

2.5 Compound Boolean Expressions

Self-Study Guide · About 45 minutes · With answer key

By the end you will be able to

- Use the logical operators NOT, AND, and OR to build a condition out of smaller true-or-false pieces.
- Work out whether a compound Boolean expression is true or false for given values.
- Apply the order NOT, then AND, then OR, and use parentheses to change that order.
- Explain when Java skips the right side of an AND or an OR, and use that to keep a condition from crashing.

What to remember from earlier

- A boolean variable holds true or false. ([Topic 1.2: Variables and Data Types, 1.2.3 The Three Primitive Types: int, double, and boolean](#))
- Relational operators compare two values and give a boolean, so `age >= 13` is true or false. ([Topic 2.2: Boolean Expressions](#))
- An if statement runs its body only when its condition is true, and else runs when it is false. ([Topic 2.3: if Statements, 2.3.3 Two-Way Selection with if-else](#))
- In nested ifs, the inner condition is checked only when the outer one is true. ([Topic 2.4: Nested if Statements, 2.4.1 An if Inside Another if](#))
- An if-else-if chain runs the body of the first condition that is true. ([Topic 2.4: Nested if Statements, 2.4.4 Multiway Selection with else if](#))
- Integer division by zero crashes with an `ArithmeticException`. ([Topic 1.3: Expressions and Output, 1.3.9 Dividing an Integer by Zero](#))

2.5.1 Reversing a Condition with NOT

So far, every condition you have written asks one question: is the score at least 90? Is the password equal to "open"? Real decisions are often built from several questions. A movie theater sells a teen ticket when a customer is at least 13 **and** at most 19. A museum lets you in free when you are under 5 **or** at least 65. A phone warns you when the battery is **not** above 20 percent. Java has three operators for combining and flipping true-or-false values like these. This chapter teaches all three, starting with the simplest: "not."

Recall from Topic 1.2 that a `boolean` holds exactly one of two values, `true` or `false`, and from Topic 2.2 that a comparison such as `battery > 20` produces a `boolean`. Sometimes the value you have is the opposite of the one you want. You have a variable that says whether it is raining, but you want to know whether it is dry. In everyday language you just say "not raining." Java lets you do the same thing.

A **logical operator** is an operator whose operands are `boolean` values and whose result is a `boolean` value. The first one is the **NOT operator**, written `!` (an exclamation mark). It goes in front of a single `boolean` value and produces the opposite value:

a	!a
---	----

true	false
false	true

Read `!isRaining` aloud as "not is raining," or more naturally, "it is not raining." The `!` takes one operand, the value right after it.

KEY IDEA

`!a` is true when `a` is false, and false when `a` is true. NOT flips a boolean value.

WORKED EXAMPLE Flipping a boolean variable

Watch which variables change and which do not.

```
1 boolean isRaining = true;
2 boolean isDry = !isRaining;
3 System.out.println(isRaining);
4 System.out.println(isDry);
5 System.out.println(!isDry);
```

Line	What Java does	isRaining	isDry	Printed
1	Stores true.	true		
2	Evaluates <code>!isRaining</code> : not true is false. Stores false.	true	false	
3	Prints <code>isRaining</code> .	true	false	true
4	Prints <code>isDry</code> .	true	false	false
5	Evaluates <code>!isDry</code> : not false is true. Prints it.	true	false	true

Output:

```
true
false
true
```

Two details matter here. First, `!isRaining` on line 2 did **not** change `isRaining`; it still holds `true` afterward. Like `battery + 1`, the expression `!isRaining` produces a new value and leaves the variable alone. Only an assignment changes a variable. Second, line 5 shows that `!` can be used anywhere a boolean value can, including right inside `println`.

Now vary one thing: instead of flipping a boolean variable, flip the result of a comparison. A phone should warn you when the battery is not above 20 percent. You can write the comparison `battery > 20`, wrap it in parentheses, and put `!` in front.

WORKED EXAMPLE Flipping a comparison

```
1 int battery = 15;
2 if (!(battery > 20))
3 {
4     System.out.println("Plug in your phone.");
5 }
6 else
7 {
8     System.out.println("Battery is fine.");
9 }
```

Java works from the inside out:

Step	Expression	Value
1	battery > 20	false
2	!(false)	true
3	Condition is true	run the if body

Output:

```
Plug in your phone.
```

The parentheses make Java finish the comparison first and then flip its result. If `battery` held 80, step 1 would be `true`, step 2 would be `false`, and the `else` body would print `Battery is fine.` instead.

WATCH OUT

Leaving out the parentheses around a comparison. It is natural to write "not battery greater than 20" exactly as it sounds:

```
1 int battery = 15;
2 if (!battery > 20)
3 {
4     System.out.println("Plug in your phone.");
5 }
```

This does not compile. Java reports something like:

```
error: bad operand type int for unary operator '!'
```

Here is why. `!` grabs only the single value right after it, and it grabs it **before** any comparison happens. So Java reads line 2 as `(!battery) > 20`. But `battery` is an `int`, and "not 15" means nothing: `!` works only on `boolean` values. ("Unary" means the operator takes one operand.) The fix is to put parentheses around the whole comparison so it becomes a `boolean` first: `!(battery > 20)`. (Exact error wording can vary between Java versions.)

