Module 04 — Conditional Statements

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

🎯 Learning Objectives

By the end of this module, you will be able to:

1. Write boolean expressions using relational operators and boolean variables.
2. Use `if`, `if-else`, and `if-else-if` statements to control program flow.
3. Combine conditions with logical operators `&&`, `||`, and `!`.
4. Explain why `==` does not work reliably for String comparison and use `.equals()` correctly.
5. Write concise expressions using the ternary operator.
6. Build a `switch` statement with proper `case`, `break`, and `default` clauses.
7. Identify and fix the most common mistakes beginners make with conditionals.

📖 Key Terms

Boolean expression · **relational operator** · **if statement** · **if-else statement** · **if-else-if chain** · **nested if** · **logical operator** · **short-circuit evaluation** · **ternary operator** · **switch statement** · **fall-through** · **.equals()** · **dangling else**

🧠 Think of It This Way

Your program is a GPS. Every time it reaches a fork in the road, it has to make a decision: *"Turn left or keep going straight?"* Conditional statements are those decision points. Without them, your program just blasts through every intersection the same way — no matter what the user does or what values it encounters.

1. Boolean Expressions

Every decision in Java starts with a question that has exactly one answer: yes or no. That question is called a **boolean expression**. It evaluates to either `true` or `false` — nothing else.

Relational Operators

You build boolean expressions by comparing values with **relational operators**:

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	`true`
`!=`	Not equal to	`5 != 3`	`true`
`>`	Greater than	`10 > 7`	`true`
`<`	Less than	`3 < 1`	`false`
`>=`	Greater than or equal to	`6 >= 6`	`true`
`<=`	Less than or equal to	`4 <= 9`	`true`

Boolean Variables

Java has a primitive type called `boolean` that stores exactly two values: `true` or `false`. You can assign a literal or the result of a relational expression directly to it:

```
boolean isRaining = true;
boolean hasUmbrella = false;
int score = 85;
boolean passed = score >= 70;  // evaluates to true

System.out.println(passed);      // true
System.out.println(isRaining);   // true
System.out.println(5 > 3);       // true
System.out.println(2 == 9);      // false
```

> 💡 **Tip: Name Booleans Like Questions**
>
> Name boolean variables so they read like a yes/no question: `isLoggedIn`, `hasPermission`, `isGameOver`. When you read the condition later, it makes instant sense: `if (isLoggedIn)` is clearer than `if (loginStatus == 1)`.

> ⚠️ **Do Not Confuse = and ==**
>
> `=` is assignment — it stores a value. `==` is comparison — it asks a question. Writing `if (x = 5)` instead of `if (x == 5)` is one of the most common beginner bugs. Java will flag this with primitives, but it pays to understand the difference deeply before we get to objects.

2. The `if` Statement

The `if` statement runs a block of code *only when* a condition is true. If the condition is false, Java skips that block entirely and continues with whatever comes after.

Syntax

```
if (condition) {
    // code here runs only when condition is true
}
```

Example

```
int temperature = 95;

if (temperature > 90) {
    System.out.println("Stay hydrated — it's hot out there!");
}

System.out.println("Program continues.");

// Output (when temperature is 95):
// Stay hydrated — it's hot out there!
// Program continues.

// Output (when temperature is 75):
// Program continues.
```

Always Use Braces

Java technically allows you to skip braces when only one statement follows the `if`:

```
if (score >= 90)
    System.out.println("A grade");   // compiles, but risky
```

⚠️ Always Use Braces — No Exceptions

In this class, and in professional Java, always use braces even for single-statement bodies. Apple's infamous SSL "goto fail" bug (2014) happened because a developer added a second line without braces — it executed unconditionally. A missing pair of curly braces compromised millions of devices. Get in the habit now: always wrap the body in `{ }`.

🔧 Try It

Write a program that reads an integer from the user (use `Scanner`). If the number is negative, print `"Negative number entered."` Then, regardless of what the user typed, print `"Done."`

3. The `if-else` Statement

When you need to do one thing *or* another — but never both — use `if-else`. Exactly one of the two blocks runs every single time.

Syntax

```
if (condition) {
    // runs when condition is true
} else {
    // runs when condition is false
}
```

Example: Pass or Fail

```
int score = 62;

if (score >= 70) {
    System.out.println("You passed!");
} else {
    System.out.println("You did not pass. Come to office hours.");
}

// Output: You did not pass. Come to office hours.
```

> 🧠 **Think of It This Way**
>
> A vending machine: you insert money. The machine checks — *"Is this enough?"* If yes, dispense the snack. If no, return the money. Both outcomes are handled. There is no third option — the machine always does one or the other.

