for Loop

The third topic in the Control Flow category: the classic three-part (init;condition; update) `for` syntax, exiting early with `break`, skipping a step with `continue`, deliberate /unintentional infinite loops, multiple variables via comma-separated init/update, and classic index-based array iteration.

Beginner 20 min

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

---

for Loop

The third topic in the Control Flow category: the for loop -- a control structure that REPEATS a block of code a known or countable number of times. `if` / `else` (see the "if / else" lesson) and `switch` (see the "switch" lesson) decide whether to run a code path ONCE; `for` runs the same block MULTIPLE times, driven by a counter variable that advances on each pass. This lesson covers the classic `for` syntax, controlling loop flow with `break` / `continue`, and index-based iteration over an array.

What Is a for Loop?

A `for` loop combines three parts in a single line: INITIALIZATION (runs once, before the loop starts), CONDITION (checked before every iteration; the loop ends once it's `false`), and UPDATE (runs at the END of every iteration). These three parts, as in `for (int i = 0; i < 5; i++)`, are separated by semicolons. The counter variable declared in the initialization (`i`) exists only within the loop's own scope -- it's inaccessible once the loop ends.

Why Does It Exist?

Repeating the same operation a fixed number of times (or until a condition holds) is one of the most fundamental needs in programming -- processing every element of a list, retrying an operation N times, checking every number in a range. Writing this WITHOUT a loop would mean manually copying the same code N times -- impossible if N itself is a variable, and hard to maintain and error-prone even if N is fixed. `for` bundles the INITIALIZATION /CONDITION/UPDATE logic of counter-based repetition into one compact place.

History

Like `if` / `switch`, the `for` loop carried over from C (1972) to Java (1995) almost verbatim -- the three-part (`init; condition; update`) syntax is a direct inheritance from C and hasn't changed since. Java 5 (2004) ADDED the "enhanced for" (`for-each`) syntax alongside classic `for`, for iterating array/collection elements without an index (see the "Enhanced for Loop" lesson) -- it did NOT replace classic `for`; both still exist side by side, for different purposes.

Basic for Loop Syntax

The most common form is a counter starting at `0`, counting up to a limit (with `<` or `<=`), incrementing by `1` on each step (`i++`) -- but all three parts are fully flexible: you can count down with `i--`, or use a different step size (`i += 2`).

```
public class ForLoopBasicsExample {
    public static void main(String[] args) {
        // The three parts of a for loop, separated by semicolons:
        // initialization (runs once) ; condition (checked before each iteration) ; update
(runs after each iteration)
        for (int i = 1; i <= 5; i++) {
            System.out.println("i = " + i);
        }

        // The loop variable declared in the header is scoped to the loop --
        // it does not exist after the loop ends.
        int total = 0;
        for (int i = 1; i <= 5; i++) {
            total += i;
        }
        System.out.println("Sum 1..5 = " + total);

        // Counting downward works the same way, with a decrementing update.
        for (int i = 5; i >= 1; i--) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

Expected Output

```
i = 1
i = 2
i = 3
i = 4
i = 5
Sum 1..5 = 15
5 4 3 2 1
```

Exiting a Loop with break

`break` immediately terminates a loop, REGARDLESS of whether the condition is still `true` -- control jumps straight to the line right after the loop. Its most common use is in a SEARCH, where continuing to check the remaining elements is pointless once the target has been found.

```
public class BreakExample {
    public static void main(String[] args) {
        int[] numbers = {4, 9, 15, 23, 42, 7};
        int target = 23;
        int foundIndex = -1;

        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] == target) {
                foundIndex = i;
                // break immediately exits the loop -- no point checking the
                // remaining elements once the target is found.
                break;
            }
        }

        if (foundIndex != -1) {
            System.out.println("Found " + target + " at index " + foundIndex);
        } else {
            System.out.println(target + " not found");
        }

        // Without break, every iteration would run even after finding a
        // match -- wasted work, and the LAST match would silently win
        // instead of the first one.
        int lastMatchIndex = -1;
        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] > 10) {
                lastMatchIndex = i;
            }
        }
        System.out.println("Last index > 10 (no break): " + lastMatchIndex);
    }
}
```

Expected Output

```
Found 23 at index 3
Last index > 10 (no break): 4
```

Skipping an Iteration with continue

`continue` skips the REST of the current iteration's body and jumps straight to the UPDATE step (followed by the CONDITION check again) -- unlike `break`, the loop itself does NOT end, only that ONE step is cut short. A common use is skipping invalid or unwanted elements in a collection while continuing to process the rest.