PAUSE AND TRY**Question 1** Tracing

What does this code print? Track both variables line by line.

```
1 boolean lightsOn = false;
2 boolean dark = !lightsOn;
3 lightsOn = !lightsOn;
4 System.out.println(lightsOn);
5 System.out.println(dark);
6 System.out.println(!dark);
```

Question 2 Multiple choice

temp is an int. Which expression is true exactly when temp is **not** greater than 100?

- A. `!(temp > 100)`
- B. `!temp > 100`
- C. `!(temp < 100)`
- D. `temp != 100`

2.5.2 Requiring Both Conditions with AND

A theater sells a teen ticket to anyone from 13 through 19 years old. One comparison cannot check that: `age >= 13` lets in a 40-year-old, and `age <= 19` lets in a 6-year-old. The customer has to pass **both** tests.

With Topic 2.4 you could already do this with a nested `if`:

```
1 if (age >= 13)
2 {
3     if (age <= 19)
4     {
5         System.out.println("Teen ticket");
6     }
7 }
```

That works, but it takes two `if` statements and two levels of braces to say one simple thing: "13 or older, and 19 or younger." Java lets you say it in a single condition.

The **AND operator**, written `&&` (two ampersands), combines two `boolean` values. The result is `true` only when **both** values are `true`; if either one is `false`, the result is `false`. It works like a door with two locks: it opens only when both locks are open.

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Now the teen check is one condition: `age >= 13 && age <= 19`. An expression like this, built from smaller `boolean` expressions joined by logical operators, is called a **compound Boolean expression**. However many pieces it has, it always evaluates to one `boolean` value, `true` or `false`, so it can go anywhere a `boolean` can: in an `if` condition, in a `boolean` variable, or in `println`.

KEY IDEA

`a && b` is `true` only when both `a` and `b` are `true`. If either side is `false`, the whole expression is `false`.

WORKED EXAMPLE Checking an age range

The same condition is evaluated twice, with two different ages.

```
1 int age = 16;
2 boolean isTeen = age >= 13 && age <= 19;
3 System.out.println(isTeen);
4 age = 22;
5 isTeen = age >= 13 && age <= 19;
6 System.out.println(isTeen);
```

Java evaluates each comparison to a `boolean` first and then combines the two results with `&&`:

Line	age	age >= 13	age <= 19	&& result
2	16	true	true	true
5	22	true	false	false

Output:

```
true
false
```

For 22, the first test passes, but one passing test is not enough for `&&`. Line 5 also shows that the condition does not update by itself when `age` changes: line 5 has to evaluate it again and store the new result.

Now vary the operands. The two sides of `&&` do not have to be comparisons; any `boolean` values work, including `boolean` variables. And the result can control an `if-else` directly.

WORKED EXAMPLE Two boolean variables in an if-else

A student can drive to school only with a license and a car.

```
1 boolean hasLicense = true;
2 boolean hasCar = false;
3 if (hasLicense && hasCar)
4 {
5     System.out.println("You can drive to school.");
6 }
7 else
8 {
9     System.out.println("Take the bus.");
10 }
```

hasLicense is true and hasCar is false. That is the second row of the table: true && false is false. The condition is false, so the else body runs:

Take the bus.

Notice there is no == true in the condition. hasLicense already holds a boolean, so it can be used as an operand as it is.

WATCH OUT

Chaining comparisons the way math does. In math class, $13 \leq \text{age} \leq 19$ is a normal way to write a range, so students try the same thing in Java:

```
1 int age = 16;
2 if (13 <= age <= 19)
3 {
4     System.out.println("Teen ticket");
5 }
```

This does not compile. Java reports something like:

```
error: bad operand types for binary operator '<='
    first type:  boolean
    second type:  int
```

Java does not read the range as one idea. It evaluates `13 <= age` first, which gives true. That leaves `true <= 19`, and a boolean cannot be compared with a number using `<=`. Java has no way to ask whether true is less than 19. The fix is to write both comparisons in full and join them with `&&`:

```
1 int age = 16;
2 if (13 <= age && age <= 19)
3 {
4     System.out.println("Teen ticket");
5 }
```

This prints `Teen ticket`. (Exact error wording can vary between Java versions.)

PAUSE AND TRY**Question 3** Tracing

A picnic needs sun and a temperature strictly between 60 and 85. What does this code print?

```
1 int temp = 68;
2 boolean sunny = true;
3 boolean picnic = sunny && temp > 60 && temp < 85;
4 System.out.println(picnic);
5 temp = 90;
6 picnic = sunny && temp > 60 && temp < 85;
7 System.out.println(picnic);
```

Question 4 Multiple choice

For which value of `points` does `points > 50 && points % 2 == 0` evaluate to `true`?

- A. 51
- B. 48
- C. 64
- D. 50

2.5.3 Accepting Either Condition with OR

A museum lets children under 5 and adults 65 or older in for free. Here a visitor needs to pass **at least one** test, not both. Nobody is both under 5 and over 65, so `&&` would let no one in free. Java needs a second way to combine conditions.

The **OR operator**, written `||` (two vertical bars; on most keyboards the bar is above the Enter key), combines two `boolean` values. The result is `true` when **at least one** of them is `true`, and `false` only when both are `false`. Think of a room with two doors: you can get in if either door is open.

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

Look at the first row. In everyday speech, "or" sometimes means "one or the other, but not both," as in "soup or salad comes with your meal." Java's `||` never means that. When both sides are `true`, `||` is `true`.