> 🔧 **Try It**
>
> Read an integer from the user. Print `"Even"` if the number is divisible by 2, or `"Odd"` otherwise. Hint: use the modulo operator `%`. The expression `number % 2 == 0` is true when a number is even.

4. The `if-else-if` Chain

When there are more than two possible paths, chain multiple `else if` clauses together. Java checks them in order from top to bottom and runs the *first one that is true*. Everything after that is skipped, even if it would also be true.

Syntax

```
if (condition1) {
    // runs if condition1 is true
} else if (condition2) {
    // runs if condition1 is false AND condition2 is true
} else if (condition3) {
    // runs if conditions 1-2 are false AND condition3 is true
} else {
    // runs if ALL conditions above were false (catch-all)
}
```

Example: Letter Grade

```
int score = 78;
char 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.printf("Grade: %c%n", grade);
// Output: Grade: C
```

> 💡 **Order Matters — Most Restrictive First**
>
> Always check the most restrictive (most specific) condition first. In the grade example, if you checked `score >= 60` before `score >= 90`, a score of 95 would fall into the "D" bucket — because 95 is also >= 60, and that branch fires first. Put the tightest check at the top.

The final `else` acts as a catch-all. It runs when nothing else matched. Always include one when there's a scenario where none of your conditions will be true — otherwise your variable might end up uninitialized and the compiler will complain.

> 🔧 **Try It**
>
> Read a temperature in Fahrenheit. Print one of these based on the value:
>
> - Below 32 — `"Freezing"`
> - 32 to 59 — `"Cold"`
> - 60 to 79 — `"Comfortable"`
> - 80 and above — `"Hot"`

5. Nested `if` Statements

You can place an `if` statement inside another `if` statement. This is called **nesting**. The inner `if` only executes when the outer condition is already true, so it applies an additional check within a specific context.

Example: Age and ID Check

```
int age = 20;
boolean hasID = true;

if (age >= 18) {
    if (hasID) {
        System.out.println("Welcome — enjoy the event.");
    } else {
        System.out.println("You're old enough, but we need to see your ID.");
    }
} else {
    System.out.println("Sorry, you must be 18 or older.");
}

// Output: Welcome — enjoy the event.
```

Here, we only check for an ID after confirming the person is old enough. The ID check makes no sense for someone under 18, so it lives inside the age check.

When to Nest vs. When to Chain

Before you nest, ask yourself: *"Can I rewrite this as a single condition using a logical operator?"* Often the answer is yes:

```
// Nested version — reads like two separate gatekeepers
if (age >= 18) {
    if (hasID) {
        System.out.println("Welcome");
    }
}

// Flattened with logical AND — often cleaner
if (age >= 18 && hasID) {
    System.out.println("Welcome");
}
```

Both produce the same result. The flattened version is usually preferred unless you need different messages for each failed condition (like the age-vs-ID example above).

> 💡 **Tip: Keep It Shallow**
>
> If you're nesting more than two levels deep, step back and refactor. Deep nesting — four or five levels — is almost always a sign there's a cleaner approach. Your future self (and your grader) will thank you.

6. Logical Operators

Logical operators let you combine multiple boolean expressions into one compound condition. Java has three: `&&` (AND), `||` (OR), and `!` (NOT).

&& — Logical AND

The whole expression is `true` only if *both* sides are `true`.

```
int age = 25;
boolean hasLicense = true;

if (age >= 16 && hasLicense) {
    System.out.println("You may drive.");
}
// Output: You may drive.
```

|| — Logical OR

The whole expression is `true` if *at least one* side is `true`.

```
boolean isMember = false;
boolean hasCoupon = true;

if (isMember || hasCoupon) {
    System.out.println("10% discount applied.");
}
// Output: 10% discount applied.
```

! — Logical NOT

Flips a boolean value: `true` becomes `false`, `false` becomes `true`.

```
boolean isLoggedIn = false;

if (!isLoggedIn) {
    System.out.println("Please log in to continue.");
}
// Output: Please log in to continue.
```

Truth Tables

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Short-Circuit Evaluation

Java doesn't evaluate more than it has to. With `&&`: if the left side is `false`, the right side is never checked — the result is already `false`. With `||`: if the left side is `true`, the right side is skipped. This is called **short-circuit evaluation**.

