Lists

The first topic in the Collections category: the `List` interface, the performance difference between `ArrayList` and `LinkedList` shown with a real warmed-up measurement, immutable lists via `List.of()` / `Collections.unmodifiableList()` / `List.copyOf()`, safe iteration with `Iterator` / `ListIterator`, sorting with `List.sort(Comparator)`, and `subList()` / `toArray()`.

Beginner 20 min

Last updated: August 17, 2026
Read online: https://www.learnforgex.com/en/topics/lists

Lists

In the Java Basics category you saw how to model single values, fixed sets of fields, and special-purpose behavior. But most real programs need to hold collections of data whose size isn't known ahead of time and that grow and shrink at runtime -- the items in a shopping cart, the error messages from a form, the records returned by an API. That's what the Collections category is about, and our first stop is Java's most widely used collection type: `List`.

What Is a List?

`List<E>` is an interface that extends `java.util.Collection` and adds two key guarantees: elements are **ordered** (insertion order is preserved) and **indexed** (any element can be accessed directly with `get(index)`). Unlike `Set`, the same value can appear in a `List` more than once.

Since `List` is an interface, it can't be instantiated directly; its two most commonly used implementations are `ArrayList` and `LinkedList`. Both honor the same contract but use completely different internal data structures -- we'll see what that difference means in practice with a real measurement shortly.

Why Does It Exist?

Java arrays have a fixed size -- once you create an `int[10]`, it always has exactly 10 slots, no more, no fewer. But in the real world, the number of elements is almost never known in

advance: how many items will the user add to a cart, how many rows will a query return? `List` solves this -- it grows and shrinks dynamically with `add()` / `remove()`, removing the fixed-size constraint of arrays.

History

The `List` interface arrived in Java 1.2 (1998) as part of the newly introduced **Collections Framework** -- before that, Java only had the old, synchronized (and therefore slow) `Vector` class. `ArrayList` came in the same release as `Vector`'s modern counterpart, without the synchronization overhead. Java 5 (2004) added generics (`List<E>`), bringing type safety; Java 9 (2017) made creating an immutable list a one-liner with `List.of()`.

Basic List Operations

The most commonly used `List` methods are `add()` (appends to the end), `get(index)` (reads), `set(index, value)` (overwrites), `remove()` (removes by value or by index), `size()`, `contains()`, and `indexOf()`. Iterating a `List` with a for-each loop also works naturally, since `List` extends `Iterable`.

```
import java.util.ArrayList;
import java.util.List;

public class ListBasicsExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("apple");
        fruits.add("pear");
        fruits.add("banana");
        fruits.add("apple"); // a List allows duplicate elements

        System.out.println("List: " + fruits);
        System.out.println("Size: " + fruits.size());
        System.out.println("index 0: " + fruits.get(0));
        System.out.println("Contains 'banana'? " + fruits.contains("banana"));
        System.out.println("First index of 'apple': " + fruits.indexOf("apple"));

        fruits.set(1, "cherry"); // overwrite index 1
        System.out.println("After set(1, cherry): " + fruits);

        fruits.remove("cherry"); // remove by value
        System.out.println("After remove(cherry): " + fruits);

        fruits.remove(0); // remove by index
        System.out.println("After remove(0): " + fruits);

        for (String fruit : fruits) {
            System.out.println("for-each: " + fruit);
        }
    }
}
```

> `remove()` has two overloads, and they're easy to mix up on a boxed numeric type like `List<Integer>`: `remove(int index)` removes by index, while `remove(Object o)` removes by value. When `list.remove(2)` is called on a `List<Integer>`, `2` is not auto-boxed to `Integer` -- it's interpreted directly as an `int`, so it removes the element at index 2, not the element with value 2. To remove by value, you need `list.remove(Integer.valueOf(2))`.

ArrayList vs. LinkedList: Two Different Implementations

`ArrayList` is backed internally by a growable array -- `get(index)` jumps straight to a memory address, making it **O(1)**. `LinkedList` is a doubly-linked list -- each element points to the previous and next one; to reach a given index, `get(index)` has to walk **one element at a time** (from whichever end is closer), making it **O(n)**.

The reverse is also true: inserting at the front of an `ArrayList` (`add(0, x)`) requires shifting every subsequent element one slot over -- **O(n)**. Inserting at the front of a `LinkedList` is just updating a couple of references -- **O(1)**.

