Iterator**: an object for visiting the elements of a collection **one by one**.
- It reaches every element without knowing the collection's internal structure (array, tree, hash table...).
- The safe way to remove while iterating is the iterator's `remove()` (or `removeIf` for bulk removal).
- A `for-each` loop uses an `Iterator` behind the scenes.
- For lists, `ListIterator` can also move backwards and supports `add` and `set`.

Method	Purpose
`hasNext()`	is there a next element?
`next()`	returns the next element and advances the cursor
`remove()`	removes the element returned by the last `next()`

Usually used in a `while (it.hasNext())` loop (or a `for`).



Iterator: Trace

```
ArrayList<Integer> list = new ArrayList<>();
list.add(3);
list.add(8);
list.add(5);
Iterator<Integer> iterator = list.iterator();
while (iterator.hasNext()) {
    Integer num = iterator.next();
    // process the element here
    System.out.println(num);
}
```



The list is set up: `[3, 8, 5]`.

Step 1 / 9

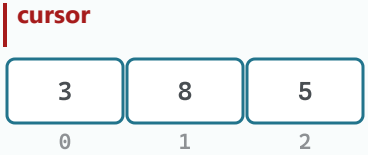
VARIABLES

—

OUTPUT

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
▶  Iterator<Integer> iterator = list.iterator();
6  while (iterator.hasNext()) {
7      Integer num = iterator.next();
8      // process the element here
9      System.out.println(num);
10 }
```



`iterator()` : the cursor is placed **before** the first element.

Step 2 / 9

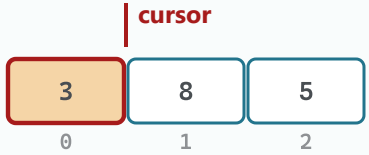
VARIABLES

—

OUTPUT

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
6  while (iterator.hasNext()) {
7      Integer num = iterator.next();
8      // process the element here
9      System.out.println(num);
10 }
```



`hasNext()` is `true`; `next()` returns 3 and advances the cursor.

Step 3 / 9

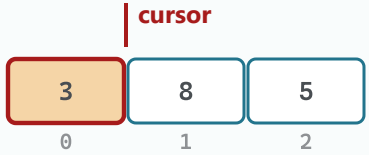
VARIABLES

`num = 3`

OUTPUT

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
6  while (iterator.hasNext()) {
7      Integer num = iterator.next();
8      // process the element here
▶      System.out.println(num);
10 }
```



The element is printed.

Step 4 / 9

VARIABLES

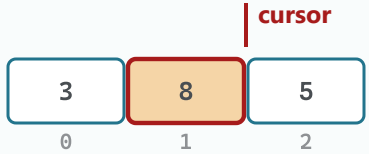
`num = 3`

OUTPUT

```
3
```

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
▶  while (iterator.hasNext()) {
▶      Integer num = iterator.next();
8      // process the element here
9      System.out.println(num);
10 }
```



`hasNext()` is `true`; `next()` returns 8 and advances the cursor.

Step 5 / 9

VARIABLES

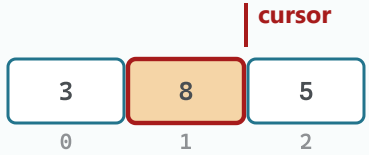
`num = 8`

OUTPUT

```
3
```

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
6  while (iterator.hasNext()) {
7      Integer num = iterator.next();
8      // process the element here
▶      System.out.println(num);
10 }
```



The element is printed.

Step 6 / 9

VARIABLES

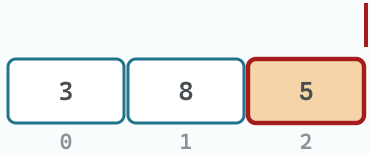
`num = 8`

OUTPUT

```
3
8
```

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
▶  while (iterator.hasNext()) {
▶      Integer num = iterator.next();
8      // process the element here
9      System.out.println(num);
10 }
```



`hasNext()` is `true`; `next()` returns 5 and advances the cursor.

Step 7 / 9

VARIABLES

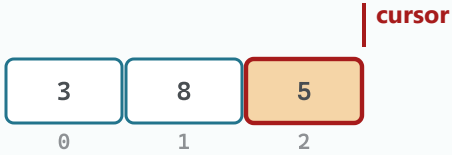
`num = 5`

OUTPUT

```
3
8
```

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
6  while (iterator.hasNext()) {
7      Integer num = iterator.next();
8      // process the element here
▶      System.out.println(num);
10 }
```



The element is printed.

Step 8 / 9

VARIABLES

`num = 5`

OUTPUT

```
3
8
5
```

Iterator: Trace

```
1  ArrayList<Integer> list = new ArrayList<>();
2  list.add(3);
3  list.add(8);
4  list.add(5);
5  Iterator<Integer> iterator = list.iterator();
▶ while (iterator.hasNext()) {
7      Integer num = iterator.next();
8      // process the element here
9      System.out.println(num);
10 }
```



The cursor is at the end: `hasNext()` is `false` and the loop ends.

Step 9 / 9

VARIABLES

`num = 5`

OUTPUT

```
3
8
5
```

Removing While Iterating

WRONG: REMOVING FROM THE COLLECTION INSIDE FOR-EACH

```
List<Integer> numbers = new ArrayList<>(Arrays.asList(4, 7, 10, 13, 16));
for (Integer s : numbers) {
    if (s % 2 == 0) {
        numbers.remove(s);
    }
}
```

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

RIGHT: ITERATOR.REMOVE()

```
List<Integer> numbers = new ArrayList<>();
Collections.addAll(numbers, 4, 7, 10, 13, 16);
Iterator<Integer> it = numbers.iterator();
while (it.hasNext()) {
    if (it.next() % 2 == 0) {
        it.remove();          // safe removal
    }
}
System.out.println(numbers);
```

SHORTCUT (JAVA 8+)

```
List<Integer> s2 = new ArrayList<>();
Collections.addAll(s2, 4, 7, 10, 13, 16);
s2.removeIf(s -> s % 2 == 0);
System.out.println(s2);
```

```
[7, 13]
```

Both ways print the same: only the odd numbers remain.

Removing Inside for-each: Why the Exception?

```
▶ List<Integer> s = new ArrayList<>();
▶ Collections.addAll(s, 6, 9, 12);
3 for (Integer x : s) {      // hidden Iterator
4     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



The list has 3 elements; `modCount` (the structural modification counter) is 0.