KEY IDEA

`a || b` is `true` when `a` is `true`, `b` is `true`, or both are. It is `false` only when both sides are `false`.

WORKED EXAMPLE Free entry for young children and seniors

```

1 int age = 70;
2 boolean freeEntry = age < 5 || age >= 65;
3 System.out.println(freeEntry);
4 age = 30;
5 freeEntry = age < 5 || age >= 65;
6 System.out.println(freeEntry);

```

Line	age	age < 5	age >= 65	result
2	70	false	true	true
5	30	false	false	false

Output:

```

true
false

```

For 70, one side is `true`, and that is enough. For 30, neither side is `true`, which is the only way `||` gives `false`.

To see the "both are true" row in action, here is a store that gives a discount to anyone with a coupon or a membership:

```

1 boolean hasCoupon = true;
2 boolean isMember = true;
3 System.out.println(hasCoupon || isMember);

```

This prints `true`. A customer with both a coupon and a membership still gets the discount; having two reasons does not cancel it out.

Now vary the operands again. The pieces of an `||` can be any `boolean` expressions, including calls to the `String` method `equals` from Topic 1.15, which returns `true` when two strings hold the same text.

WORKED EXAMPLE Checking for either weekend day

```

1 String day = "Sun";
2 if (day.equals("Sat") || day.equals("Sun"))
3 {
4     System.out.println("Weekend hours: 10 to 4");
5 }
6 else
7 {
8     System.out.println("Weekday hours: 8 to 6");
9 }

```

Step	Expression	Value
1	<code>day.equals("Sat")</code>	false
2	<code>day.equals("Sun")</code>	true

3	false true	true
---	---------------	------

The condition is `true`, so the `if` body runs:

```
Weekend hours: 10 to 4
```

If `day` held `"Wed"`, both calls would return `false`, the condition would be `false`, and the `else` body would print the weekday hours.

WATCH OUT

Shortening the second comparison. In English you might say "if the day is 6 or 7," so students write:

```
1 int day = 6;
2 if (day == 6 || 7)
3 {
4     System.out.println("Weekend");
5 }
```

This does not compile. Java reports something like:

```
error: bad operand types for binary operator '||'
    first type:  boolean
    second type:  int
```

Both sides of `||` must be `boolean` values. The left side, `day == 6`, is a `boolean`. The right side is just the number `7`, which is not true or false. Java does not fill in the missing `day ==` for you. Write each comparison in full:

```
1 int day = 6;
2 if (day == 6 || day == 7)
3 {
4     System.out.println("Weekend");
5 }
```

This prints `Weekend`. (Exact error wording can vary between Java versions.)

WATCH OUT

Using `&&` when the situation needs `||`. English "and" can fool you. You might describe the museum rule as "children under 5 and seniors get in free," and then write `&&`:

```
1 int age = 70;
2 if (age < 5 && age >= 65)
3 {
4     System.out.println("Free entry");
5 }
6 else
7 {
8     System.out.println("Pay full price");
9 }
```

This compiles and prints `Pay full price`, even for a 70-year-old. For `age` equal to 70, `age < 5` is `false`, so `&&` is `false`. In fact, no number is both less than 5 and at least 65, so this condition is `false` for every age and nobody ever gets in free. Java gives no error, because the code is legal; it just does not mean what you intended. Before choosing an operator, ask: must **one** person pass **both** tests (`&&`), or is passing **either** test enough (`||`)? Here, either is enough, so the fix is `age < 5 || age >= 65`.

PAUSE AND TRY

Question 5 Tracing

What does this code print? Show the value of each side of the `||`.

```
1 String weather = "snow";
2 int temp = 40;
3 if (weather.equals("rain") || temp < 32)
4 {
5     System.out.println("Stay inside.");
6 }
7 else
8 {
9     System.out.println("Go outside.");
10 }
```

Question 6 Multiple choice

A ride closes when `windSpeed` (an `int`) is above 40 or when `isRaining` (a `boolean`) is `true`. Which expression is `true` exactly when the ride should close?

- A. `windSpeed > 40 && isRaining`
- B. `windSpeed > 40 || isRaining`
- C. `windSpeed || isRaining > 40`
- D. `!(windSpeed > 40) || isRaining`

2.5.4 Order of Evaluation

Once a condition has more than one logical operator, a question comes up that a single operator never raised. What does this mean?

```
1 a || b && c
```

Does Java combine `a || b` first and then `&& c`? Or `b && c` first and then `a ||`? The two orders can give different answers, so Java needs a rule. You met the same problem with arithmetic in Topic 1.3: `2 + 3 * 4` is 14, not 20, because multiplication has higher **precedence** (it is done first) than addition.

Logical operators have their own precedence:

1. `!` (NOT) first
2. `&&` (AND) next
3. `||` (OR) last

So in `a || b && c`, the `&&` happens before the `||`, as if you had written `a || (b && c)`. A memory hook: AND is like multiplication and OR is like addition, so AND goes first.

These operators also fit into the bigger order you already know. `!` applies to the single value right after it, before any arithmetic or comparison, which is why `!battery > 20` failed in 2.5.1. Arithmetic and comparisons such as `<`, `>`, `<=`, `>=`, and `==` all happen **before** `&&` and `||`. That is why `age >= 13 && age <= 19` works without parentheses: both comparisons become `boolean` values first, and only then does `&&` combine them.

