if / else

The first topic in the new Control Flow category: the basics of `if` / `else`, `else if` chains, nested conditions, the six comparison operators (`==` / `!=` / `<` / `>` / `<=` / `>=`) with decimal number and `String` comparison pitfalls, the `&&` / `||` / `!` logical operators and short-circuit evaluation, and the ternary operator.

Beginner 20 min

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

---

if / else

The first topic in the Java course's new "Control Flow" category: if / else -- the most fundamental decision-making structure that lets a program run different code paths depending on a condition. `if` runs a block of code when a condition is `true`; `else` defines an alternative block that runs when that condition is `false`. You may have already run into this construct implicitly in examples from Java Basics topics like `String` / `Arrays` / `Scanner` -- this lesson covers it systematically, from the ground up, for the first time.

What Is if / else?

An `if` statement evaluates a `boolean` condition in parentheses -- if the condition is `true`, the block that follows runs immediately; if it's `false`, the block is skipped. The `else` block is optional and runs only when the `if` condition is `false`. `else if` combines `else` with a new `if` to check several conditions in sequence (see "else if Chains"). Only a `boolean` expression is accepted as a condition -- unlike C/C++, Java never implicitly converts a numeric value like `0` / `1` to `boolean`; code like `if (number)` simply DOES NOT COMPILE.

Why Does It Exist?

If a program always did exactly the same thing, there would be no need for conditional logic -- but every real program needs branching: is a user's input valid, is there enough stock, does an age cross a threshold? `if` / `else` is the most direct way to express these binary (or, with `else if`, multi-way) decisions in a readable, sequential form. Nearly every algorithm -- validation, filtering, error handling -- has an `if` / `else` somewhere at its core.

History

`if` / `else` became standard in the late 1960s as part of the structured programming movement, in languages like ALGOL and Pascal, and carried over into C (1972) -- Java (1995) inherited its syntax almost verbatim from C. Java's one major departure is requiring the condition to be strictly a `boolean`: the common C idiom `if (x)` (true when `x` is nonzero) was deliberately turned into a COMPILE ERROR in Java -- this rules out, at the language level, a whole class of bugs common in C where `=` (assignment) is typed instead of `==` (comparison).

Basic if / else Usage

In its simplest form, `if` can stand alone, be paired with `else`, or be skipped entirely. When either block is a single statement, curly braces (`{}`) are technically optional -- but this is a real source of bugs (see "Common Mistakes").

```
public class IfElseBasicsExample {
    public static void main(String[] args) {
        int number = 7;

        if (number % 2 == 0) {
            System.out.println(number + " is even.");
        } else {
            System.out.println(number + " is odd.");
        }

        int age = 16;

        if (age >= 18) {
            System.out.println("Eligible to vote.");
        } else {
            System.out.println("Not eligible to vote yet.");
        }

        // A single-statement if/else WITHOUT braces -- legal, but risky.
        // See "Common Mistakes" for why braces should be used anyway.
        int score = 42;
        if (score > 50)
            System.out.println("Passed.");
        else
            System.out.println("Failed.");
    }
}
```

Expected Output

```
7 is odd.
Not eligible to vote yet.
Failed.
```

> An `if` / `else` written without braces only covers the SINGLE statement that immediately follows it -- adding a second line later leaves that line outside the condition even though indentation still makes it LOOK like it's inside. That's why using braces even for one-line blocks is a strongly recommended practice.

else if Chains

When more than two branches are needed, `else if` combines `else` with a new `if` to check conditions in order, TOP TO BOTTOM -- the block for the first condition that evaluates to `true` runs, and every remaining condition is SKIPPED (even if it would also have been `true`). This means the order of conditions matters: NARROWER/more specific conditions generally need to come before BROADER ones.

```
public class ElseIfChainExample {
    public static void main(String[] args) {
        printGrade(95);
        printGrade(82);
        printGrade(71);
        printGrade(65);
        printGrade(40);
    }

    // Note: repeating this for a list of scores with a loop is exactly what
    // you'll learn in the "for Loop" lesson -- this topic keeps it loop-free.
    private static void printGrade(int score) {
        String grade;

        if (score >= 90) {
            grade = "A";
        } else if (score >= 80) {
            grade = "B";
        } else if (score >= 70) {
            grade = "C";
        } else if (score >= 60) {
            grade = "D";
        } else {
            grade = "F";
        }

        System.out.println("Score " + score + " -> Grade " + grade);
    }
}
```

Expected Output

```
Score 95 -> Grade A
Score 82 -> Grade B
Score 71 -> Grade C
Score 65 -> Grade D
Score 40 -> Grade F
```