Step 1 / 8

VARIABLES

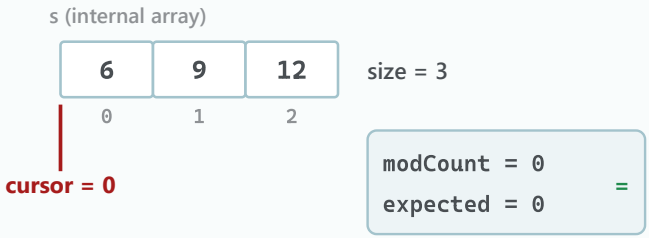
`s = [6, 9, 12]` `modCount = 0`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
▶ for (Integer x : s) {       // hidden Iterator
4     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



for-each secretly calls `s.iterator()` : cursor = 0, and the iterator saves the current modCount: expectedModCount = 0.

Step 2 / 8

VARIABLES

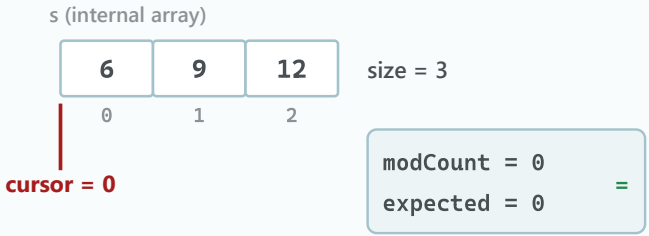
`s = [6, 9, 12]` `modCount = 0`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
▶ for (Integer x : s) {       // hidden Iterator
4     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



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

Step 3 / 8

VARIABLES

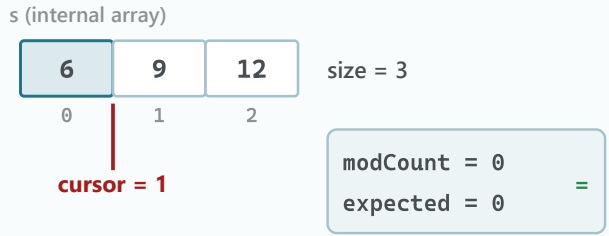
`s = [6, 9, 12]` `modCount = 0` `hasNext = true`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
▶ for (Integer x : s) {       // hidden Iterator
4     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



`next()` : first the counters are compared (0 = 0); x = 6, cursor = 1.

Step 4 / 8

VARIABLES

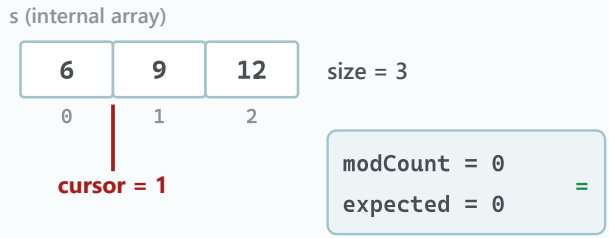
`s = [6, 9, 12]` `modCount = 0` `x = 6`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
3 for (Integer x : s) {       // hidden Iterator
▶     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



6 is even.

Step 5 / 8

VARIABLES

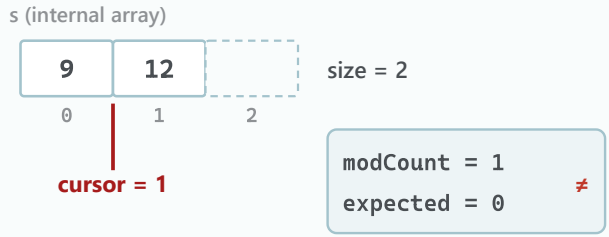
`s = [6, 9, 12]` `modCount = 0` `x = 6`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
3 for (Integer x : s) {      // hidden Iterator
4     if (x % 2 == 0) {
▶         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



`s.remove(x)` is the list's own method: 9 and 12 shift left, size becomes 2, **modCount becomes 1**. The iterator does not know.

Step 6 / 8

VARIABLES

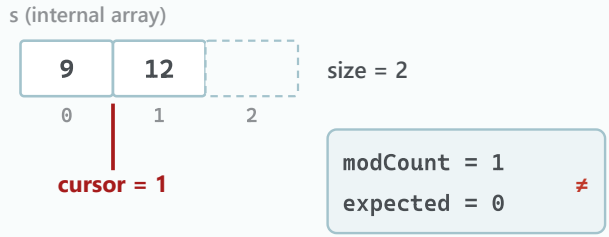
`s = [9, 12]` `modCount = 1` `x = 6`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
▶ for (Integer x : s) {       // hidden Iterator
4     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



`hasNext()` : cursor (1) ≠ size (2) → true; the loop goes on.

Step 7 / 8

VARIABLES

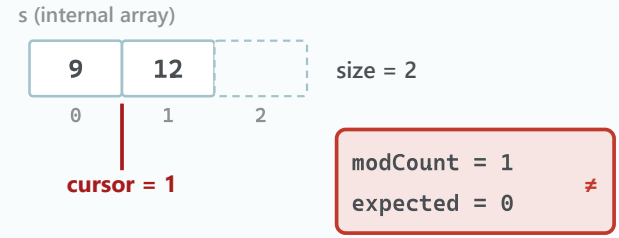
`s = [9, 12]` `modCount = 1` `hasNext = true`

OUTPUT

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

Removing Inside for-each: Why the Exception?

```
1 List<Integer> s = new ArrayList<>();
2 Collections.addAll(s, 6, 9, 12);
▶ for (Integer x : s) {       // hidden Iterator
4     if (x % 2 == 0) {
5         s.remove(x);
6     }
7 }
8 System.out.println(s);
```



ConcurrentModificationException

`next()` : modCount (1) ≠ expectedModCount (0) → `ConcurrentModificationException`. With {6, 9}, hasNext would be false: no exception, and 9 would go unchecked.

Step 8 / 8

VARIABLES

`s = [9, 12]` `modCount = 1`

OUTPUT

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

The fail-fast check is best-effort, not a guarantee: never rely on the exception.

The Collections Class and Algorithms

- `Collection` is an **interface**; `Collections` (with an **s**) is a **utility class** of ready-made `static` methods that work on collections.
- These algorithms are fast and well tested; use them instead of hand-written loops.
- Choosing the right algorithm has a big impact on performance.