Order	Operators
1	parentheses ()
2	!
3	* / %, then + -
4	< > <= >=, then == !=
5	&&
6	

Parentheses beat everything, so whenever you want a different order, or just want the order to be obvious to a reader, add them.

KEY IDEA

Comparisons are evaluated before logical operators. Among the logical operators, `!` goes first, then `&&`, then `||`. Parentheses override the order.

Precedence tells you how an expression is **grouped**: which operands each operator combines, as if Java had added invisible parentheses. Once you know the grouping, you can find the value by working out the innermost groups first, as the tables below do. (In 2.5.5 you will see that Java sometimes skips part of an expression. That never changes the value; it only matters when the skipped part would have crashed.)

WORKED EXAMPLE The same operators, with and without parentheses

```

1 boolean a = true;
2 boolean b = false;
3 boolean c = false;
4 System.out.println(a || b && c);
5 System.out.println((a || b) && c);

```

Line 4, `a || b && c`, follows the precedence rule: `&&` first.

Step	Expression	Value
1	<code>b && c</code> → <code>false && false</code>	false
2	<code>a false</code> → <code>true false</code>	true

Line 5, `(a || b) && c`, uses parentheses to force `||` first.

Step	Expression	Value
1	<code>(a b)</code> → <code>true false</code>	true
2	<code>true && c</code> → <code>true && false</code>	false

Output:

```

true
false

```

The same three variables and the same two operators give different results. The only difference is which operator runs first.

Now add `!` and some comparisons, and compare `!` on one value with `!` on a whole parenthesized expression.

WORKED EXAMPLE NOT, comparisons, AND, and OR together

```

1 int x = 4;
2 boolean flag = false;
3 System.out.println(!flag && x > 5 || x == 4);
4 System.out.println(!(flag && x > 5 || x == 4));

```

Line 3 has no extra parentheses, so every rule from the table applies. `x` is 4 and `flag` is false.

Step	What runs	Expression becomes
1	<code>!flag</code> (NOT first)	<code>true && x > 5 x == 4</code>
2	comparisons	<code>true && false true</code>
3	<code>&&</code>	<code>false true</code>
4	<code> </code>	<code>true</code>

Line 4 puts the whole expression in parentheses with `!` outside, so the inside is finished first and `!` flips the final result.

Step	What runs	Expression becomes
------	-----------	--------------------

1	comparisons	!(false && false true)
2	&&	!(false true)
3		!(true)
4	!	false

Output:

```
true
false
```

On line 3, `!` flipped only `false`. On line 4, it flipped the whole result. Where the parentheses go decides what `!` applies to.

WATCH OUT

Writing a condition in the order you would say it. A theater gives free popcorn to members or seniors, but only on Tuesdays. A student writes:

```
1 boolean isMember = true;
2 int age = 30;
3 boolean isTuesday = false;
4 if (isMember || age >= 65 && isTuesday)
5 {
6     System.out.println("Free popcorn!");
7 }
8 else
9 {
10    System.out.println("Popcorn is $5.");
11 }
```

It is Wednesday (`isTuesday` is `false`), yet this prints `Free popcorn!`. Because `&&` runs before `||`, Java reads the condition as `isMember || (age >= 65 && isTuesday)`. That says "members any day, or seniors on Tuesday." Step by step: `age >= 65 && isTuesday` is `false && false`, which is `false`; then `isMember || false` is `true || false`, which is `true`.

The intended meaning is "(member or senior) and Tuesday," so the `||` must happen first. Add parentheses:

```
1 boolean isMember = true;
2 int age = 30;
3 boolean isTuesday = false;
4 if ((isMember || age >= 65) && isTuesday)
5 {
6     System.out.println("Free popcorn!");
7 }
8 else
9 {
10    System.out.println("Popcorn is $5.");
11 }
```

Now `(isMember || age >= 65)` is `true`, and `true && isTuesday` is `true && false`, which is `false`. It prints `Popcorn is $5`. Whenever a condition mixes `&&` and `||`, check which one you need to happen first, and use parentheses if it is the `||`.

PAUSE AND TRY

Question 7 Tracing

What does this code print? Show the order in which the operators run on each line.

```
1 boolean p = false;
2 boolean q = true;
3 boolean r = true;
4 System.out.println(p && q || r);
5 System.out.println(p && (q || r));
6 System.out.println(!p || q && !r);
```

Question 8 Multiple choice

Which choice adds parentheses that show exactly how Java groups `a || b && !c`, where `a`, `b`, and `c` are boolean variables?

- A. `(a || b) && (!c)`
- B. `a || (b && (!c))`
- C. `!(a || b && c)`
- D. `(a || b) && !c`

2.5.5 Short-Circuit Evaluation

A grade program finds a student's average by dividing the total of their scores by how many scores there are. Early in the term a new student has no scores yet, so the count is `0`. From Topic 1.3, integer division by zero crashes the program with an `ArithmeticException`. The program should check "there is at least one score **and** the average is at least 80," but the second part is exactly the part that can crash.

In Topic 2.4 you saw that an inner `if` is checked only when the outer `if`'s condition is `true`, so you could protect the division by nesting it inside `if (count > 0)`. It turns out `&&` already protects it on its own, because of how Java evaluates `&&` and `||`.

