Enhanced for Loop

The fourth topic in the Control Flow category: the for-each syntax over arrays /collections, its three real limitations (no index access, the loop variable being a copy plus structural changes throwing ConcurrentModificationException, and no parallel iteration of multiple collections), and when to use it versus classic `for`.

Beginner 20 min

Last updated: October 19, 2026
Read online: https://www.learnforgex.com/en/topics/enhanced-for-loop

---

Enhanced for Loop

The fourth topic in the Control Flow category: the enhanced for (for-each) loop -- a syntax built on top of the classic "for Loop" lesson that walks every element of an array or collection WITHOUT writing a counter variable. Classic `for` is required whenever you need the POSITION (index); enhanced for does the same job with far less code whenever you only care about the VALUES themselves -- position doesn't matter at all. This lesson covers the basic syntax, three real LIMITATIONS (no index, the loop variable being a copy, no parallel iteration), and the choice between it and classic `for`.

What Is an Enhanced for Loop?

Enhanced for, written `for (Type element : collection)`, reads as "for each element in collection" -- unlike classic `for`, none of the INITIALIZATION/CONDITION/UPDATE parts are written; the JVM manages them behind the scenes. It works on any array, or any type implementing the `Iterable` interface (e.g. `List`, `Set`).

Why Does It Exist?

Walking an array/collection with classic `for` -- `for (int i = 0; i < array.length; i++) { array[i] ... }` -- is a common pattern, but carries two unnecessary risks: getting the bound condition wrong (`<=` instead of `<`, see "Common Mistakes" in the "for Loop" lesson) and confusing the index with the VALUE (accidentally using `i` instead of `array[i]`). In the MAJORITY of cases where position doesn't matter at all -- only the values themselves do -- both risks are unnecessary; enhanced for removes them from the syntax entirely.

History

Enhanced for was introduced in Java 5 (2004), in the SAME release as generics and autoboxing (see "Wrapper Classes & Autoboxing"), along with the `Iterable` / `Iterator` interfaces -- the goal was to hide the manual `Iterator` usage pattern (`while (it.hasNext()) { Type element = it.next(); ... }`) that every collection walk had required up to that point, collapsing it into shorter syntax. It did NOT replace classic `for` ; it simply provided an ALTERNATIVE for the cases where position isn't needed.

Basic Enhanced for Syntax (Arrays)

On an array, enhanced for assigns each element to the `element` variable in order -- no index is ever written, and the array's bounds are checked automatically by the JVM (so there's NO risk of `ArrayIndexOutOfBoundsException`).

```
public class EnhancedForArrayExample {
    public static void main(String[] args) {
        int[] scores = {70, 85, 92, 60, 78};

        // "for (Type element : array)" reads as "for each element in array" --
        // no counter, no condition, no update to write yourself.
        for (int score : scores) {
            System.out.println("Score: " + score);
        }

        int total = 0;
        for (int score : scores) {
            total += score;
        }
        System.out.println("Total: " + total);
    }
}
```

Expected Output

```
Score: 70
Score: 85
Score: 92
Score: 60
Score: 78
Total: 385
```

Enhanced for with Collections

Enhanced for isn't limited to arrays -- it works with the same syntax on ANY type implementing `Iterable` (that is, `List` , `Set` , and nearly every collection class).

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

public class EnhancedForCollectionExample {
    public static void main(String[] args) {
        List<String> languages = new ArrayList<>();
        languages.add("Java");
        languages.add("Kotlin");
        languages.add("TypeScript");

        // Enhanced for works on anything Iterable, not just arrays -- List,
        // Set, and every other collection type included.
        for (String language : languages) {
            System.out.println("Language: " + language);
        }

        int totalLength = 0;
        for (String language : languages) {
            totalLength += language.length();
        }
        System.out.println("Total character count: " + totalLength);
    }
}
```

Expected Output

```
Language: Java
Language: Kotlin
Language: TypeScript
Total character count: 20
```

Limitation: No Index Access

Enhanced for gives you each element's VALUE but never its POSITION (index) -- if you need output like "1st place: X", you either need to keep a manual counter variable or use classic `for` (see "Iterating Arrays with a Classic for Loop" in the "for Loop" lesson).

```
public class NoIndexAccessExample {
    public static void main(String[] args) {
        String[] winners = {"Alice", "Bob", "Carol"};

        // Enhanced for gives you each VALUE, but never its position -- there
        // is no built-in way to print "1st place: Alice" from inside the loop.
        System.out.println("Without an index (position is missing):");
        for (String winner : winners) {
            System.out.println("Winner: " + winner);
        }

        // A manual counter alongside enhanced for works, but at that point a
        // classic for loop (see the "for Loop" lesson) is usually clearer --
        // you're already tracking an index by hand.
        System.out.println("With a manual counter:");
        int place = 1;
        for (String winner : winners) {
            System.out.println("Place " + place + ": " + winner);
            place++;
        }
    }
}
```

Expected Output