Sorting

`sort`, `reverse`, `reverseOrder`

Shuffling

`shuffle`

Searching

`binarySearch`, `max`, `min`, `frequency`

Composition

`copy`, `fill`, `swap`, `reverse`

Sorting Methods

```
List<String> names = new ArrayList<>(
        Arrays.asList("Cem", "Ayşe", "Banu", "Ali"));
Collections.sort(names);
System.out.println(names);
Collections.sort(names, Collections.reverseOrder());
System.out.println(names);
Collections.reverse(names);
System.out.println(names);
```

```
[Ali, Ayşe, Banu, Cem]
[Cem, Banu, Ayşe, Ali]
[Ali, Ayşe, Banu, Cem]
```

Method	Purpose
`sort(list)`	by natural order; elements must be `Comparable`
`sort(list, c)`	by the given `Comparator`
`reverse(list)`	reverses the order of the list

Classes such as `String`, `Integer` and `Double` are already `Comparable`. Java 8+: `list.sort(null)` also sorts by natural order.

`Collections.sort`: **TimSort**, O(n log n), **stable** (equal elements keep their order); `Arrays.sort` on primitives uses Dual-Pivot Quicksort.

Comparable: Natural Order

```java
class Student implements Comparable<Student> {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    // ascending by score
    @Override
    public int compareTo(Student other) {
        return Integer.compare(score, other.score);
    }

    @Override
    public String toString() {
        return name + ":" + score;
    }
}
```

```java
List<Student> roster = new ArrayList<>();
roster.add(new Student("Cem", 72));
roster.add(new Student("Ayşe", 95));
roster.add(new Student("Banu", 60));

Collections.sort(roster);
System.out.println(roster);
```

```
[Banu:60, Cem:72, Ayşe:95]
```

- The class implements `Comparable<Student>` and writes `compareTo`.
- Result: negative → `this` first; 0 → equal; positive → `other` first.
- `Integer.compare` is safer than subtraction (`score - other.score`): no overflow.

Comparator: Another Order

```
Collections.sort(roster, new Comparator<Student>() {
    @Override
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);   // by name
    }
});
System.out.println(roster);

// Java 8+: the same, shorter
roster.sort(Comparator.comparing(o -> o.score));
System.out.println(roster);
```

```
[Ayşe:95, Banu:60, Cem:72]
[Banu:60, Cem:72, Ayşe:95]
```

- `Comparator` : a separate ordering written outside the class; a class can have several.
- Classic form: an anonymous inner class with `compare(a, b)` .
- Java 8+: one line with `Comparator.comparing(…)` and a lambda; `.reversed()` for reverse order.
- The natural order (`compareTo`) does not change; another key is used for that sort only.

ALGORITHMS

Shuffling

- Used to put the elements of a collection in **random** order.
- `Collections.shuffle(list)` uses the **Fisher–Yates** (Knuth) algorithm: starting from the end of the list, it swaps each element with a randomly chosen one.
- Fast: *n* − 1 swaps, O(n). Every possible ordering is **equally likely** (a uniform distribution).
- Uses: dealing cards, drawing lots, randomizing question order.

```
List<String> students = new ArrayList<>(
        Arrays.asList("Ali", "Banu", "Cem", "Deniz"));
Collections.shuffle(students);
System.out.println(students);
```

OUTPUT OF ONE RUN

```
[Cem, Ali, Deniz, Banu]
```

The output differs on every run. For a reproducible shuffle (in tests), pass a seeded `new Random(42)` as the second argument.

Lottery Example

```java
public class Lottery {
    // draws lottery numbers
    public static Integer[] draw(int total, int count) {
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        for (int i = 0; i < total; i++) {
            numbers.add(i + 1);          // 1 … total
        }
        Collections.shuffle(numbers);    // shuffle
        Integer[] drawn = new Integer[count];
        for (int i = 0; i < count; i++) {
            drawn[i] = numbers.get(i);
        }
        return drawn;
    }

    public static void main(String[] args) {
        Integer[] result = draw(49, 6);   // 6 out of 49
        for (Integer item : result) {
            System.out.print(item + " ");
        }
    }
}
```

SAMPLE OUTPUT

```
46 18 22 11 20 10
```

The list 1–49 is shuffled and the first 6 elements are taken: the numbers **never repeat**. The output differs on every run.

ALGORITHMS

Searching Algorithms

Linear search

Checks the elements one by one from start to end. No sorting required; good for small collections. O(n). Examples: `indexOf`, `contains`.

Binary search

Checks the middle of a sorted collection, then goes left or right; each step halves the range. O(log n). **The collection must be sorted.**

```
List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 9, 11);
System.out.println(Collections.binarySearch(numbers, 7));
System.out.println(Collections.binarySearch(numbers, 6));
System.out.println(numbers.indexOf(7));    // linear search
```

```
3
-4
3
```

7 is found at index 3 in three comparisons. 6 is absent; its insertion point is 3, so the result is −(3) − 1 = −4.

If absent, the result is **negative**: −(insertion point) − 1; on an unsorted list it is **undefined**. For a million elements: linear search up to a million comparisons, binary at most 20.

Composition Algorithms

Method	Purpose
`copy(target, source)`	copies the source elements to the start of the target; requires `target.size() ≥ source.size()`
`fill(list, d)`	sets every element to `d`
`reverse(list)`	reverses the order of the elements
`swap(list, i, j)`	swaps positions `i` and `j`
`max` / `min`	largest / smallest element
`frequency(c, o)`	how many times `o` occurs

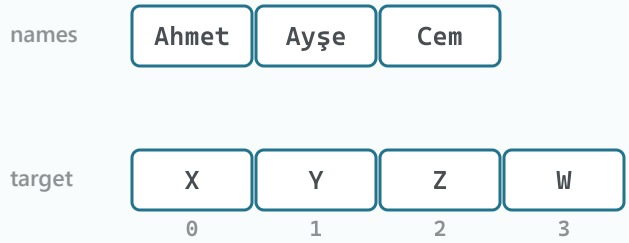
```
List<Integer> n = Arrays.asList(4, 9, 2, 9, 5);
System.out.println(Collections.max(n));
System.out.println(Collections.min(n));
System.out.println(Collections.frequency(n, 9));
```

```
9
2
2
```

If the target is too short, `copy` throws `IndexOutOfBoundsException: Source does not fit in dest`.