Look back at the AND table. If the left side of `&&` is `false`, the result is `false` no matter what the right side is. Java notices this. It evaluates the left side first, and if that side alone decides the answer, it **does not evaluate the right side at all**. The same is true for `||`: if the left side is `true`, the result is `true` no matter what, so the right side is skipped.

Operator	Left side is	Result	Right side
<code>&&</code>	<code>false</code>	<code>false</code>	skipped
<code>&&</code>	<code>true</code>	depends on the right	evaluated
<code> </code>	<code>true</code>	<code>true</code>	skipped

	false	depends on the right	evaluated
--	-------	----------------------	-----------

Skipping the right side when the left side already decides the result is called **short-circuit evaluation**. It is like a teacher who checks whether homework was turned in before grading it: if there is no homework, there is nothing to grade, so the second step never happens.

Usually skipping the right side only saves a little time. It matters when the right side would crash, because then a check on the left can act as a **guard**: a condition placed first so that the risky part runs only when it is safe.

KEY IDEA

Java evaluates the left side of `&&` or `||` first. If the left side of `&&` is `false`, or the left side of `||` is `true`, the right side is never evaluated.

When an expression mixes operators, the "sides" are the groups that precedence makes (2.5.4). In `a || b && c`, the right side of `||` is the whole group `b && c`. If `a` is `true`, Java skips that entire group.

WORKED EXAMPLE A guard before a division with AND

```

1 int total = 0;
2 int count = 0;
3 if (count > 0 && total / count >= 80)
4 {
5     System.out.println("Honor roll");
6 }
7 else
8 {
9     System.out.println("No honor roll yet");
10 }
```

Step	Expression	Value	Notes
1	<code>count > 0</code> → <code>0 > 0</code>	false	left side
2	<code>false && ...</code>	false	right side, <code>total / count >= 80</code> , is skipped
3	condition is false		run the else body

Output:

```
No honor roll yet
```

The division `total / count` would be `0 / 0`, which crashes, but it never runs. If `count` held 4 and `total` held 340, the left side would be `true`, Java would evaluate the right side (`340 / 4` is 85, and `85 >= 80` is `true`), and the program would print `Honor roll`.

Now vary the operator. With `||`, the guard is written the other way around: the left side checks for the dangerous case and is `true` when it happens.

WORKED EXAMPLE A guard before a division with OR

```

1 int total = 0;
2 int count = 0;
3 if (count == 0 || total / count < 60)
4 {
5     System.out.println("Check in with your teacher");
6 }
7 else
8 {
9     System.out.println("On track");
10 }

```

Step	Expression	Value	Notes
1	count == 0 → 0 == 0	true	left side
2	true ...	true	right side, total / count < 60, is skipped
3	condition is true		run the if body

Output:

```
Check in with your teacher
```

Again the division never runs. A student with no scores yet is sent to check in, and Java does not need the average to decide that.

WATCH OUT

Putting the risky part first. Short-circuiting only skips the **right** side. If you write the guard second, the division runs before the guard is ever checked:

```

1 int total = 0;
2 int count = 0;
3 if (total / count >= 80 && count > 0)
4 {
5     System.out.println("Honor roll");
6 }
7 else
8 {
9     System.out.println("No honor roll yet");
10 }

```

Nothing prints. The program crashes on line 3 with:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Java evaluates the left side of && first, and that left side is now `total / count >= 80`, which divides by zero. The `count > 0` check comes too late to help. The fix is to swap the two sides so the guard is on the left: `count > 0 && total / count >= 80`. Which side the guard is on decides whether the program crashes.

PAUSE AND TRY**Question 9** Tracing

What does this code print? Is the `substring` call ever run?

```
1 String word = "cat";
2 if (word.length() > 3 && word.substring(3, 4).equals("s"))
3 {
4     System.out.println("Fourth letter is s");
5 }
6 else
7 {
8     System.out.println("No fourth s");
9 }
```

Question 10 Multiple choice

`n` is an `int` that holds 0. Which expression crashes with an `ArithmeticException` when Java evaluates it?

- A. `n != 0 && 10 / n > 1`
- B. `n == 0 || 10 / n > 1`
- C. `10 / n > 1 && n != 0`
- D. `n > 0 && 10 / n > 1`

Putting it together

This program decides whether a child can ride a roller coaster. The ride is open when it is not raining and the wind is under 30 mph. A rider must be at least 48 inches tall and must be 12 or older or riding with an adult. Before reading the trace, predict the four lines of output.

WORKED EXAMPLE Roller coaster check

RideCheck.java

```
1 public class RideCheck
2 {
3     public static void main(String[] args)
4     {
5         int heightInches = 50;
6         int age = 11;
7         boolean withAdult = true;
8         boolean isRaining = false;
9         int windSpeed = 25;
10
11         boolean rideOpen = !isRaining && windSpeed < 30;
12         boolean tallEnough = heightInches >= 48;
13         boolean oldEnough = age >= 12 || withAdult;
14
15         System.out.println("Ride open: " + rideOpen);
16         System.out.println("Tall enough: " + tallEnough);
17         System.out.println("Old enough: " + oldEnough);
```

```

18
19     if (!rideOpen)
20     {
21         System.out.println("Sorry, the ride is closed.");
22     }
23     else if (tallEnough && oldEnough)
24     {
25         System.out.println("Enjoy the ride!");
26     }
27     else
28     {
29         System.out.println("Not this time.");
30     }
31 }
32 }

```

Lines 5–9 set the situation: a child 50 inches tall, age 11, with an adult, on a dry day with 25 mph wind. Lines 11–13 each store the result of a Boolean expression in a variable with a clear name, so the `if` statement later reads almost like English.