```
Without an index (position is missing):
Winner: Alice
Winner: Bob
Winner: Carol
With a manual counter:
Place 1: Alice
Place 2: Bob
Place 3: Carol
```

Limitation: The Loop Variable and Structural Changes

Enhanced for's loop variable is a COPY of each element -- assigning it a new value does NOT change the underlying array/collection. Also, changing a `List`'s STRUCTURE (adding /removing an element) while an enhanced for is running over it generally throws `ConcurrentModificationException` -- this is how the loop detects that the collection changed out from under it. The real way to remove elements safely (`Iterator.remove()`) is covered in "Iterator and ListIterator" in the "Lists" lesson.

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

public class ModifyingDuringIterationExample {
    public static void main(String[] args) {
        // The loop variable is a COPY of each value -- reassigning it does
        // NOT change the underlying array. A common misconception.
        int[] numbers = {1, 2, 3};
        for (int n : numbers) {
            n = n * 10; // only changes the LOCAL copy, not numbers[]
        }
        System.out.println("Array after the loop (unchanged): " + numbers[0] + ", " +
numbers[1] + ", " + numbers[2]);

        // Removing an element from a List WHILE iterating it with enhanced
        // for throws ConcurrentModificationException -- the loop detects
        // that the collection's structure changed underneath it.
        List<String> items = new ArrayList<>();
        items.add("remove-me");
        items.add("keep");
        items.add("keep");

        try {
            for (String item : items) {
                if (item.equals("remove-me")) {
                    items.remove(item);
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("Caught: " + e.getClass().getSimpleName());
        }
    }
}
```

> Don't rely on `ConcurrentModificationException` ALWAYS being thrown when removing from a list -- removing an element near the END of a list can, due to how the internal counters happen to line up, avoid throwing at all (silently producing a wrong result instead). The only safe rule is to never make STRUCTURAL changes inside an enhanced for.

for-each vs Classic for: When to Use Which

When position is NOT needed (only values are being processed), enhanced for should be preferred -- shorter, less error-prone. When position IS needed (see the previous section), or when the array/collection needs to be modified IN PLACE (see "Iterating Arrays with a Classic for Loop" in the "for Loop" lesson), classic `for` is required.

```
public class ForEachVsClassicForExample {
    public static void main(String[] args) {
        int[] numbers = {2, 4, 6, 8, 10};

        // Same result, two ways.
        System.out.println("Classic for (has an index, more to write):");
        int classicTotal = 0;
        for (int i = 0; i < numbers.length; i++) {
            classicTotal += numbers[i];
        }
        System.out.println("Total: " + classicTotal);

        System.out.println("Enhanced for (no index, less to write):");
        int enhancedTotal = 0;
        for (int number : numbers) {
            enhancedTotal += number;
        }
        System.out.println("Total: " + enhancedTotal);
    }
}
```

Expected Output

```
Classic for (has an index, more to write):
Total: 30
Enhanced for (no index, less to write):
Total: 30
```

Limitation: Iterating Multiple Collections in Parallel

Enhanced for walks ONE `Iterable` at a time -- there's no direct way to walk two arrays /collections "together, at the same position" (nesting two enhanced-for loops pairs EVERY element with EVERY element of the other collection, which is NOT the intended behavior). Classic `for`, with a single shared index, solves this directly (see "Multiple Variables in for" in the "for Loop" lesson).

```
public class ParallelIterationLimitationExample {
    public static void main(String[] args) {
        String[] names = {"Alice", "Bob", "Carol"};
        int[] ages = {30, 25, 35};

        // Enhanced for iterates ONE Iterable at a time -- there's no direct
        // way to walk two arrays together, because there's no shared index.
        // Nesting two enhanced-for loops would pair EVERY name with EVERY
        // age, not the matching ones.

        // A classic for loop, sharing a single index across both arrays,
        // solves this cleanly (see "Multiple Variables in for" in the
        // "for Loop" lesson for the two-index version of this pattern).
        for (int i = 0; i < names.length; i++) {
            System.out.println(names[i] + " is " + ages[i] + " years old.");
        }
    }
}
```

Expected Output

```
Alice is 30 years old.
Bob is 25 years old.
Carol is 35 years old.
```

Best Practices

- **Prefer enhanced for whenever position (the index) isn't needed** -- shorter, and immune to boundary mistakes.
- **Don't change a collection's STRUCTURE (adding/removing elements) inside an enhanced for** -- use `Iterator.remove()`, or a classic `for` (iterating backward), when that's needed.
- **Remember that assigning a new value to the loop variable does NOT change the array/collection** -- this is a common false expectation.
- **Use classic `for` with a single shared index when you need to walk two collections in parallel.**

Common Mistakes

- **Assuming that changing enhanced for's loop variable updates the array /collection.** The loop variable is only a copy -- the underlying data structure is UNAFFECTED.
- **Calling `list.remove(...)` directly inside an enhanced for over a `List` and getting `ConcurrentModificationException`.** Safe removal requires `Iterator.remove()`.
- **Trying to use enhanced for when position is needed, ending up forced to track a manual counter anyway.** At that point, classic `for` already requires less code.

- **Nesting two enhanced-for loops to pair up two arrays.** This pairs every element with EVERY element of the other one -- not parallel iteration, but a cartesian product.

Summary, Cheat Sheet, and Glossary

Enhanced for (`for (Type element : collection)`) walks every element of an array/collection without writing an index -- it works on any type implementing `Iterable`. It has three real limitations: POSITION is inaccessible, assigning the loop variable does NOT change the underlying data (and structural changes generally throw `ConcurrentModificationException`), and multiple collections cannot be walked in PARALLEL. In all three cases, classic `for` (see the "for Loop" lesson) is still the right tool.

Quick reference:

```
for (int element : array) {
    // VALUE only, no index
}

for (String element : list) {
    // works on any type implementing Iterable
}

// WRONG -- does not change the underlying array:
for (int element : array) {
    element = element * 2;
}

// Classic for when position is needed:
for (int i = 0; i < array.length; i++) {
    System.out.println(i + ": " + array[i]);
}
```

Glossary

Enhanced for (for-each) — A loop syntax that walks every element of an array/collection without writing an index.

Iterable — The interface every type walkable with enhanced for implements (including `List`, `Set`).

ConcurrentModificationException — The runtime exception generally thrown when a collection is STRUCTURALLY changed while an enhanced for is running over it.