while & do-while Loops

The fifth topic in the Control Flow category: basic `while` syntax, `do-while`'s run-at-least-once body, while vs do-while comparison, break/continue, a Scanner-based input validation loop, and a worked example: the Number Guessing Game.

Beginner 20 min

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

---

while & do-while Loops

The `for` loop was ideal for "I know how many times this will run" situations. `while` and `do-while` answer the opposite question: *"I know when to stop, but I don't know how many steps it will take."* Asking a user until valid input arrives, reading until the end of a file, retrying until a connection succeeds -- these are all "keep going while a condition holds" logic, not counter logic.

What Is a while Loop?

A `while` loop repeatedly runs a block of code as long as a condition is true:

```
while (condition) {
    // runs as long as condition is true
}
```

The condition is checked **before** every iteration. If it's false from the very start, the loop body never runs at all -- the same behavior as a `for` loop, except `while` has no built-in counter/increment requirement; the condition can be any arbitrary logic.

Why Does It Exist?

A `for` loop bundles "initialization; condition; increment" into one line because those three usually change together (a counter variable). But many real-world repetitions aren't about a counter at all -- they're about a **state**: keep asking until the user enters something valid, keep retrying until a connection succeeds, keep processing until a queue is empty. Forcing these into a `for` loop would be awkward (writing `for (; condition; )` and leaving the init /increment parts empty). `while` expresses this "I only have a condition" case directly and readably. `do-while` adds one more capability on top: the "must run at least once" need --

showing a menu at least once, or reading a first piece of user input before you can even evaluate whether it's valid.

History

`while` and `do-while` have been in Java since day one (JDK 1.0, 1996) -- inherited directly from C and C++, two of the fundamental building blocks common to C-style languages. Like the `for` loop, they predate the Enhanced for Loop introduced in Java 5 by nearly a decade, and their syntax hasn't changed since.

Basic while Syntax

In its simplest form, `while` runs its body as long as the condition is true:

```
public class WhileBasicsExample {
    public static void main(String[] args) {
        int count = 1;

        // The condition is checked BEFORE each iteration -- if it's false
        // from the start, the body never runs at all.
        while (count <= 5) {
            System.out.println("count = " + count);
            count++;
        }

        int sum = 0;
        int n = 1;
        while (sum < 20) {
            sum += n;
            n++;
        }
        System.out.println("Stopped once sum reached " + sum + " (n was " + n + ")");
    }
}
```

Expected Output

```
count = 1
count = 2
count = 3
count = 4
count = 5
Stopped once sum reached 21 (n was 7)
```

Notice that something inside the body must affect the condition (`count++` or `sum += n` in the example above) -- otherwise the loop never ends (see "Common Mistakes").

do-while: A Loop That Runs at Least Once

`do-while` checks its condition **after** the body:

```
do {
    // runs at least once
} while (condition);
```

Unlike `while`, this means the body runs at least once even if the condition is already false at the very start:

```
public class DoWhileBasicsExample {
    public static void main(String[] args) {
        int count = 10;

        // do-while checks the condition AFTER the body -- so the body runs
        // at least ONCE even though "count <= 5" is already false when the
        // loop starts.
        do {
            System.out.println("count = " + count);
            count++;
        } while (count <= 5);

        System.out.println("Loop ran exactly once, count is now " + count);
    }
}
```

Expected Output

```
count = 10
Loop ran exactly once, count is now 11
```

This behavior fits a specific pattern well: "do something first, then decide whether to keep going" -- for example, reading user input. You can't check whether input is valid before you've actually read it.

while vs do-while: Which One, When

The only difference between the two is **when** the condition is checked -- but that difference determines whether the body runs zero times or one time when the condition is false from the start:

```
public class WhileVsDoWhileExample {
    public static void main(String[] args) {
        int x = 100;

        System.out.println("while (condition is false from the start):");
        int whileRuns = 0;
        while (x < 10) {
            whileRuns++;
        }
        System.out.println("Body ran " + whileRuns + " times.");

        System.out.println("do-while (same condition, false from the start):");
        int doWhileRuns = 0;
        do {
            doWhileRuns++;
        } while (x < 10);
        System.out.println("Body ran " + doWhileRuns + " times.");
    }
}
```

Expected Output

```
while (condition is false from the start):
Body ran 0 times.
do-while (same condition, false from the start):
Body ran 1 times.
```

General rule: if you know the body must run at least once (showing a menu, prompting for input), use `do-while`. Otherwise -- if it's entirely possible the body should never run -- use `while`.