Line	Expression	Steps	Result
11	<code>!isRaining && windSpeed < 30</code>	<code>!false</code> is true; <code>25 < 30</code> is true; <code>true && true</code>	true
12	<code>heightInches >= 48</code>	<code>50 >= 48</code>	true
13	<code>age >= 12 withAdult</code>	<code>11 >= 12</code> is false; <code>false true</code>	true

Lines 15–17 print each result. String concatenation (Topic 1.15) turns a `boolean` into the text `true` or `false`.

Lines 19–30 are an `if-else-if` chain from Topic 2.4. The first condition, `!rideOpen`, is `!true`, which is `false`, so Java moves on. The second, `tallEnough && oldEnough`, is `true && true`, which is `true`, so its body runs and the rest of the chain is skipped.

```

Ride open: true
Tall enough: true
Old enough: true
Enjoy the ride!

```

Two details are worth a second look. On line 13, the 11-year-old is too young on age alone, but `||` needs only one `true`, and `withAdult` supplies it. Because `||` evaluated its left side first and found it `false`, Java did evaluate the right side here; short-circuiting happens only when the left side settles the answer.

Now suppose it starts raining. Here are just the lines that matter, with `isRaining` set to `true`:

```

1 boolean isRaining = true;
2 int windSpeed = 25;
3 boolean rideOpen = !isRaining && windSpeed < 30;
4 System.out.println("Ride open: " + rideOpen);
5 if (!rideOpen)
6 {
7     System.out.println("Sorry, the ride is closed.");
8 }

```

```

Ride open: false
Sorry, the ride is closed.

```

`!isRaining` is now `false`, so the `&&` on line 3 short-circuits: the wind check is skipped and `rideOpen` is `false`. In the full program, the first branch of the chain, `!rideOpen`, would be `true`, so the closed message prints and the height and age checks never matter.

Vocabulary

Term	Definition	Example
logical operator	An operator that works on boolean values and produces a boolean value: <code>!</code> , <code>&&</code> , or <code> </code>	<code>isOpen && hasTicket</code>
NOT operator	<code>!</code> , written before one boolean value; produces the opposite value	<code>!isRaining</code>
AND operator	<code>&&</code> ; true only when both sides are true	<code>age >= 13 && age <= 19</code>
compound Boolean expression	A Boolean expression built from smaller Boolean expressions joined by logical operators; evaluates to <code>true</code> or <code>false</code>	<code>age < 5 age >= 65</code>
OR operator	<code> </code> ; true when at least one side is true	<code>hasCoupon isMember</code>
precedence	The order in which operators are evaluated; for logical operators, <code>!</code> then <code>&&</code> then <code> </code>	<code>a b && c</code> means <code>a (b && c)</code>
short-circuit evaluation	Skipping the right side of <code>&&</code> or <code> </code> because the left side already decides the result	<code>count > 0 && total / count > 80</code>
guard	A condition placed on the left so the risky right side runs only when it is safe	<code>count > 0</code> in the line above

Chapter summary

- `!` flips a `boolean`: `!a` is `true` exactly when `a` is `false`. To flip a comparison, wrap it in parentheses: `!(battery > 20)`.
- `a && b` is `true` only when both sides are `true`; `a || b` is `true` when at least one side is `true`, including when both are. Every compound Boolean expression evaluates to a single `boolean`.
- Each side of `&&` and `||` must be a complete `boolean` expression: write `13 <= age && age <= 19`, not `13 <= age <= 19`, and `day == 6 || day == 7`, not `day == 6 || 7`.
- Comparisons happen before logical operators, and among logical operators `!` comes first, then `&&`, then `||`. Use parentheses to change the order.
- Java evaluates the left side of `&&` and `||` first and skips the right side when the left side decides the result (`false` for `&&`, `true` for `||`). Put a guard on the left to protect a risky right side.

Review questions

Question 11 Multiple choice

What does this code print?

```
1 int boxes = 0;
2 int items = 24;
3 boolean ok = boxes != 0 && items / boxes > 5 || items > 20;
4 System.out.println(ok);
```

- A. `true`
- B. `false`
- C. Nothing; it crashes with an `ArithmeticException`
- D. Nothing; it does not compile

Question 12 Tracing

What does this code print? For each condition in the chain, show whether it is `true` or `false`.

```
1 int score = 84;
2 int absences = 5;
3 boolean extraCredit = true;
4 String grade;
5 if (score >= 90 || score >= 88 && extraCredit)
6 {
7     grade = "A";
8 }
9 else if (score >= 80 && !(absences > 3))
10 {
11     grade = "B";
12 }
13 else if (score >= 70 || extraCredit)
14 {
15     grade = "C";
16 }
17 else
18 {
19     grade = "D";
20 }
21 System.out.println("Grade: " + grade);
```

Question 13 Debugging

This code should print `Valid month` only when `month` is from 1 through 12. With `month` equal to 15, it prints `Valid month`. Find the error, explain why every number passes, and correct the condition.

```
1 int month = 15;
2 if (month >= 1 || month <= 12)
3 {
4     System.out.println("Valid month");
5 }
6 else
7 {
8     System.out.println("Invalid month");
9 }
```

Question 14 Short answer

`count` holds 0. Both segments below run without crashing. Explain why each one is safe, and say what feature of Java makes the second one safe.

```
1 if (count > 0)
2 {
3     if (total / count > 10)
4     {
5         System.out.println("High average");
6     }
7 }
```

```
1 if (count > 0 && total / count > 10)
2 {
3     System.out.println("High average");
4 }
```

Question 15 Multiple choice

`a` and `b` are `boolean` variables. `a && b` evaluates to `false`, and `a || b` evaluates to `true`. What must be the case?