Nested Conditions

Writing one `if` inside the body of another -- a nested condition -- is used when two conditions need to be evaluated as DEPENDENT on each other, i.e. the second condition only makes sense once the first one holds. This is often equivalent to a single `if` combined with `&&` (see "Logical Operators and Short-Circuit Evaluation"), but nesting reads more clearly when each condition needs its OWN separate `else` branch (such as a different error message).

```
public class NestedConditionsExample {
    public static void main(String[] args) {
        checkLogin("alice", "correct-password");
        checkLogin("unknown-user", "whatever");
        checkLogin("alice", "wrong-password");
    }

    private static void checkLogin(String username, String password) {
        String storedUsername = "alice";
        String storedPassword = "correct-password";

        // An if INSIDE another if's body -- a nested condition. The inner check
        // only runs when the outer one is already true.
        if (username.equals(storedUsername)) {
            if (password.equals(storedPassword)) {
                System.out.println("Login successful for " + username);
            } else {
                System.out.println("Wrong password for " + username);
            }
        } else {
            System.out.println("No such user: " + username);
        }
    }
}
```

Expected Output

```
Login successful for alice
No such user: unknown-user
Wrong password for alice
```

> More than three or four levels of nested `if` blocks usually hurts readability badly (known as the "arrow anti-pattern") -- at that point, flattening the conditions with an early `return`, or combining them into a single `if` with `&&`, is generally preferred.

Comparison Operators

Java provides six comparison operators: `==` (equal to), `!=` (not equal to), `<`, `>`, `<=`, `>=`. All of them produce a `boolean` result and can be used directly as an `if` condition. `==` compares the VALUE for primitive types, but compares the REFERENCE for objects (e.g. `String`) -- see "The String Pool and == vs equals()" in the String lesson.

```
public class ComparisonOperatorsExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;

        System.out.println("a == b: " + (a == b));
        System.out.println("a != b: " + (a != b));
        System.out.println("a < b: " + (a < b));
        System.out.println("a > b: " + (a > b));
        System.out.println("a <= b: " + (a <= b));
        System.out.println("a >= b: " + (a >= b));

        // Comparing floating-point numbers directly with == is risky -- binary
        // floating point cannot represent 0.1 or 0.2 exactly, so their sum is
        // not exactly 0.3.
        double sum = 0.1 + 0.2;
        System.out.println("0.1 + 0.2 == 0.3: " + (sum == 0.3));
        System.out.println("Actual value: " + sum);

        // Comparing String OBJECTS with == compares references, not content --
        // see the "String Pool ve == vs equals()" section in the String lesson.
        String literalOne = "hello";
        String literalTwo = "hello";
        String constructed = new String("hello");

        System.out.println("literalOne == literalTwo: " + (literalOne == literalTwo));
        System.out.println("literalOne == constructed: " + (literalOne == constructed));
        System.out.println("literalOne.equals(constructed): " + literalOne.equals
(constructed));
    }
}
```

Expected Output

```
a == b: false
a != b: true
a < b: true
a > b: false
a <= b: true
a >= b: false
0.1 + 0.2 == 0.3: false
Actual value: 0.30000000000000004
literalOne == literalTwo: true
literalOne == constructed: false
literalOne.equals(constructed): true
```

Comparing decimal numbers (`double` / `float`) with `==` is risky -- binary floating-point representation cannot store decimals like `0.1` EXACTLY, which is why `0.1 + 0.2 == 0.3` returns `false` . For approximate comparisons, check whether the difference between two values is smaller than a small threshold (epsilon).

Logical Operators and Short-Circuit Evaluation

`&&` (AND), `||` (OR), and `!` (NOT) let you combine multiple `boolean` expressions. Both `&&` and `||` are evaluated with SHORT-CIRCUIT logic: if the left side of `&&` is already `false`, the right side is NEVER EVALUATED (since the result is already known to be `false`); if the left side of `||` is already `true`, the right side is NEVER EVALUATED. This isn't just a performance optimization -- it's safe to rely on it, for example calling a method on the right side only after a `null` check has already passed on the left.

```
public class LogicalOperatorsExample {
    public static void main(String[] args) {
        int age = 25;
        boolean hasLicense = true;

        System.out.println("Can drive: " + (age >= 18 && hasLicense));

        boolean isWeekend = false;
        boolean isHoliday = true;

        System.out.println("Day off: " + (isWeekend || isHoliday));
        System.out.println("Not a day off: " + !(isWeekend || isHoliday));

        // Short-circuit evaluation: && skips the right-hand side once the
        // left-hand side is already false; || skips it once the left-hand
        // side is already true.
        System.out.println("Short-circuit && demo:");
        System.out.println(checkFirst() && checkSecond());

        System.out.println("Short-circuit || demo:");
        System.out.println(checkThird() || checkFourth());
    }

    private static boolean checkFirst() {
        System.out.println("  checkFirst() called");
        return false;
    }

    private static boolean checkSecond() {
        System.out.println("  checkSecond() called -- should NOT print, short-circuited");
        return true;
    }

    private static boolean checkThird() {
        System.out.println("  checkThird() called");
        return true;
    }

    private static boolean checkFourth() {
        System.out.println("  checkFourth() called -- should NOT print, short-circuited");
        return false;
    }
}
```