```
public class ContinueExample {
    public static void main(String[] args) {
        System.out.println("Odd numbers from 1 to 10:");
        for (int i = 1; i <= 10; i++) {
            // continue skips the REST of this iteration's body and jumps
            // straight to the update step -- the loop keeps running, unlike
            // break which would exit it entirely.
            if (i % 2 == 0) {
                continue;
            }
            System.out.print(i + " ");
        }
        System.out.println();

        // A common real use: skipping invalid entries while summing valid ones.
        int[] scores = {85, -1, 92, -1, 78, 60};
        int sum = 0;
        int validCount = 0;

        for (int i = 0; i < scores.length; i++) {
            if (scores[i] < 0) {
                continue; // -1 marks a missing score -- skip it
            }
            sum += scores[i];
            validCount++;
        }

        System.out.println("Valid scores: " + validCount + ", average: " + (sum /
validCount));
    }
}
```

Expected Output

```
Odd numbers from 1 to 10:
1 3 5 7 9
Valid scores: 4, average: 78
```

> `break` means "leave the loop entirely," while `continue` means "just skip this step, keep looping" -- confusing the two is a common conceptual mistake (see "Common Mistakes").

Infinite Loops

The `for (;;)` form, which skips all three parts (`init`, `condition`, `update`) entirely, creates a DELIBERATE infinite loop that never ends on its own -- only a `break` (or a `return`) inside the body can stop it. This is useful when you don't know in advance when to stop (e.g. keep asking until valid input arrives).

```
public class InfiniteLoopExample {
    public static void main(String[] args) {
        // Leaving out all three parts (for (;;)) creates a loop with no
        // built-in stopping condition -- it is intentionally infinite unless
        // something INSIDE the body breaks out of it.
        int attempts = 0;

        for (;;) {
            attempts++;
            System.out.println("Attempt " + attempts);

            if (attempts >= 3) {
                System.out.println("Condition met -- breaking out.");
                break;
            }
        }

        System.out.println("Done after " + attempts + " attempts.");

        // A common ACCIDENTAL infinite loop: forgetting the update step, or
        // writing an update that never makes the condition false. This is
        // shown as a comment, not run, because it would hang forever:
        //
        // for (int i = 0; i < 5; ) {
        //     System.out.println(i);
        //     // missing i++ -- the condition (i < 5) is NEVER updated
        // }
    }
}
```

Expected Output

```
Attempt 1
Attempt 2
Attempt 3
Condition met -- breaking out.
Done after 3 attempts.
```

An UNINTENTIONAL (i.e. buggy) infinite loop causes a real program to CRASH or HANG -- the most common cause is an update step that NEVER makes the condition `false` (e.g. forgetting to write `i++`, or setting up a condition that's accidentally always `true`). Whenever you write a `for`, make sure the update step will actually make the condition `false` at some point.

Multiple Variables in for

The initialization and update parts of a `for` can each hold MULTIPLE statements, separated by commas -- useful when two variables need to advance TOGETHER, such as scanning from both ends of an array toward the middle.

```
public class MultipleVariablesForExample {
    public static void main(String[] args) {
        // A for loop's init and update sections can each hold more than one
        // statement, separated by commas -- useful for two variables that
        // move together, like scanning from both ends toward the middle.
        int[] numbers = {10, 20, 30, 40, 50, 60, 70};

        for (int left = 0, right = numbers.length - 1; left < right; left++, right--) {
            System.out.println("left=" + left + " (" + numbers[left] + "), right=" + right
+ " (" + numbers[right] + ")");
        }

        // Checking whether an array is a palindrome uses the exact same pattern.
        int[] palindrome = {1, 2, 3, 2, 1};
        boolean isPalindrome = true;

        for (int left = 0, right = palindrome.length - 1; left < right; left++, right--) {
            if (palindrome[left] != palindrome[right]) {
                isPalindrome = false;
                break;
            }
        }