Composition: Trace

```
List<String> names =
        Arrays.asList("Ahmet", "Ayşe", "Cem");
List<String> target = new ArrayList<>(
        Arrays.asList("X", "Y", "Z", "W"));
Collections.copy(target, names);
System.out.println(target);
Collections.reverse(target);
System.out.println(target);
Collections.swap(target, 0, 2);
System.out.println(target);
Collections.fill(target, "Java");
System.out.println(target);
```



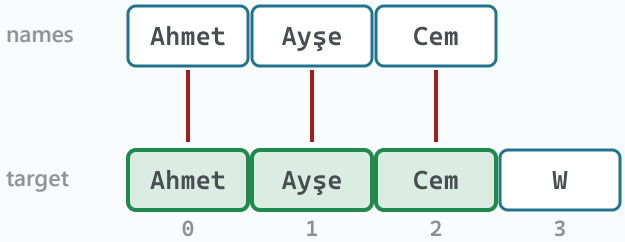
`Arrays.asList` is a fixed-size list; `target` is a 4-element `ArrayList`.

Step 1 / 5

OUTPUT

Composition: Trace

```
1  List<String> names =
2          Arrays.asList("Ahmet", "Ayşe", "Cem");
3  List<String> target = new ArrayList<>(
4          Arrays.asList("X", "Y", "Z", "W"));
▶  Collections.copy(target, names);
▶  System.out.println(target);
7  Collections.reverse(target);
8  System.out.println(target);
9  Collections.swap(target, 0, 2);
10 System.out.println(target);
11 Collections.fill(target, "Java");
12 System.out.println(target);
```



`copy` : the 3 elements are written to positions 0–2 of the target; W stays.

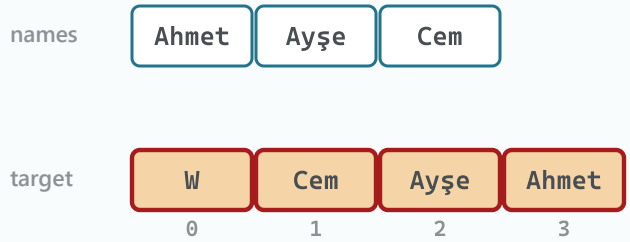
Step 2 / 5

OUTPUT

```
[Ahmet, Ayşe, Cem, W]
```

Composition: Trace

```
1  List<String> names =
2          Arrays.asList("Ahmet", "Ayşe", "Cem");
3  List<String> target = new ArrayList<>(
4          Arrays.asList("X", "Y", "Z", "W"));
5  Collections.copy(target, names);
6  System.out.println(target);
▶  Collections.reverse(target);
▶  System.out.println(target);
9  Collections.swap(target, 0, 2);
10 System.out.println(target);
11 Collections.fill(target, "Java");
12 System.out.println(target);
```



`reverse` : the order flips.

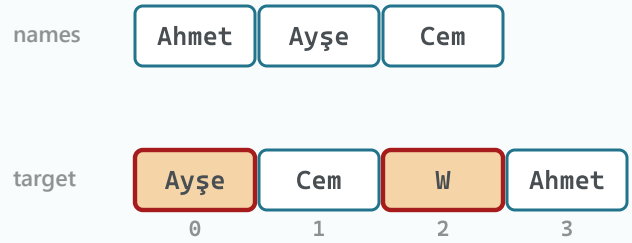
Step 3 / 5

OUTPUT

```
[Ahmet, Ayşe, Cem, W]
[W, Cem, Ayşe, Ahmet]
```

Composition: Trace

```
1  List<String> names =
2          Arrays.asList("Ahmet", "Ayşe", "Cem");
3  List<String> target = new ArrayList<>(
4          Arrays.asList("X", "Y", "Z", "W"));
5  Collections.copy(target, names);
6  System.out.println(target);
7  Collections.reverse(target);
8  System.out.println(target);
▶  Collections.swap(target, 0, 2);
▶  System.out.println(target);
11 Collections.fill(target, "Java");
12 System.out.println(target);
```



`swap(target, 0, 2)` : positions 0 and 2 trade places.

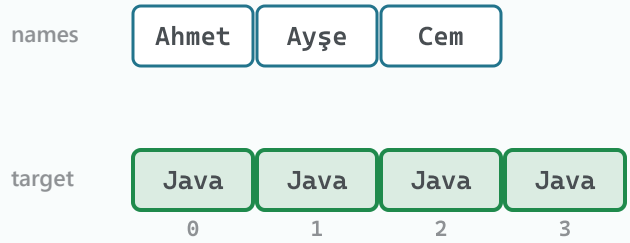
Step 4 / 5

OUTPUT

```
[Ahmet, Ayşe, Cem, W]
[W, Cem, Ayşe, Ahmet]
[Ayşe, Cem, W, Ahmet]
```

Composition: Trace

```
1  List<String> names =
2          Arrays.asList("Ahmet", "Ayşe", "Cem");
3  List<String> target = new ArrayList<>(
4          Arrays.asList("X", "Y", "Z", "W"));
5  Collections.copy(target, names);
6  System.out.println(target);
7  Collections.reverse(target);
8  System.out.println(target);
9  Collections.swap(target, 0, 2);
10 System.out.println(target);
▶  Collections.fill(target, "Java");
▶  System.out.println(target);
```



`fill` : every element becomes `"Java"` . The size does not change.

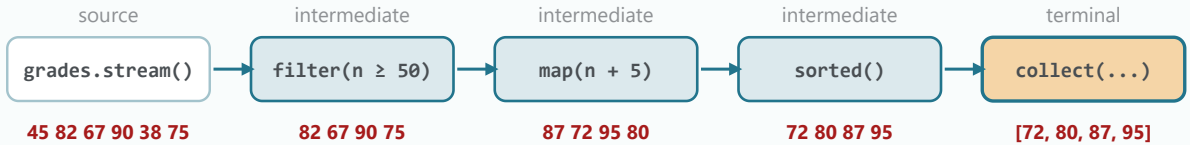
Step 5 / 5

OUTPUT

```
[Ahmet, Ayşe, Cem, W]
[W, Cem, Ayşe, Ahmet]
[Ayşe, Cem, W, Ahmet]
[Java, Java, Java, Java]
```

The Stream API (`java.util.stream`)

- `java.util.stream`, new in Java 8, processes a sequence of data as a **pipeline**: we say "what to do", and the library builds the loop.
- A stream here is **not** an I/O stream like `InputStream`: it stores no data, passes over a source (a list, an array, lines of a file) and does not modify the source.



intermediate operations return a new stream; nothing is computed until the terminal operation is called

```
List<Integer> grades = Arrays.asList(45, 82, 67, 90, 38, 75);
List<Integer> passed = grades.stream()
    .filter(n -> n >= 50)
    .map(n -> n + 5)
    .sorted()
    .collect(Collectors.toList());
System.out.println(passed);
```

OUTPUT

```
[72, 80, 87, 95]
```

From Java 16 on, the last line can be shortened to `.toList()`. Imports needed: `java.util.*`, `java.util.stream.Collectors`.

