Nested Loops

The sixth and final topic in the Control Flow category: basic nested for loops, 2D array traversal, how unlabeled break/continue only affect the innermost loop, labeled break /continue, and O(n²) performance awareness.

Beginner 22 min

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

---

Nested Loops

Every loop we've seen so far (`for` , `while` , `do-while`) ran on its own, in a flat sequence. But many real problems aren't one-dimensional -- every cell of a table, every pixel of an image, every possible pairing between two lists. To handle that, we put one loop inside another loop's body: **nested loops**. This last topic in the Control Flow category brings together everything learned in the previous 5 topics (`for` , `break` , `continue` , `while`) and covers the new behavior that shows up once they're nested.

What Are Nested Loops?

A nested loop is a loop (`for` , `while` , or `do-while` -- it doesn't matter which) whose body runs another loop inside it:

```
for (int i = 0; i < 3; i++) {      // outer loop
    for (int j = 0; j < 3; j++) {  // inner loop
        // ...
    }
}
```

Every time the outer loop advances one step, the inner loop runs **completely, start to finish**. So if the outer loop runs 3 times and the inner loop runs 3 times on each of those, the body runs a total of 3 × 3 = 9 times -- this multiplicative relationship is the defining trait of nested loops.

Why Does It Exist?

A single loop is enough to walk linear (1-dimensional) data -- an array, a list. But a lot of real-world data and problems are 2-dimensional (or more): a chessboard, a spreadsheet, an

image's width×height pixels, or "every student against every course" -- all combinations between two separate collections. There's no way to express that with a single loop; each dimension needs its own loop level. Nested loops let us express this multi-dimensional traversal naturally, using the language's own basic building blocks, without learning any new syntax.

History

Nested loops aren't a separate language feature -- they're simply the natural consequence of being able to write one loop inside another loop's body, so they've existed since Java's very first day (1996). No special syntax or keyword is required. What actually is new here is how `break` / `continue` behave once more than one loop level is in play, and the mechanism Java provides to make that explicit: **labeled break/continue** -- itself a Java feature that has been present from the start and that C does not have.

Basic Nested for Loop

In its plainest form, an inner loop runs completely on every step of an outer loop:

```
public class NestedForBasicsExample {
    public static void main(String[] args) {
        // The outer loop runs 3 times; for EACH of those, the inner loop runs
        // 3 full times -- so the body executes 3 * 3 = 9 times in total.
        int totalIterations = 0;

        for (int row = 1; row <= 3; row++) {
            for (int col = 1; col <= 3; col++) {
                System.out.print("(" + row + "," + col + ") ");
                totalIterations++;
            }
            System.out.println();
        }

        System.out.println("Total iterations: " + totalIterations);
    }
}
```

Expected Output

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1) (3,2) (3,3)
Total iterations: 9
```

Notice the output: the outer loop variable (`row`) stays fixed while the inner loop variable (`col`) runs through all its values from 1 to 3 -- then the outer loop advances one step and the inner loop starts over from the beginning.

Working with 2D Arrays

One of the most natural uses of nested loops is 2D arrays (see the "Multi-Dimensional Arrays" section of the "Arrays" lesson) -- the outer loop walks the rows, the inner loop walks that row's columns:

```
public class TwoDArrayExample {
    public static void main(String[] args) {
        // A 2D array is really an array of arrays -- matrix[row] is itself an
        // int[]. Nested loops are the natural way to visit every cell: the
        // outer loop walks the rows, the inner loop walks the columns of
        // whichever row we're currently on.
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int sum = 0;
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++) {
                System.out.print(matrix[row][col] + " ");
                sum += matrix[row][col];
            }
            System.out.println();
        }

        System.out.println("Sum of all elements: " + sum);
    }
}
```

Expected Output

```
1 2 3
4 5 6
7 8 9
Sum of all elements: 45
```

Notice the use of `matrix[row].length` rather than `matrix.length` -- this stays correct even for a "jagged" array, where rows can have different lengths.

break and continue in Nested Loops

An unlabeled `break` (see the "Exiting a Loop with break" section of the "for Loop" lesson) and `continue` (see the "Skipping an Iteration with continue" section of the "for Loop" lesson) only affect **the innermost loop they're written in** -- neither has any effect on the outer loop:

```
public class BreakContinueInNestedLoopExample {
    public static void main(String[] args) {
        // An unlabeled break only exits the loop it's directly written
        // inside -- the INNER loop here. The outer loop is completely
        // unaffected and keeps running its remaining iterations.
        System.out.println("break in a nested loop:");
        for (int row = 1; row <= 3; row++) {
            System.out.println("Starting row " + row);
            for (int col = 1; col <= 5; col++) {
                if (col == 3) {
                    System.out.println("  breaking inner loop at col = " + col);
                    break;
                }
                System.out.println("  col = " + col);
            }
            System.out.println("Finished row " + row);
        }

        // Just like break, an unlabeled continue only affects the loop it's
        // directly written inside. It skips the rest of the current inner
        // iteration and moves on -- the outer loop's iteration count is
        // untouched.
        System.out.println("continue in a nested loop:");
        for (int row = 1; row <= 2; row++) {
            for (int col = 1; col <= 4; col++) {
                if (col % 2 == 0) {
                    continue;
                }
                System.out.println("row=" + row + ", col=" + col);
            }
        }
    }
}
```

Expected Output

```
break in a nested loop:
Starting row 1
  col = 1
  col = 2
  breaking inner loop at col = 3
Finished row 1
Starting row 2
  col = 1
  col = 2
  breaking inner loop at col = 3
Finished row 2
Starting row 3
  col = 1
  col = 2
  breaking inner loop at col = 3
Finished row 3
continue in a nested loop:
row=1, col=1
row=1, col=3
row=2, col=1
row=2, col=3
```

As the output shows, `break` breaks the inner loop at `col = 3` on every row, but the outer loop still completes all 3 rows normally; `continue` skips even numbers, but that only affects the inner loop -- the outer loop (`row`) advances with no steps skipped there. Both effects are scoped to the loop level they're written in.

Labeled break and continue

Sometimes you need to affect the outer loop directly from inside the inner loop -- for example, stopping an entire search as soon as a match is found. For that, a **label** (an identifier followed by `:`) is placed right before a loop, and `break` / `continue` is used together with that label:

```
public class LabeledBreakContinueExample {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // A label (an identifier followed by ':') placed right before a loop
        // lets break/continue target THAT loop specifically, even from
        // inside a nested loop. Without the label, break/continue could only
        // ever affect the innermost loop they're written in.
        int target = 5;
        boolean found = false;

        searchLoop:
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++) {
                if (matrix[row][col] == target) {
                    System.out.println("Found " + target + " at [" + row + "][" + col +
"]");
                    found = true;
                    break searchLoop; // exits BOTH loops immediately
                }
            }
        }
        if (!found) {
            System.out.println(target + " not found");
        }

        // Labeled continue: skip straight to the outer loop's next
        // iteration as soon as any negative number is found in a row.
        int[][] withNegatives = {
            {1, 2, -3},
            {4, -5, 6},
            {7, 8, 9}
        };

        rowLoop:
        for (int row = 0; row < withNegatives.length; row++) {
            for (int col = 0; col < withNegatives[row].length; col++) {
                if (withNegatives[row][col] < 0) {
                    System.out.println("Row " + row + " has a negative value, skipping it");
                    continue rowLoop; // jumps straight to row + 1
                }
            }
            System.out.println("Row " + row + " is all non-negative");
        }
    }
}
```

Expected Output

```
Found 5 at [1][1]
Row 0 has a negative value, skipping it
Row 1 has a negative value, skipping it
Row 2 is all non-negative
```

`break searchLoop;` ends both the inner and outer loop at once -- an unlabeled `break` would only have stopped the inner loop, letting the outer loop keep going. `continue rowLoop;` jumps straight to the outer loop's next iteration, skipping the rest of that row's columns entirely.

Performance: Why O(n²)?

Nested loops have a cost: every extra loop level multiplies the work. A single loop over `n` elements takes `n` steps (`O(n)`); nest a second loop of the same size inside it, and the body now runs `n × n` times (`O(n²)`):

```
public class NestedLoopPerformanceExample {
    public static void main(String[] args) {
        // Every extra level of nesting multiplies the work. One loop over n
        // elements is O(n). Nest a second loop of the same size inside it,
        // and the body now runs n * n times -- O(n^2). This example doesn't
        // measure wall-clock time (unreliable on a shared machine); instead
        // it counts actual operations to make the growth concrete.
        int[] sizes = {10, 100, 1000};

        for (int n : sizes) {
            long operations = 0;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    operations++;
                }
            }
            System.out.println("n = " + n + " -> " + operations + " operations (n^2 = " +
((long) n * n) + ")");
        }
    }
}
```

Expected Output