break and continue with while

`break` and `continue` work exactly the same way here as they do in a `for` loop (see the "Exiting a Loop with break" and "Skipping an Iteration with continue" sections of the "for Loop" lesson) -- only the loop's own header syntax differs:

```
public class BreakContinueInWhileExample {
    public static void main(String[] args) {
        // break and continue work exactly the same way in while/do-while as
        // they do in for (see the "for Loop" lesson) -- only the loop's own
        // header differs.
        int i = 0;
        System.out.println("break in a while loop:");
        while (true) {
            i++;
            if (i > 3) {
                break;
            }
            System.out.println("i = " + i);
        }

        System.out.println("continue in a while loop:");
        int j = 0;
        while (j < 10) {
            j++;
            if (j % 2 == 0) {
                continue;
            }
            System.out.println("odd j = " + j);
        }
    }
}
```

Expected Output

```
break in a while loop:
i = 1
i = 2
i = 3
continue in a while loop:
odd j = 1
odd j = 3
odd j = 5
odd j = 7
odd j = 9
```

`break` ends the loop entirely; `continue` only skips the rest of the current iteration and jumps back to the condition check.

A Validation Loop with User Input (Using Scanner)

One of the most natural uses of `do-while` is asking a user repeatedly until they provide a valid value (see the "Basic Usage: Reading Tokens" section of the "Scanner" lesson):

```
import java.util.Scanner;

public class InputValidationLoopExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int age;

        // do-while is the natural fit here: you must ask AT LEAST once,
        // and keep re-asking only if the input turns out to be invalid.
        do {
            System.out.print("Enter your age (1-120): ");
            age = scanner.nextInt();

            if (age < 1 || age > 120) {
                System.out.println("Invalid age, try again.");
            }
        } while (age < 1 || age > 120);

        System.out.println("Valid age received: " + age);
        scanner.close();
    }
}
```

`do-while` is the right choice here for an obvious reason: you have to ask the user at least once -- you can't know whether the input is valid before you've read it.

Worked Example: The Number Guessing Game

Let's combine everything covered so far -- `do-while`, conditional validation, and reading input with `Scanner` -- into one small but complete program: the classic number guessing game:

```
import java.util.Scanner;

public class NumberGuessingGameExample {
    public static void main(String[] args) {
        // In a real game this would be random (new Random().nextInt(100) + 1) --
        // fixed here so the walkthrough below is reproducible.
        int target = 42;
        Scanner scanner = new Scanner(System.in);
        int guess;
        int attempts = 0;

        System.out.println("Guess a number between 1 and 100.");

        do {
            System.out.print("Your guess: ");
            guess = scanner.nextInt();
            attempts++;

            if (guess < target) {
                System.out.println("Too low!");
            } else if (guess > target) {
                System.out.println("Too high!");
            } else {
                System.out.println("Correct! You guessed it in " + attempts + " attempts.");
            }
        } while (guess != target);

        scanner.close();
    }
}
```

The program keeps asking until the user guesses the right number -- another textbook case of "I don't know how many steps this takes, but I know exactly when it stops," which is precisely why `do-while` exists.

Best Practices

Always make sure something in the loop body updates the variable the condition depends on -- forgetting to do so is the single most common cause of an accidental infinite loop. Keep the condition itself simple and readable; if it needs to be complex, consider capturing it in a meaningfully-named `boolean` variable instead (something like `while (hasMoreRecords)`). Prefer `do-while` over `while` whenever the body must run at least once -- it expresses the "try it once up front" intent directly in the code.

Common Mistakes

The most common mistake is forgetting to update the variable the condition depends on, accidentally creating an infinite loop. A second common mistake is picking the wrong loop for the situation -- especially using `while` for an "must run at least once" case like input validation and then manually duplicating the first check outside the loop (leading to

unnecessary code duplication). A third mistake is forgetting the semicolon at the end of a `do-while`'s closing line -- `} while (condition)` requires a trailing `;`, and omitting it is a compile error.

Summary, Cheat Sheet, and Glossary

`while` checks its condition **before** the body; the body might never run at all. `do-while` checks its condition **after** the body; the body always runs at least once. `break` ends the loop entirely, `continue` only skips the current iteration -- both behave exactly as they do in a `for` loop. For "try it at least once" scenarios like input validation, `do-while` is the natural choice.

```
// while: condition checked first
while (condition) {
    // ...
}

// do-while: condition checked last, body runs at least once
do {
    // ...
} while (condition);
```

Glossary

- **while loop:** A loop construct that checks its condition before every iteration and runs as long as the condition is true.
- **do-while loop:** A loop construct that checks its condition after every iteration, so its body always runs at least once.
- **Input validation loop:** A loop pattern that keeps asking the user until a valid value is provided.
- **Infinite loop:** A loop whose condition never becomes false, so it never ends -- usually caused by forgetting to update the variable the condition depends on.