        System.out.println("Is palindrome: " + isPalindrome);
    }
}
```

Expected Output

```
left=0 (10), right=6 (70)
left=1 (20), right=5 (60)
left=2 (30), right=4 (50)
Is palindrome: true
```

Iterating Arrays with a Classic for Loop

Classic `for` also gives you the INDEX at every step of an array, not just the value (see "Basic Usage: Creation, Access, Default Values" in the "Arrays" lesson) -- this is needed whenever you also need the POSITION of a value, such as modifying an array IN PLACE. Why that matters -- and that a shorter alternative exists when you don't need the index -- will be covered in the upcoming "Enhanced for Loop" lesson.

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

        // The classic index-based for loop gives you the INDEX at every
        // step, not just the value -- useful for printing positions, or
        // comparing neighboring elements.
        for (int i = 0; i < scores.length; i++) {
            System.out.println("scores[" + i + "] = " + scores[i]);
        }

        // Because you have the index, you can also MODIFY the array in
        // place -- something a value-only loop cannot do (see "Enhanced for
        // Loop" for why).
        for (int i = 0; i < scores.length; i++) {
            scores[i] = scores[i] + 5; // curve every score by +5
        }

        System.out.print("Curved scores: ");
        for (int i = 0; i < scores.length; i++) {
            System.out.print(scores[i] + " ");
        }
        System.out.println();
    }
}
```

Expected Output

```
scores[0] = 70
scores[1] = 85
scores[2] = 92
scores[3] = 60
scores[4] = 78
Curved scores: 75 90 97 65 83
```

Best Practices

- **Keep the loop counter's scope as narrow as possible** -- declaring it in the `for`'s own initialization prevents it from being accidentally used outside the loop.
- **Only use `break` when an early exit is actually NEEDED** -- unnecessary `break` usage can make the flow harder to follow.
- **Consider inverting the condition instead of using `continue`, when possible** -- sometimes clearer for readability, though both approaches are valid.
- **If you write a `for (;;)`, make sure a `break` condition that will actually stop the loop exists.**
- **Always verify that the update step will make the condition `false` at some point** -- this is the most common cause of unintentional infinite loops.

Common Mistakes

- **Forgetting the update step (like `i++`), creating an unintentional infinite loop.** The condition never becomes `false`, and the program hangs.
- **Confusing `break` with `continue`.** `break` ends the loop entirely; `continue` only skips the current step -- these are very different behaviors.
- **Checking an array's bound with `<=` against `.length` and getting an `ArrayIndexOutOfBoundsException`.** Valid indexes run from `0` to `length - 1`, so the condition should be `i < array.length`, NOT `i <= array.length`.
- **Trying to use a counter declared inside a loop outside of it.** The counter only exists within the `for`'s own scope, and this results in a compile error.

Summary, Cheat Sheet, and Glossary

A `for` loop is a counter-driven repetition structure that combines INITIALIZATION/CONDITION/UPDATE in a single line. `break` terminates the loop entirely; `continue` only skips the current step. A `for (;;)` that skips all three parts creates a deliberate infinite loop, stoppable only by a `break` inside it. The initialization/update parts can each hold multiple statements separated by commas. Classic `for` gives you both the VALUE and the INDEX while iterating an array.

Quick reference:

```
for (int i = 0; i < 5; i++) {
    // ...
}

for (int i = 0; ; i++) {
    if (condition) break;
}

for (int i = 0; i < array.length; i++) {
    if (array[i] < 0) continue;
    // ...
}

for (int i = 0, j = array.length - 1; i < j; i++, j--) {
    // ...
}
```

Glossary

Initialization — The first part of a `for`, which runs only once, before the loop starts.

Condition — The `boolean` expression checked before every iteration, which ends the loop once it's `false`.

Update — The part that runs at the end of every iteration, usually advancing the counter.

break — The keyword that immediately terminates a loop, regardless of the condition.

continue — The keyword that skips the rest of the current iteration and jumps straight to the update step.