> 💡 **Tip: Put the Cheap Check First**
>
> Place the condition that's quickest to evaluate on the left. If it short-circuits, the right side never runs. This matters in performance-sensitive code, but it's also a good habit to build now.

7. Comparing Strings

This one trips up almost every Java beginner. When you compare two Strings using `==`, you're not comparing their contents — you're comparing their *memory addresses*. Two String variables can hold the exact same text but still fail a `==` check.

Why == Doesn't Work for Strings

```
String a = new String("hello");
String b = new String("hello");

System.out.println(a == b);        // false  (different objects in memory)
System.out.println(a.equals(b));   // true   (same content)
```

Think of it this way: `==` asks "are these the same physical object?" `.equals()` asks "do these contain the same text?" For Strings, you almost always want the second question.

Using .equals()

```
String userInput = "yes";

if (userInput.equals("yes")) {
    System.out.println("Great, let's continue.");
}
// Output: Great, let's continue.
```

> 💡 **Tip: Put the Literal on the Left**
>
> It's a common Java convention to call `.equals()` on the known string literal, not the variable: `"yes".equals(userInput)` instead of `userInput.equals("yes")`. If `userInput` is `null`, the first version won't crash — the second will throw a NullPointerException.

Case-Insensitive Comparison: .equalsIgnoreCase()

When you don't care whether the user typed "YES", "yes", or "Yes", use `.equalsIgnoreCase()`:

```
String answer = "YES";

if (answer.equalsIgnoreCase("yes")) {
    System.out.println("Got it — answer is yes.");
}
// Output: Got it — answer is yes.
```

The .compareTo() Method (Brief Introduction)

There's a third option: `.compareTo()`. It returns an integer, not a boolean. It returns 0 if the strings are equal, a negative number if the calling string comes before the argument alphabetically, and a positive number if it comes after. We'll see this more when we get to sorting. For now, just know it exists:

```
String s1 = "apple";
String s2 = "banana";

System.out.println(s1.compareTo(s2));  // negative (apple comes before banana)
System.out.println(s2.compareTo(s1));  // positive
System.out.println(s1.compareTo(s1));  // 0 (same)
```

> ⚠️ **Always Use .equals() for String Comparison**
>
> Using `==` on Strings produces unpredictable results. Sometimes it works (when Java reuses the same string literal from its pool). Sometimes it doesn't. Never rely on it — always use `.equals()` or `.equalsIgnoreCase()`.

8. The Conditional (Ternary) Operator

The **ternary operator** is a compact way to write a simple `if-else` that produces a value. The name comes from the fact that it takes three operands.

Syntax

```
result = condition ? valueIfTrue : valueIfFalse;
```

Read it as: *"If condition is true, use valueIfTrue; otherwise use valueIfFalse."*

Example

```
int score = 74;

// Long way with if-else:
String result;
if (score >= 70) {
    result = "Pass";
} else {
    result = "Fail";
}

// Shorter with ternary:
String result2 = (score >= 70) ? "Pass" : "Fail";

System.out.println(result2);
// Output: Pass
```

Another Common Use

```
int a = 15, b = 22;
int max = (a > b) ? a : b;

System.out.println("Larger value: " + max);
// Output: Larger value: 22
```

> 💡 **Tip: Use It for Simple Assignments**
>
> The ternary operator is great for one-liners where you're assigning one of two values. But don't nest ternary operators or use them for complex logic — that's when regular `if-else` is clearer. Readability beats cleverness every time.

9. The `switch` Statement

When you're testing a single variable against a list of specific values, `switch` is often cleaner than a long `if-else-if` chain. It jumps directly to the matching `case` instead of checking each condition in sequence.

Syntax

```
switch (expression) {
    case value1:
        // code for value1
        break;
    case value2:
        // code for value2
        break;
    case value3:
        // code for value3
        break;
    default:
        // code when nothing matched
}
```

Example: Day of Week

```
int day = 3;  // 1 = Monday, 7 = Sunday

switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
}
// Output: Wednesday
```

The break Statement

The `break` at the end of each case tells Java to exit the `switch` block. Without it, Java "falls through" to the next case and keeps executing code until it hits a `break` or the end of the switch. This is called **fall-through**.

Fall-Through (When It's Intentional)

Most of the time, fall-through is a mistake. But occasionally it's useful when multiple cases should do the same thing:

```
int month = 4;  // April

switch (month) {
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println("30 days");
        break;
    case 2:
        System.out.println("28 or 29 days");
        break;
    default:
        System.out.println("31 days");
}
// Output: 30 days
```

Cases 4, 6, 9, and 11 all fall through to the same output. This is intentional and clean — no duplicated code.