Intermediate and Terminal Operations

Intermediate	Purpose
`filter(p)`	passes elements that satisfy the condition
`map(f)`	transforms each element
`flatMap(f)`	flattens nested streams
`sorted()`	sorts (waits for all elements)
`distinct()`	drops duplicates (`equals`)
`limit(n)` / `skip(n)`	takes / skips the first *n* elements
`peek(f)`	observes elements (debugging)

Terminal	Result
`collect(...)`	list, set, map…
`forEach(f)`	an action for each element
`count()`	number of elements (`long`)
`reduce(...)`	reduces the elements to one value
`findFirst()`	first element (`Optional`)
`anyMatch` / `allMatch` / `noneMatch`	`boolean`
`min` / `max`	`Optional`

An intermediate operation returns **a new stream** and the chain goes on; a terminal operation returns a non-stream result (or nothing) and **runs** the pipeline. A pipeline without a terminal operation does nothing.

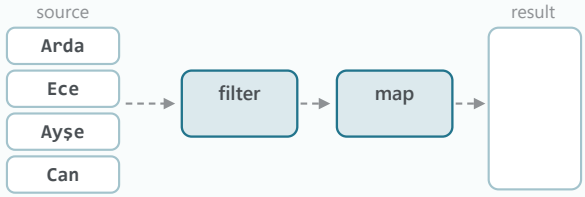
Lazy Evaluation and Single Use

```
List<String> names =
    Arrays.asList("Arda", "Ece", "Ayşe", "Can");
Stream<String> pipe = names.stream()
    .filter(s -> s.startsWith("A"))
    .map(String::toUpperCase);
// nothing processed yet.
List<String> result =
    pipe.collect(Collectors.toList());
System.out.println(result);  // [ARDA, AYŞE]
pipe.count();   // IllegalStateException
```

- **Lazy**: intermediate operations only record a **recipe**; elements are processed when the terminal operation is called.
- Elements pass through the pipeline **one at a time**: the first goes through `filter` and `map`, then the second comes.
- `findFirst` and `limit` stop once they have the result; the remaining elements are never processed.
- A stream is **single-use**: reusing it after a terminal operation throws `IllegalStateException`; get a new stream from the source if needed.

A Lazy Stream Pipeline, Step by Step

```
List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
"Can");
Stream<String> pipe = names.stream()
    .filter(s -> { System.out.println("filter " + s);
                   return s.startsWith("A"); })
    .map(s -> { System.out.println("map " + s);
                return s.toUpperCase(); });
System.out.println("pipe ready");
List<String> result = pipe.collect(Collectors.toList());
System.out.println(result);
```



The pipeline is built: `filter` and `map` are only recorded. No lambda has run; the output is empty.

Step 1 / 10

VARIABLES

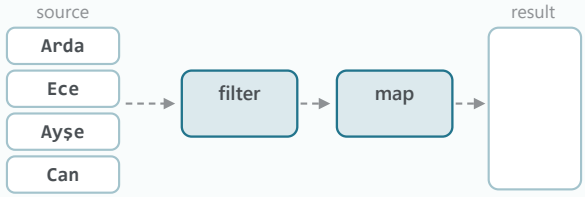
`names = [Arda, Ece, Ayşe, Can]`

`pipe = Stream`

OUTPUT

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
3     .filter(s -> { System.out.println("filter " + s);
4                    return s.startsWith("A"); })
5     .map(s -> { System.out.println("map " + s);
6                 return s.toUpperCase(); });
▶ System.out.println("pipe ready");
8 List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



This line is printed **before** any processing: intermediate operations are lazy.

Step 2 / 10

VARIABLES

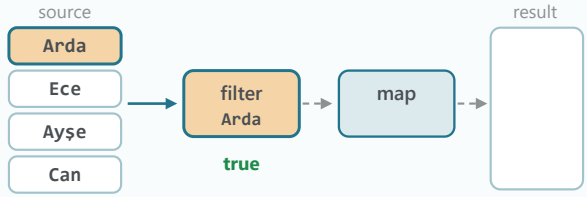
`pipe = Stream`

OUTPUT

```
pipe ready
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
▶     .filter(s -> { System.out.println("filter " + s);
▶                    return s.startsWith("A"); })
5     .map(s -> { System.out.println("map " + s);
                  return s.toUpperCase(); });
7 System.out.println("pipe ready");
▶ List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



`collect` runs the pipeline. The first element, `Arda`, enters `filter`: it starts with "A" → it passes.

Step 3 / 10

VARIABLES

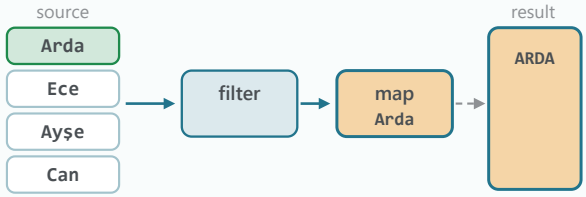
`s = "Arda"`

OUTPUT

```
pipe ready
filter Arda
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
3     .filter(s -> { System.out.println("filter " + s);
4                    return s.startsWith("A"); })
▶     .map(s -> { System.out.println("map " + s);
▶                  return s.toUpperCase(); });
7 System.out.println("pipe ready");
▶ List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



The same element goes straight on to `map`, and `ARDA` is added to the result. Only now is it the second element's turn.