Expected Output

```
Can drive: true
Day off: true
Not a day off: false
Short-circuit && demo:
  checkFirst() called
false
Short-circuit || demo:
  checkThird() called
true
```

The Ternary Operator

The ternary operator, written `condition ? value1 : value2`, is a short alternative to a simple `if` / `else` when it needs to PRODUCE A VALUE (such as an assignment) -- `if` / `else` is a STATEMENT, not an expression, while the ternary operator is an EXPRESSION that evaluates directly to a value.

```
public class TernaryOperatorExample {
    public static void main(String[] args) {
        int age = 20;
        String status = (age >= 18) ? "adult" : "minor";
        System.out.println("Status: " + status);

        int a = 15;
        int b = 42;
        int max = (a > b) ? a : b;
        System.out.println("Max: " + max);

        // Ternary expressions can be nested, but this quickly hurts readability --
        // prefer an if/else-if chain (see "else if Chains") once you need more
        // than two outcomes.
        int score = 72;
        String grade = (score >= 90) ? "A" : (score >= 80) ? "B" : (score >= 70) ? "C" :
"F";
        System.out.println("Nested ternary grade: " + grade);
    }
}
```

Expected Output

```
Status: adult
Max: 42
Nested ternary grade: C
```

Nesting ternary operators (`a ? b : c ? d : e`) technically works but quickly becomes unreadable -- once more than two outcomes are needed, an `else if` chain is almost always clearer.

Best Practices

- **Use braces even for one-line `if` / `else` blocks** -- it prevents a line added later from accidentally ending up outside the condition.
- **Order conditions in `else if` chains from narrowest/most specific to broadest** -- the first matching condition wins, so a misordered broad condition can "shadow" a more specific one.
- **Prefer an early `return` or a single condition combined with `&&` over more than three or four levels of nested `if`.**
- **Compare decimal numbers with an epsilon threshold, not `==`.**
- **Reserve the ternary operator for simple, single-value choices** -- nested ternaries hurt readability.

Common Mistakes

- **Accidentally adding a second line after a brace-less `if`.** Indentation makes it look like the line belongs to the `if`, but the compiler disagrees -- the line always runs, REGARDLESS of the condition.
- **Ordering an `else if` chain wrong, putting a broader condition BEFORE a more specific one.** The specific condition then never runs, because the broader one before it is already `true`.
- **Comparing decimal numbers with `==` and getting unexpected `false` results.** `0.1 + 0.2 == 0.3` is the classic example -- it comes from binary floating-point representation.
- **Comparing two `String`s with `==`.** This compares REFERENCE, not CONTENT -- `.equals()` should always be used instead.

Summary, Cheat Sheet, and Glossary

`if` / `else` is the most fundamental branching structure, running different code paths based on a `boolean` condition. `else if` checks multiple conditions in order (the first `true` one wins). Nested conditions are used for dependent checks but can quickly hurt readability. Comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) produce a `boolean`; `==` compares REFERENCE for objects. Logical operators (`&&`, `||`, `!`) are evaluated with short-circuit logic. The ternary operator (`? :`) is a short form of a simple `if` / `else` that produces a VALUE.

Quick reference:

```
if (condition) {
    // ...
} else if (otherCondition) {
    // ...
} else {
    // ...
}

boolean result = (a > b) && (c < d);              // logical AND, short-circuited
String label = (age >= 18) ? "adult" : "minor";   // ternary operator
```

Glossary

Condition — The `boolean` expression an `if` / `else if` evaluates, producing `true` or `false`.

Branching — A program running different code paths depending on a condition.

Short-Circuit Evaluation — Skipping the right-hand side of `&&` / `||` once the result is already determined by the left-hand side.

Ternary Operator — A conditional expression of the form `condition ? value1 : value2` that evaluates directly to a value.