```
n = 10 -> 100 operations (n^2 = 100)
n = 100 -> 10000 operations (n^2 = 10000)
n = 1000 -> 1000000 operations (n^2 = 1000000)
```

When `n` grows from 10 to 100 (10x), the operation count grows from 100 to 10,000 (100x) -- not linear growth, but quadratic. Working with large data sets means keeping this cost in mind whenever you reach for a nested loop.

Worked Example: Printing a Pyramid with Stars

Let's bring everything together in one classic exercise: printing a centered pyramid of stars. Two SEPARATE loops -- one for spaces, one for stars -- run one after another on every step of the outer loop. **This is still a single level of nesting** -- the spaces loop and the stars loop aren't inside each other, they're each just inside the outer loop, running in sequence:

```
public class PyramidPrintingExample {
    /**
     * Prints a pyramid of stars, centered with leading spaces.
     * For rows = 4:
     *    *
     *   ***
     *  *****
     * *******
     *
     * This is still just ONE level of nesting -- the "spaces" loop and the
     * "stars" loop each sit one level inside the outer row loop and run one
     * after another, not inside each other.
     */
    public static void main(String[] args) {
        int rows = 4;

        for (int i = 0; i < rows; i++) {
            // Row i needs (rows - i - 1) leading spaces...
            for (int j = 0; j < rows - i - 1; j++) {
                System.out.print(" ");
            }
            // ...followed by (2 * i + 1) stars.
            for (int k = 0; k < 2 * i + 1; k++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Expected Output

```
   *
  ***
 *****
*******
```

Each row's math depends on the row index `i` (starting from 0, for `rows = 4`): `i=0` needs 3 spaces + 1 star, `i=1` needs 2 spaces + 3 stars, `i=2` needs 1 space + 5 stars, `i=3` needs 0 spaces + 7 stars.

As `i` increases, the space count shrinks while the star count grows -- together these two produce the pyramid's centered shape. This is a good example of nested loops being used not just to "repeat something," but to CALCULATE one loop's bounds from another loop's current value.

Best Practices

Reserve nested loops for cases that genuinely need a multi-dimensional structure (a 2D array, all pairwise combinations) -- adding an extra loop level where it isn't needed both slows the code down and makes it harder to read. Use labeled break/continue only when you actually need to affect the outer loop; if affecting just the inner loop is enough, prefer the unlabeled form -- unnecessary labels add complexity. Give loop variables meaningful names (`row`, `col` instead of `i`, `j`) -- with more than two nested levels, `i`/`j`/`k` confusion becomes a real readability problem.

Common Mistakes

The most common mistake is assuming an unlabeled `break`/`continue` will also affect the outer loop -- it only ever affects the innermost loop; affecting the outer loop always requires a label. A second common mistake is mixing up inner and outer loop variables -- for example reusing `i` in both loops and accidentally updating the outer loop's variable from inside the inner loop (or vice versa). A third mistake is ignoring the cost of nested loops on large data sets (e.g. `n = 100,000`) and ending up with an `O(n²)` algorithm -- this can noticeably slow an application down as `n` grows.

Summary, Cheat Sheet, and Glossary

Nested loops mean writing one loop inside another loop's body -- the inner loop runs completely on every step of the outer loop (a multiplicative relationship). An unlabeled `break`/`continue` only affects the innermost loop; affecting the outer loop requires a labeled `break label;` / `continue label;`. Two nested loops over `n` elements cost `O(n²)`.

```
// Basic nested loop
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // runs n * n times
    }
}

// Labeled break: ends the outer loop too
outer:
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (condition) {
            break outer;
        }
    }
}
```

Glossary

- **Nested loop:** A loop that runs inside another loop's body.

- **Outer loop / Inner loop:** In two nested loops, the loop whose body contains the other is the outer loop; the loop inside it is the inner loop.
- **Label:** An identifier (`name:`) placed right before a loop that lets `break` / `continue` target that specific loop level.
- **O(n²) (quadratic time complexity):** A situation where the number of operations grows with the square of the input size `n` -- typically caused by two nested loops running over data of the same size.
- **Pattern printing:** A classic nested-loop exercise where a loop's bounds (how many times it runs) are calculated from the outer loop's current value.