Step 4 / 10

VARIABLES

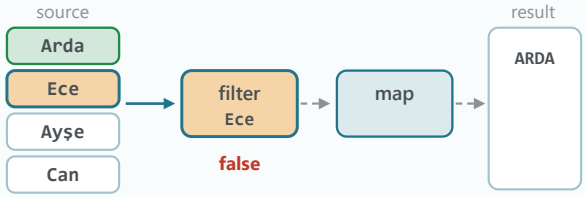
`s = "Arda"`

OUTPUT

```
pipe ready
filter Arda
map Arda
```

A Lazy Stream Pipeline, Step by Step

```
1  List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
   "Can");
2  Stream<String> pipe = names.stream()
       .filter(s -> { System.out.println("filter " + s);
                      return s.startsWith("A"); })
5      .map(s -> { System.out.println("map " + s);
                   return s.toUpperCase(); });
7  System.out.println("pipe ready");
   List<String> result = pipe.collect(Collectors.toList());
9  System.out.println(result);
```



`Ece` fails the condition: it is dropped and never reaches `map`.

Step 5 / 10

VARIABLES

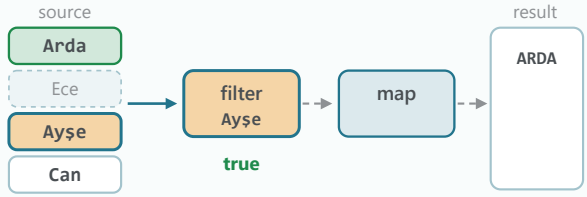
`s = "Ece"`

OUTPUT

```
pipe ready
filter Arda
map Arda
filter Ece
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
▶     .filter(s -> { System.out.println("filter " + s);
▶                    return s.startsWith("A"); })
5     .map(s -> { System.out.println("map " + s);
6                 return s.toUpperCase(); });
7 System.out.println("pipe ready");
▶ List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



`Ayşe` passes the filter.

Step 6 / 10

VARIABLES

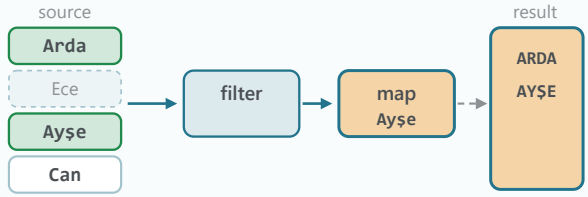
`s = "Ayşe"`

OUTPUT

```
pipe ready
filter Arda
map Arda
filter Ece
filter Ayşe
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
3     .filter(s -> { System.out.println("filter " + s);
4                    return s.startsWith("A"); })
▶     .map(s -> { System.out.println("map " + s);
▶                  return s.toUpperCase(); });
7 System.out.println("pipe ready");
▶ List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



`map` converts to upper case: `AYŞE` is added to the result.

Step 7 / 10

VARIABLES

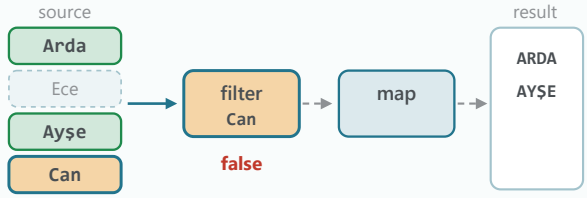
`s = "Ayşe"`

OUTPUT

```
pipe ready
filter Arda
map Arda
filter Ece
filter Ayşe
map Ayşe
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
▶     .filter(s -> { System.out.println("filter " + s);
▶                    return s.startsWith("A"); })
5     .map(s -> { System.out.println("map " + s);
6                 return s.toUpperCase(); });
7 System.out.println("pipe ready");
▶ List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



`Can` is dropped. The source has no elements left.

Step 8 / 10

VARIABLES

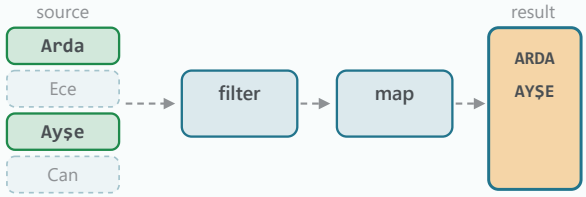
`s = "Can"`

OUTPUT

```
pipe ready
filter Arda
map Arda
filter Ece
filter Ayşe
map Ayşe
filter Can
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
3     .filter(s -> { System.out.println("filter " + s);
4                    return s.startsWith("A"); })
5     .map(s -> { System.out.println("map " + s);
6                 return s.toUpperCase(); });
7 System.out.println("pipe ready");
▶ List<String> result = pipe.collect(Collectors.toList());
9 System.out.println(result);
```



`collect` returns the list. `filter` ran 4 times, `map` only twice.

Step 9 / 10

VARIABLES

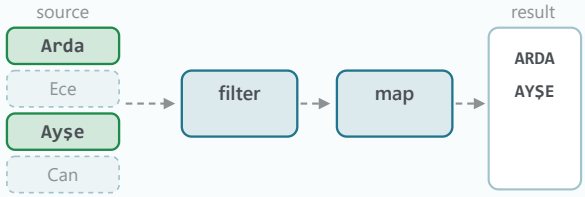
`result = [ARDA, AYŞE]`

OUTPUT

```
pipe ready
filter Arda
map Arda
filter Ece
filter Ayşe
map Ayşe
filter Can
```

A Lazy Stream Pipeline, Step by Step

```
1 List<String> names = Arrays.asList("Arda", "Ece", "Ayşe",
  "Can");
2 Stream<String> pipe = names.stream()
3     .filter(s -> { System.out.println("filter " + s);
4                    return s.startsWith("A"); })
5     .map(s -> { System.out.println("map " + s);
6                 return s.toUpperCase(); });
7 System.out.println("pipe ready");
8 List<String> result = pipe.collect(Collectors.toList());
▶ System.out.println(result);
```



The result is printed. The order of the output shows that elements pass through the pipeline **one at a time**.

Step 10 / 10

VARIABLES

`result = [ARDA, AYŞE]`

OUTPUT

```
pipe ready
filter Arda
map Arda
filter Ece
filter Ayşe
map Ayşe
filter Can
[ARDA, AYŞE]
```

`reduce`, `Collectors` and Primitive Streams

```
List<String> fruits = Arrays.asList(
    "pear", "kiwi", "fig", "plum", "lemon");
int total = Arrays.asList(4, 8, 15, 16, 23)
    .stream().reduce(0, Integer::sum);
Map<Integer, List<String>> byLen = fruits
    .stream().collect(
        Collectors.groupingBy(String::length));
String all = fruits.stream()
    .collect(Collectors.joining(", "));
int t = IntStream.rangeClosed(1, 100).sum();
Optional<String> z = fruits.stream()
    .filter(m -> m.startsWith("z")).findFirst();
System.out.println(z.orElse("none"));
```

VALUES