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

public class ArrayListVsLinkedListExample {
    public static void main(String[] args) {
        int size = 20_000;
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();
        for (int i = 0; i < size; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }

        int middle = size / 2;
        int warmupRounds = 3_000;
        int timedRounds = 3_000;

        // Warm-up: run both paths a lot BEFORE measuring, so the JIT can optimize both --
        // a single, un-warmed-up measurement can be misleading (whichever path runs first
        // can look unfairly slow).
        for (int i = 0; i < warmupRounds; i++) {
            arrayList.get(middle);
            linkedList.get(middle);
        }

        long arrayListStart = System.nanoTime();
        for (int i = 0; i < timedRounds; i++) {
            arrayList.get(middle);
        }
        long arrayListNanos = System.nanoTime() - arrayListStart;

        long linkedListStart = System.nanoTime();
```

```
        for (int i = 0; i < timedRounds; i++) {
            linkedList.get(middle);
        }
        long linkedListNanos = System.nanoTime() - linkedListStart;

        System.out.println("get(middle element), " + timedRounds + " times, a " + size + "-
element list:");
        System.out.println("  ArrayList:  " + (arrayListNanos / 1_000_000) + " ms");
        System.out.println("  LinkedList: " + (linkedListNanos / 1_000_000) + " ms");

        // Second measurement: inserting at the front (add(0, ...))
        List<Integer> arrayList2 = new ArrayList<>();
        List<Integer> linkedList2 = new LinkedList<>();
        int addRounds = 20_000;

        for (int i = 0; i < 2_000; i++) {
            arrayList2.add(0, i);
            linkedList2.add(0, i);
        }
        arrayList2.clear();
        linkedList2.clear();

        long arrayListAddStart = System.nanoTime();
        for (int i = 0; i < addRounds; i++) {
            arrayList2.add(0, i);
        }
        long arrayListAddNanos = System.nanoTime() - arrayListAddStart;

        long linkedListAddStart = System.nanoTime();
        for (int i = 0; i < addRounds; i++) {
            linkedList2.add(0, i);
        }
        long linkedListAddNanos = System.nanoTime() - linkedListAddStart;

        System.out.println();
        System.out.println("add(0, element), " + addRounds + " times (inserting at the
front):");
        System.out.println("  ArrayList:  " + (arrayListAddNanos / 1_000_000) + " ms");
        System.out.println("  LinkedList: " + (linkedListAddNanos / 1_000_000) + " ms");
    }
}
```

This example confirms it with a real, warmed-up measurement: on a list of 20,000 elements, calling `get()` on the middle element 3,000 times is immeasurably fast on `ArrayList` (0 ms), while on `LinkedList` it takes several milliseconds (around 48 ms) -- because every call has to walk halfway through the list. Conversely, inserting 20,000 elements at the front finishes almost instantly on `LinkedList` (around 1 ms), while `ArrayList` takes noticeably longer (around 16-17 ms) -- every insertion has to shift all the elements accumulated so far.

> In practice `ArrayList` is almost always the right choice -- random access (`get(index)`) is a far more common operation, and on modern hardware contiguous memory access also gets a speed boost from CPU caching. Only reach for `LinkedList` if you're genuinely doing frequent insertions/removals at the ends of the list (for example, using it as a queue).

Immutable Lists: List.of(), Collections.unmodifiableList(), List.copyOf()

Sometimes you want to guarantee a list never changes -- a fixed list of configuration values, for example. Java offers three different immutable-list tools, and the difference between them matters: `List.of(...)` builds a brand-new unmodifiable list from scratch; `Collections.unmodifiableList(list)` returns an unmodifiable **view** of an existing list -- if the original list changes, the view changes too; `List.copyOf(list)` creates a completely independent, separate immutable **copy**.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ListOfImmutableExample {
    public static void main(String[] args) {
        List<String> immutable = List.of("red", "green", "blue");
        System.out.println("List.of(): " + immutable);

        try {
            immutable.add("yellow");
        } catch (UnsupportedOperationException e) {
            System.out.println("add() on a List.of() result: " + e.getClass().
getSimpleName());
        }

        try {
            immutable.set(0, "black");
        } catch (UnsupportedOperationException e) {
            System.out.println("set() on a List.of() result: " + e.getClass().
getSimpleName());
        }

        // Collections.unmodifiableList(): an UNMODIFIABLE "view" of an existing list
        List<String> mutable = new ArrayList<>(List.of("a", "b"));
        List<String> readOnlyView = Collections.unmodifiableList(mutable);
        try {
            readOnlyView.add("c");
        } catch (UnsupportedOperationException e) {
            System.out.println("add() on unmodifiableList(): " + e.getClass().
getSimpleName());
        }

        // But watch out: unmodifiableList() is just a VIEW, the original list can still
change
        mutable.add("c");
        System.out.println("The view changes when the original list changes: " +
readOnlyView);

        // List.copyOf(): creates a completely independent, separate immutable COPY
        List<String> independentCopy = List.copyOf(mutable);
        mutable.add("d");
        System.out.println("Original list changed: " + mutable);
        System.out.println("List.copyOf() copy was NOT affected: " + independentCopy);
    }
}
```

> The list returned by `Collections.unmodifiableList()` being "read-only" doesn't mean the original list won't change -- it only blocks modification through the view itself. If you want a truly independent, unchanging copy, use `List.copyOf()`.