- A. Both `a` and `b` are `true`.
- B. Both `a` and `b` are `false`.
- C. Exactly one of `a` and `b` is `true`.
- D. `a` is `true` and `b` is `false`; no other combination works.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can use `!` to flip a boolean value or a comparison in parentheses. Reread: 2.5.1 Reversing a Condition with NOT
- ☐ I can use `&&` to require two conditions and write a range check correctly. Reread: 2.5.2 Requiring Both Conditions with AND
- ☐ I can use `||` to accept either condition, and I know it is true when both sides are true. Reread: 2.5.3 Accepting Either Condition with OR
- ☐ I can evaluate an expression that mixes `!`, `&&`, and `||` in the right order, and use parentheses to change it. Reread: 2.5.4 Order of Evaluation
- ☐ I can explain when Java skips the right side of `&&` or `||` and use a guard to prevent a crash. Reread: 2.5.5 Short-Circuit Evaluation

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

It prints:

```
true
true
false
```

Line	What Java does	lightsOn	dark
1	Stores false.	false	
2	!false is true; store it in dark.	false	true
3	!false is true; store it back in lightsOn.	true	true
4–6	Print lightsOn, dark, then !dark.	true	true

The trap is line 3. It changes `lightsOn`, but `dark` was given a copy of a value on line 2 and is not recalculated, so `dark` stays `true`. Line 6 prints `!true`, which is `false`. Line 3 also shows a handy pattern: `lightsOn = !lightsOn`; switches a boolean to its opposite, like flipping a light switch.

Reread 2.5.1 *Reversing a Condition with NOT* if your answer differed.

Question 2

A. The parentheses make Java evaluate `temp > 100` first, and `!` flips that result. When `temp` is above 100 the comparison is `true` and the whole expression is `false`; otherwise it is `true`.

- **B** is wrong: it does not compile. `!` applies to `temp` alone before the comparison, and `!` cannot be used on an `int`.
- **C** is wrong: `!(temp < 100)` is `true` when `temp` is **not less than** 100, for example 150. That is the opposite of what was asked for large temperatures.
- **D** is wrong: `temp != 100` means "not equal to 100." For `temp` equal to 150 it is `true`, even though 150 is greater than 100.

Reread 2.5.1 *Reversing a Condition with NOT* if you chose differently.

Question 3

It prints:

```
true
false
```

With three operands joined by `&&`, every one must be `true`.

Line	sunny	temp > 60	temp < 85	Result
3	true	68 > 60 true	68 < 85 true	true
6	true	90 > 60 true	90 < 85 false	false

On line 6, two of the three pieces are `true`, but one `false` makes the whole `&&` expression `false`.

Reread 2.5.2 *Requiring Both Conditions with AND* if your answer differed.

Question 4

C. `64 > 50` is true, and `64 % 2` is 0, so `64 % 2 == 0` is true. Both sides are true, so the `&&` is true. (Recall from Topic 1.3 that `% 2` gives 0 for an even number.)

- **A** is wrong: `51 > 50` is true, but `51 % 2` is 1, so the second side is false.
- **B** is wrong: 48 is even, but `48 > 50` is false.
- **D** is wrong: 50 is even, but `50 > 50` is false; "greater than" does not include equal.

Reread 2.5.2 *Requiring Both Conditions with AND* if you chose differently.

Question 5

It prints `Go outside.`

Step	Expression	Value
1	<code>weather.equals("rain")</code>	false (the text is "snow")
2	<code>temp < 32</code>	false (40 is not below 32)
3	<code>false false</code>	false

Both sides are false, the only case where `||` is false, so the `else` body runs. If you picked `Stay inside.`, check that you compared "snow" with "rain", not with any bad weather.

Reread 2.5.3 *Accepting Either Condition with OR* if your answer differed.

Question 6

B. Either reason alone closes the ride, so the two conditions are joined with `||`. `windSpeed > 40` is a boolean, and `isRaining` already is one.

- **A** is wrong: `&&` closes the ride only when it is windy **and** raining at the same time. A dry day with 50 mph wind would stay open.
- **C** is wrong: it does not compile. `windSpeed` is an `int`, not true or false, so it cannot be a side of `||`; and `isRaining` is a boolean, which cannot be compared with 40 using `>`. The comparison belongs next to `windSpeed`, as in choice B.
- **D** is wrong: `!(windSpeed > 40)` is true when the wind is **not** above 40, so this closes the ride on calm days.

Reread 2.5.3 *Accepting Either Condition with OR* if you chose differently.