```
total = 66
byLen = {3=[fig],
  4=[pear, kiwi, plum],
  5=[lemon]}
all = pear, kiwi, fig,
  plum, lemon
t = 5050
none
```

- `reduce`: 0 + 4 + 8 + 15 + 16 + 23.
- `groupingBy` groups, `joining` concatenates.
- `IntStream`: a stream of numbers without boxing; `sum` and `average` built in.
- `Optional`: there may be no result; `orElse`, `ifPresent`.

Side Effects and Parallel Streams

```
List<Integer> list = new ArrayList<>();
IntStream.range(0, 10000).parallel()
    .forEach(list::add);          // WRONG: shared list

List<Integer> safe = IntStream.range(0, 10000).parallel()
    .boxed()
    .collect(Collectors.toList()); // RIGHT: 10000 items
```

RESULT OF THE FIRST PIPELINE (DIFFERS PER RUN)

```
ArrayIndexOutOfBoundsException
or fewer than 10000 items
```

- Lambdas should **not modify** outside variables; build the result with `map`, `collect`, `reduce`.
- `parallel()` / `parallelStream()` splits the work across the common `ForkJoinPool` threads. `ArrayList` is not thread-safe, so the first pipeline breaks.
- A parallel stream is **not always faster**: small data, cheap or I/O-bound work and order-dependent operations (`sorted`, `findFirst`) can slow it; measure first.

Common mistakes: reusing a stream; doing the real work inside `peek`; filling an outside collection with `forEach`; assuming a parallel stream is automatically faster.

CHOOSING

Which Collection to Choose?

Need	Choice	Why
Ordered list, indexed access	`ArrayList`	fast `get(i)`; the default list
Frequent add/remove at the ends, queue, stack	`ArrayDeque`	O(1) at both ends; faster than `Stack` and `LinkedList`
Repeatedly taking the highest-priority element	`PriorityQueue`	`poll` gives the smallest (or highest-priority by the comparator)
Unique elements, fast "contains?"	`HashSet`	`contains` O(1) average
Unique + insertion order	`LinkedHashSet`	order preserved
Unique + sorted, range queries	`TreeSet`	`first`, `headSet`, `subSet`
Fast lookup from key to value	`HashMap`	`get` O(1) average; the default map
Sorted keys	`TreeMap`	`firstKey`, `headMap`, `subMap`

With variables declared by interface type (`List`, `Set`, `Map`, `Queue`, `Deque`), only the `new` line changes when your needs change.

CHOOSING

Unmodifiable Collections

FACTORY METHODS (JAVA 9+)

```
List<String> r = List.of("red", "green");
Set<Integer> primes = Set.of(2, 3, 5);
Map<String, Integer> score =
        Map.of("Ali", 90, "Banu", 85);
System.out.println(r);
System.out.println(score.get("Banu"));
r.add("blue");
```

```
[red, green]
85
```

```
Exception in thread "main"
java.lang.UnsupportedOperationException
```

`add`, `remove` and `put` throw. `null` elements are rejected; `Set.of(1, 1)` → `IllegalArgumentException`.

READ-ONLY VIEW (JAVA 1.2+)

```
List<String> list = new ArrayList<>();
list.add("a");
List<String> readOnly =
        Collections.unmodifiableList(list);
list.add("b");
System.out.println(readOnly);
readOnly.add("c");
```

```
[a, b]
```

```
Exception in thread "main"
java.lang.UnsupportedOperationException
```

`readOnly` is a **view**: it cannot be modified itself, but it changes when the underlying `list` changes. Independent copy: `List.copyOf(list)` (Java 10+). Good for constant lists and hash keys.

Summary

Class	Interface	Order	Duplicates	Underlying structure
`ArrayList`	List	insertion	yes	dynamic array
`LinkedList`	List, Deque	insertion	yes	doubly linked list
`Vector` / `Stack`	List	insertion / LIFO	yes	synchronized array (legacy)
`ArrayDeque`	Deque	FIFO or LIFO	yes	circular array
`PriorityQueue`	Queue	removal by priority	yes	binary heap
`HashSet`	Set	no guarantee	no	hash table
`LinkedHashSet`	Set	insertion	no	hash + linked list
`TreeSet`	NavigableSet	sorted	no	red-black tree
`HashMap`	Map	no guarantee	key: no	hash table
`LinkedHashMap`	Map	insertion	key: no	hash + linked list
`TreeMap`	NavigableMap	sorted keys	key: no	red-black tree

Iteration: `for-each` or `Iterator` (for removal). Algorithms: `Collections.sort`, `shuffle`, `binarySearch`, `copy`, `fill`, `reverse`, `swap`. Streams: `filter`, `map`, `sorted` (intermediate, lazy) and `collect`, `reduce`, `count` (terminal).

PRACTICE

Common Mistakes (1/4)

PRIMITIVE TYPE IN A GENERIC

```
List<int> grades = new ArrayList<>();
```

```
error: unexpected type
        List<int> grades = new ArrayList<>();
             ^
  required: reference
  found:    int
```

Collections hold objects: the right type is `List<Integer>`. `int` values are autoboxed.

ADDING AN ELEMENT OF THE WRONG TYPE

```
List<String> names = new ArrayList<>();
names.add(42);
```

```
error: incompatible types: int cannot be
converted to String
        names.add(42);
                  ^
```

The generic type is checked at compile time. To store a number as a string, add `String.valueOf(42)`.

PRACTICE

Common Mistakes (2/4)

CONFUSING REMOVE(INT) AND REMOVE(OBJECT)

```
List<Integer> list = new ArrayList<>();
Collections.addAll(list, 10, 20, 30);
list.remove(1);                    // index 1
System.out.println(list);
list.remove(Integer.valueOf(10)); // value 10
System.out.println(list);
```

```
[10, 30]
[30]
```

On a `List<Integer>`, `remove(1)` removes **index** 1. To remove a value, pass `Integer.valueOf(…)`.

COMPARING INTEGERS WITH ==

```
List<Integer> a = new ArrayList<>();
Collections.addAll(a, 1000, 1000);
System.out.println(a.get(0) == a.get(1));
System.out.println(a.get(0).equals(a.get(1)));
```