Iterator and ListIterator

If you need to remove or add elements while iterating a `List`, calling `List.remove()` directly throws a `ConcurrentModificationException` -- because a for-each loop uses an `Iterator` under the hood, and the `Iterator` detects that the list changed "unexpectedly". The correct approach is to use the `Iterator.remove()` method, which also updates the iterator's own internal bookkeeping. `ListIterator` is an extended version of `Iterator`: it can move in both directions (`hasPrevious()`/`previous()`) and also supports `set()`/`add()` while iterating.

```
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class IteratorExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));

        // Safe removal with Iterator: use Iterator.remove() instead of calling
        // List.remove() DURING a for-each loop.
        Iterator<Integer> it = numbers.iterator();
        while (it.hasNext()) {
            int value = it.next();
            if (value % 2 == 0) {
                it.remove(); // safe -- the iterator updates its own internal bookkeeping
            }
        }
        System.out.println("Even numbers removed with Iterator.remove(): " + numbers);

        // ListIterator: unlike Iterator, it can move in BOTH directions (hasPrevious
/previous)
        // and also supports add()/set().
        List<String> letters = new ArrayList<>(List.of("a", "b", "c"));
        ListIterator<String> listIt = letters.listIterator();
        while (listIt.hasNext()) {
            String value = listIt.next();
            listIt.set(value.toUpperCase());
        }
        System.out.println("Converted to uppercase with ListIterator.set(): " + letters);

        while (listIt.hasPrevious()) {
            System.out.println("going backwards: " + listIt.previous());
        }

        // REAL ERROR: calling List.remove() directly DURING a for-each loop
        List<Integer> unsafe = new ArrayList<>(List.of(10, 20, 30, 40));
        try {
            for (Integer value : unsafe) {
                if (value == 20) {
                    unsafe.remove(value); // throws ConcurrentModificationException
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("List.remove() during a for-each loop: " + e.getClass().
getSimpleName());
        }
    }
}
```

Sorting: List.sort() and Comparator

`List.sort(Comparator)` sorts the list **in place** -- it doesn't return a new list, it mutates the existing one. You can pass `Comparator.naturalOrder()` (natural ordering), `Comparator.`

`reverseOrder()` (reversed), or use `Comparator.comparing(...)` to define a custom ordering based on a specific field of an object. `Collections.sort(list)` is the older way, predating `List.sort()` (Java 8) -- it still works, but `List.sort()` is now preferred.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class SortingExample {
    record Person(String name, int age) {
    }

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(List.of(5, 3, 8, 1, 9, 2));

        // List.sort(): sorts in place; Comparator.naturalOrder() for natural ordering
        numbers.sort(Comparator.naturalOrder());
        System.out.println("Natural order: " + numbers);

        numbers.sort(Comparator.reverseOrder());
        System.out.println("Reversed order: " + numbers);

        // Collections.sort(): the old way, predating List.sort() (pre-Java 8), still works
        List<String> words = new ArrayList<>(List.of("banana", "apple", "kiwi", "pear"));
        Collections.sort(words);
        System.out.println("Collections.sort(): " + words);

        // Comparator.comparing() + thenComparing(): sorting objects by a field
        List<Person> people = new ArrayList<>(List.of(
                new Person("Alice", 30),
                new Person("Bob", 25),
                new Person("Alice", 22)
        ));

        people.sort(Comparator.comparing(Person::name).thenComparing(Person::age));
        System.out.println("By name, then age: " + people);

        people.sort(Comparator.comparingInt(Person::age).reversed());
        System.out.println("By age, descending: " + people);
    }
}
```

> Chaining like `Comparator.comparing(Person::name).thenComparing(Person::age)` means "sort by name first, and if names are equal, sort by age" -- for sorting by multiple fields, it's far more readable than a hand-written `compareTo()`.

subList() and toArray()

`subList(from, to)` returns a **view** of the original list between `from` (inclusive) and `to` (exclusive) -- not an independent copy. Changes made through this view (adding, removing, `set()`) are reflected in the original list. `toArray()` offers three ways to turn a `List` into an

array: the no-argument version returns an `Object[]` that loses type information, while `toArray(new String[0])` or (Java 11+) `toArray(String[]::new)` produce a correctly typed array.