Strings in switch (Java 7+)

Since Java 7, you can use `String` values in a `switch` expression:

```
String command = "start";

switch (command) {
    case "start":
        System.out.println("Starting the engine...");
        break;
    case "stop":
        System.out.println("Stopping the engine...");
        break;
    case "status":
        System.out.println("Engine is running.");
        break;
    default:
        System.out.println("Unknown command: " + command);
}
// Output: Starting the engine...
```

> 💡 **switch vs. if-else-if — Which One?**
>
> Use `switch` when: (1) you're testing one variable against a fixed list of exact values, and (2) the cases are discrete — not ranges. Use `if-else-if` for ranges (`score >= 90`), complex boolean expressions, or when you need `&&` / `||` . You can't do `case score >= 90:` — that's not valid in a switch.

10. Common Mistakes

These are the bugs I see most often in student code. Learn them now so you don't waste debugging time later.

Mistake 1: Forgetting Braces

```
// WRONG — second line always runs
if (score >= 90)
    System.out.println("Excellent!");
    System.out.println("You got an A.");   // not inside the if!

// CORRECT
if (score >= 90) {
    System.out.println("Excellent!");
    System.out.println("You got an A.");
}
```

The indentation tricks your eye, but Java doesn't care about indentation. Without braces, only the *first* line after the `if` is conditional. The second line runs no matter what.

Mistake 2: Using = Instead of ==

```
// WRONG — this assigns 5 to x, doesn't compare
if (x = 5) {   // compiler error with int; doesn't work as expected
    ...
}

// CORRECT
if (x == 5) {
    ...
}
```

Mistake 3: The Dangling Else

When you have nested `if`s without braces, it can be unclear which `if` an `else` belongs to. This is called the **dangling else** problem:

```
// Ambiguous formatting — where does the else go?
if (x > 0)
    if (y > 0)
        System.out.println("Both positive");
else                        // this else belongs to the INNER if!
    System.out.println("x is not positive");  // misleading

// CORRECT — use braces to make it explicit
if (x > 0) {
    if (y > 0) {
        System.out.println("Both positive");
    }
} else {
    System.out.println("x is not positive");
}
```

Java matches an `else` to the nearest preceding `if`. When you use braces consistently, this is never a problem.

Mistake 4: Missing break in switch

```
int day = 2;

switch (day) {
    case 1:
        System.out.println("Monday");
        // forgot break!
    case 2:
        System.out.println("Tuesday");
        // forgot break!
    case 3:
        System.out.println("Wednesday");
        break;
}
// Output (when day = 2):
// Tuesday
// Wednesday      <-- not what you wanted!
```

Without `break`, execution falls through from case 2 into case 3. Always add `break` unless you're intentionally using fall-through, and comment it clearly when you do.

Mistake 5: Using == for String Comparison

```
String input = "yes";

// WRONG — may work sometimes, but unreliable
if (input == "yes") {
    System.out.println("OK");
}

// CORRECT
if (input.equals("yes")) {
    System.out.println("OK");
}
```

📋 **Common Mistakes Summary**

- Always use `{ }` — even for single-statement bodies.
- Use `==` for comparison, `=` for assignment. Never swap them.
- Brace your nested `if` s to avoid dangling else confusion.
- Every `case` in a `switch` needs a `break` (unless fall-through is intentional).
- Compare Strings with `.equals()` , never `==` .

Vocabulary Review

Term	Definition
boolean expression	An expression that evaluates to either `true` or `false` .
relational operator	An operator that compares two values: `==` , `!=` , `>` , `<` , `>=` , `<=` .
if statement	A conditional statement that executes a block only when its condition is true.
if-else statement	A conditional that chooses between two paths: one when true, one when false.
if-else-if chain	Multiple conditions checked in sequence; the first true branch runs.
nested if	An `if` statement inside another `if` statement.
logical operator	An operator that combines boolean expressions: `&&` , `||` , `!` .
short-circuit evaluation	Java stops evaluating a compound condition as soon as the result is determined.
ternary operator	A compact three-part operator: `condition ? valueIfTrue : valueIfFalse` .
switch statement	A multi-branch statement that jumps to the matching `case` value.

Term	Definition
fall-through	Execution continues into the next `case` when no `break` is present.
.equals()	A String method that compares content (not memory address).
dangling else	An ambiguous `else` that attaches to the wrong `if` due to missing braces.