```
false
true
```

`==` compares references (it may happen to work between −128 and 127 because of caching). Compare values with `equals`.

PRACTICE

Common Mistakes (3/4)

ADDING TO AN ARRAYS.ASLIST LIST

```
List<Integer> fixed = Arrays.asList(1, 2, 3);
fixed.add(4);
```

```
Exception in thread "main"
java.lang.UnsupportedOperationException
```

`Arrays.asList` returns a **fixed-size** list (`set` works, `add` / `remove` do not). Fix: `new ArrayList<>(Arrays.asList(…))`.

ITERATING A MAP DIRECTLY WITH FOR-EACH

```
Map<String, Integer> m = new HashMap<>();
for (String s : m) {
    System.out.println(s);
}
```

```
error: for-each not applicable to expression
type
        for (String s : m) {
                         ^
  required: array or java.lang.Iterable
  found:    Map<String,Integer>
```

`Map` is not `Iterable` : iterate over `keySet()` , `values()` or `entrySet()` .

PRACTICE

Common Mistakes (4/4)

AN INDEX THAT DOES NOT EXIST

```
List<String> l = new ArrayList<>();
Collections.addAll(l, "a", "b", "c");
System.out.println(l.get(3));
```

```
Exception in thread "main"
java.lang.IndexOutOfBoundsException:
Index 3 out of bounds for length 3
```

Valid indices are 0 … `size() - 1`. On an empty queue `remove()` also throws `NoSuchElementException`; `poll()` returns `null`.

SORTING TURKISH NAMES

```
List<String> names = new ArrayList<>();
Collections.addAll(names,
        "Zeynep", "Çağla", "Can", "Ömer");
Collections.sort(names);
System.out.println(names);
Locale tr = Locale.forLanguageTag("tr-TR");
Collections.sort(names, Collator.getInstance(tr));
System.out.println(names);
```

```
[Can, Zeynep, Çağla, Ömer]
[Can, Çağla, Ömer, Zeynep]
```

`String` sorts by character code: Ç, Ö, Ş, Ü, İ come after “Z”. A `Collator` (java.text) gives Turkish alphabetical order.

PRACTICE

Practice Problems (1)

```
public class Fibonacci {
    public static int fibonacci(int num) {
        List<Integer> list = new ArrayList<>();
        list.add(0);
        list.add(1);
        for (int i = 2; i < num + 1; i++) {
            list.add(list.get(i - 2) + list.get(i - 1));
        }
        return list.get(num);
    }
    public static void main(String[] args) {
        int result = fibonacci(10);
        System.out.println(result);
    }
}
```

1. What does the program print?
2. How many elements does the list hold when `fibonacci(10)` returns? What are the last three?
3. How many times does the loop run for `fibonacci(1)`?
4. Is a list really needed for this problem?

▶ **Show answers**

PRACTICE

Practice Problems (2)

```
public class Dedup {
    static List<String> dedup(List<String> input) {
        Set<String> seen = new LinkedHashSet<>(input);
        return new ArrayList<>(seen);
    }

    public static void main(String[] args) {
        List<String> l = Arrays.asList("b", "a", "b", "c", "a");
        System.out.println(dedup(l));
    }
}
```

1. What does the program print?
2. What would change with a `HashSet` or `TreeSet` instead of the `LinkedHashSet`?

▶ **Show answers**

PRACTICE

Practice Problems (3)

```
static boolean isBalanced(String expr) {
    Deque<Character> stack = new ArrayDeque<>();
    for (char c : expr.toCharArray()) {
        if (c == '(' || c == '[') {
            stack.push(c);
        } else if (c == ')' || c == ']') {
            if (stack.isEmpty()) return false;
            char open = stack.pop();
            if (c == ')' && open != '(') return false;
            if (c == ']' && open != '[') return false;
        }
    }
    return stack.isEmpty();
}
```

Balanced brackets: every opening bracket must close with the right type, in the right order.

1. What do `isBalanced("(a[b]c)")`, `isBalanced("(a[b)c]")` and `isBalanced("((")` return?
2. Why use a stack?

▶ **Show answers**

PRACTICE

Practice Problems (4)

```
Deque<Integer> d = new ArrayDeque<>();
d.offerLast(1);
d.offerFirst(2);
d.offerLast(3);
d.push(4);
System.out.println(d);
System.out.println(d.pollLast() + d.pop());
System.out.println(d);
```

```
Map<String, Integer> m = new TreeMap<>();
m.put("b", 1);
m.put("a", 2);
m.put("b", 3);
Set<Integer> s = new TreeSet<>(m.values());
s.add(2);
System.out.println(m + " " + s + " " + m.size());
```

What do the two fragments print? (`push` = `offerFirst`, `pop` = `removeFirst`.)

▶ **Show answers**

PRACTICE

Exercises (1)

1. Which collection? (a) fast lookup of a student by number, (b) listing distinct words alphabetically, (c) print jobs, (d) a browser's "back" button, (e) calling emergency patients by urgency.
2. Using a `HashMap`, write a method `mostFrequent` that returns the most frequent word in a `String[]`.

▶ **Show answers**

PRACTICE

Exercises (2)

3. Write the lottery draw without `shuffle`, using `Random` and a `TreeSet`: 6 distinct numbers from 1–49, printed in order.

▶ **Show answers**

PRACTICE

Exercises (3)

4. Build a `Map<String, List<Integer>>` holding each student's list of grades; print the averages for Ali: 80, 90 and Banu: 70.

▶ **Show answers**

SUMMARY

References

- Oracle, *The Java Tutorials*: Collections (Interfaces, Implementations, Algorithms) and Generics.
- Oracle, *Java SE 21 API Documentation*: package `java.util` (`Collection`, `List`, `Map`, `Collections`…).
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (4.5 Parameterized Types, 5.1.7 Boxing Conversion, 14.14.2 Enhanced for).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 16: Generic Collections).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly (Chapter 11: Data Structures).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 9: Collections).
- M. Naftalin and P. Wadler, *Java Generics and Collections*, O'Reilly.
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley (Item 11: override hashCode when you override equals; Item 64: refer to objects by their interfaces).

The End

Chapter 5: Collections

Previous: Chapter 4.5: Abstract Classes and Interfaces · Next: Chapter 6: Exceptions