```
import java.util.ArrayList;
import java.util.List;

public class SubListAndToArrayExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9));

        // subList(from, to): from inclusive, to exclusive -- NOT an independent copy, it's
        // a "view" of the original list.
        List<Integer> middle = numbers.subList(3, 6);
        System.out.println("subList(3, 6): " + middle);

        // Changes made through the subList also change the ORIGINAL list
        middle.set(0, 999);
        System.out.println("Original list after set(0, 999) via subList: " + numbers);

        middle.clear();
        System.out.println("Original list after clear() via subList: " + numbers);

        // toArray(): two ways to convert a List to an array
        List<String> letters = List.of("x", "y", "z");

        Object[] rawArray = letters.toArray();
        System.out.println("toArray() (Object[]): " + rawArray.length + " elements");

        String[] typedArray = letters.toArray(new String[0]);
        System.out.println("toArray(new String[0]) (String[]): " + String.join(", ",
typedArray));

        // toArray(IntFunction) -- Java 11+, a type-safe array without specifying the size
        String[] typedArray2 = letters.toArray(String[]::new);
        System.out.println("toArray(String[]::new): " + String.join(", ", typedArray2));
    }
}
```

> The fact that `subList()` returns a view rather than a copy is a common pitfall -- calling `clear()` on the view also removes that range from the original list. If you want an independent sub-list, copy it explicitly with `new ArrayList<>(numbers.subList(3, 6))`.

Best Practices

- **Default to `ArrayList`**, and only consider `LinkedList` (or better yet, `ArrayDeque`) if you're doing frequent insertions/removals at the ends of the list.
- **Prefer `List.of()` for a list that won't change** -- it both signals intent clearly and throws `UnsupportedOperationException` on the first accidental modification attempt, at least catching it at runtime.

- **Use `Iterator.remove()` / `ListIterator` if you need to remove or add elements while iterating**, not `List.remove()` directly.
- **Use a `Comparator.comparing(...).thenComparing(...)` chain for sorting by multiple fields** -- it's far less error-prone than a hand-written `compareTo()`.

Common Mistakes

- **Confusing `remove(int)` with `remove(Object)` on a `List<Integer>`.** `list.remove(2)` removes the element at index 2; to remove the element with value 2, you need `list.remove(Integer.valueOf(2))`.
- **Calling `List.remove()` directly during a for-each loop.** This throws `ConcurrentModificationException` -- use `Iterator.remove()` instead.
- **Assuming `subList()` returns an independent copy.** It's a view; changes made through it are reflected in the original list.
- **Choosing `LinkedList` for a scenario dominated by random access (`get(index)`).** `ArrayList`'s O(1) access versus `LinkedList`'s O(n) access adds up to a measurable performance difference on large lists.

Summary, Cheat Sheet, and Glossary

`List<E>` is an ordered, indexed collection interface that allows duplicate elements. `ArrayList` is fast for random access (O(1)), while `LinkedList` is fast for inserting/removing at the ends (O(1)). `List.of()` / `List.copyOf()` create immutable lists, while `Collections.unmodifiableList()` returns a read-only view of an existing list. Use `Iterator` / `ListIterator` for safe modification while iterating, and `List.sort(Comparator)` for sorting.

Quick reference:

```
List<String> list = new ArrayList<>();      // dynamic-array backed, the default choice
List<String> linked = new LinkedList<>();    // when insert/remove at the ends dominates
List<String> immutable = List.of("a", "b");  // unmodifiable, from scratch
List<String> copy = List.copyOf(list);       // unmodifiable, independent copy
List<String> view = Collections.unmodifiableList(list); // unmodifiable VIEW
list.sort(Comparator.comparing(String::length));         // in-place sort
List<String> part = new ArrayList<>(list.subList(1, 3)); // independent sub-list copy
```

Glossary

List — A `Collection` sub-interface that is ordered, indexed, and allows duplicate elements.

ArrayList — The `List` implementation backed by a dynamic array, with O(1) random access.

LinkedList — The `List` implementation backed by a doubly-linked list, with O(1) insertion /removal at the ends.

View — An object, such as the one returned by `subList()` / `unmodifiableList()`, that stays connected to the original data rather than being an independent copy.

ConcurrentModificationException — The exception thrown when a collection is modified from outside the `Iterator` that is currently traversing it.