📋 Module 04 Summary

- Boolean expressions evaluate to `true` or `false` using relational operators.
- The `if` statement conditionally executes a block; always wrap the body in braces.
- `if-else` handles two-way decisions; `if-else-if` handles multiple paths checked in order.
- Nested `if` s work but can often be simplified with logical operators.
- `&&` requires both sides true; `||` requires at least one; `!` flips a boolean.
- Use `.equals()` (not `==`) to compare String contents.
- The ternary operator (`? :`) is a shorthand for simple one-value `if-else` assignments.
- `switch` is ideal for testing one variable against specific discrete values; always include `break` .
- The five most common mistakes: missing braces, `=` vs `==` , dangling else, missing `break` , `==` on Strings.

Module 04 Quiz — Conditional Statements

Choose the best answer for each question. Write your answer on the line provided.

1. Which of the following boolean expressions evaluates to `true`?

A) `10 >= 10`

B) `5 != 5`

C) `7 == 8`

D) `3 > 9`

Answer: _____

2. What is the output of the following code?

```
int x = 15;
if (x > 10) {
    System.out.println("Big");
} else {
    System.out.println("Small");
}
```

A) `Small`

B) `Big`

C) `Big Small`

D) Nothing prints

Answer: _____

3. In an `if-else-if` chain, what happens after one branch executes?

A) All remaining branches are checked and may also run.

B) The remaining branches are skipped entirely.

C) The program restarts from the top of the chain.

D) A compile-time error is thrown.

Answer: _____

4. What is the output of the following code?

```
boolean a = true;
boolean b = false;

if (a && b) {
    System.out.println("Both true");
} else if (a || b) {
    System.out.println("At least one true");
} else {
    System.out.println("Neither true");
}
```

A) `Both true`

B) `Neither true`

C) `At least one true`

D) `Both true At least one true`

Answer: _____

5. Why should you use `.equals()` instead of `==` when comparing two String objects?

A) `==` only works with numbers; it causes a compile error with Strings.

B) `==` compares memory addresses, not string content, which can give wrong results.

C) `.equals()` is faster than `==` for all data types.

D) `==` ignores whitespace differences; `.equals()` does not.

Answer: _____

6. What is the value of `result` after this line executes?

```
int score = 55;
String result = (score >= 70) ? "Pass" : "Fail";
```

A) `"Pass"`

B) `null`

C) Compile error — ternary can't return Strings

D) `"Fail"`

Answer: _____

7. What is the output of the following `switch` statement?

```
int n = 2;
switch (n) {
    case 1:
        System.out.println("One");
        break;
    case 2:
        System.out.println("Two");
    case 3:
        System.out.println("Three");
        break;
    default:
        System.out.println("Other");
}
```

A) `Two`

B) `Two Three Other`

C) `Two Three`

D) `Three`

Answer: _____

8. Which of the following correctly checks whether a String variable `color` equals `"blue"`, ignoring case?

A) `color == "blue"`

B) `color.equals("blue")`

C) `color.equalsIgnoreCase("blue")`

D) `"blue".compareTo(color) == 0`

Answer: _____

9. What is printed by the following code?

```
int x = 5;
if (x > 3)
    System.out.println("A");
    System.out.println("B");
```

A) `A B`

B) `A` only

C) `B` only

D) Nothing

Answer: _____

10. When should you prefer a `switch` statement over an `if-else-if` chain?

A) When you need to check a range of values (e.g., `score >= 90`).

B) When you need to use `&&` or `||` inside the conditions.

C) When the condition involves multiple variables at the same time.

D) When you are testing one variable against a list of specific discrete values.

Answer: _____

✅ Answer Key

Question	Answer	Explanation
1	**A**	The `==` operator compares two values for equality.
2	**B**	The `if-else` statement provides an alternative path when the condition is false.
3	**B**	Logical AND (`&&`) requires both conditions to be true.
4	**C**	A `switch` statement compares one variable against multiple constant values.
5	**B**	Without `break` , execution falls through to subsequent cases.
6	**D**	The ternary operator syntax is `condition ? valueIfTrue : valueIfFalse` .
7	**C**	Logical OR (`||`) is true when at least one condition is true.
8	**C**	String comparison uses `.equals()` , not `==` .
9	**A**	The `!` (NOT) operator inverts a boolean value.
10	**D**	`else if` chains test multiple conditions in sequence.