Question 7

It prints:

```
true
false
true
```

- **Line 4**, `p && q || r`: `&&` first. `false && true` is false. Then `false || true` is true.
- **Line 5**, `p && (q || r)`: parentheses first. `true || true` is true. Then `false && true` is false.
- **Line 6**, `!p || q && !r`: `!` first, so `!p` is true and `!r` is false. Then `&&`: `q && false` is `true && false`, which is false. Then `||`: `true || false` is true.

If you got false on line 4, you grouped it as `p && (q || r)`. Java uses that grouping only when parentheses ask for it, as on line 5.

Reread 2.5.4 *Order of Evaluation* if your answer differed.

Question 8

B. `!` goes first and applies only to `c`. Then `&&` combines `b` with `!c`. Then `||` combines `a` with that result.

- **A** is wrong: it makes `||` happen before `&&`, which is the reverse of Java's order.
- **C** is wrong: it moves `!` so it flips the whole expression. In the original, `!` is directly in front of `c` and applies only to `c`.
- **D** is wrong for the same reason as A: the parentheses force `||` first.

Reread 2.5.4 *Order of Evaluation* if you chose differently.

Question 9

It prints `No fourth s`, and `substring` is never called.

Step	Expression	Value
1	<code>word.length() > 3</code> → <code>3 > 3</code>	<code>false</code>
2	<code>false && ...</code>	<code>false</code> ; the right side is skipped
3	condition is <code>false</code>	run the <code>else</code> body

`"cat"` has indexes 0 to 2, so `word.substring(3, 4)` would ask for a character past the end and throw a `StringIndexOutOfBoundsException` (Topic 1.15). The length check on the left is a guard: because it is `false`, short-circuit evaluation skips the risky call.

Reread 2.5.5 *Short-Circuit Evaluation* if your answer differed.

Question 10

C. Java always evaluates the left side first, and in C the left side is the division `10 / 0`, which crashes before `n != 0` is checked.

- **A** is wrong: `n != 0` is `false`, so `&&` short-circuits to `false` and the division is skipped.
- **B** is wrong: `n == 0` is `true`, so `||` short-circuits to `true` and the division is skipped.
- **D** is wrong: `n > 0` is `false`, so `&&` short-circuits to `false` and the division is skipped.

Reread 2.5.5 *Short-Circuit Evaluation* and its Watch Out if you chose differently.

Question 11

A. By precedence, `&&` runs before `||`, so Java groups the expression as

`(boxes != 0 && items / boxes > 5) || items > 20`.

1. `boxes != 0` is `0 != 0`, which is `false`.
2. The `&&` short-circuits: its right side, `items / boxes > 5`, is skipped, and the `&&` part is `false`.
3. The `||` now has `false` on its left, so Java evaluates its right side: `24 > 20` is `true`.
4. `false || true` is `true`.

- **B** is wrong: it stops after the `&&` part. That part is `false`, but it is only the left side of `||`, and the right side of `||` is `true`.
- **C** is wrong: the division never runs, because the `&&` short-circuits when `boxes != 0` is `false`.
- **D** is wrong: every operand of `&&` and `||` is a complete `boolean` expression, so the code compiles.

Reread 2.5.4 *Order of Evaluation* and 2.5.5 *Short-Circuit Evaluation* if you chose differently.

Question 12

It prints `Grade: C`.

Condition	Steps	Value
<code>score >= 90 score >= 88 && extraCredit</code>	Grouped as <code>score >= 90 (score >= 88 && extraCredit)</code> . <code>84 >= 90</code> is false, so Java evaluates the group on the right: <code>84 >= 88</code> is false, so the <code>&&</code> is false. <code>false false</code> is false.	false
<code>score >= 80 && !(absences > 3)</code>	<code>84 >= 80</code> is true. <code>5 > 3</code> is true, so <code>!(true)</code> is false. <code>true && false</code> is false.	false
<code>score >= 70 extraCredit</code>	<code>84 >= 70</code> is true, so <code> </code> short-circuits to true.	true

The third condition is the first true one, so grade becomes "C" and the `else` is skipped (Topic 2.4). If you answered B, you probably missed the `!`: `absences > 3` is true, and `!` flips it to false, so too many absences block the B.

Reread 2.5.1 *Reversing a Condition with NOT* and 2.5.4 *Order of Evaluation* if your answer differed.

Question 13

The error is on line 2: `||` should be `&&`. A valid month must pass both tests, at least 1 **and** at most 12. With `||`, passing either test is enough. For 15, `month >= 1` is true, so the `||` is true (and short-circuits). In fact every whole number is either at least 1 or at most 12 (numbers below 1 pass the second test), so the condition is true for every value and `Invalid month` can never print. The code compiles, so only testing reveals the bug.

Corrected code:

```

1 int month = 15;
2 if (month >= 1 && month <= 12)
3 {
4     System.out.println("Valid month");
5 }
6 else
7 {
8     System.out.println("Invalid month");
9 }

```

Now `15 >= 1` is true but `15 <= 12` is false, so `true && false` is false, and it prints `Invalid month`.

Reread 2.5.2 *Requiring Both Conditions with AND* and the second Watch Out in 2.5.3 *Accepting Either Condition with OR* if you missed it.

Question 14

Model answer: In the first segment, the outer condition `count > 0` is false, so Java skips the whole body of the outer `if`, including the inner `if`; its division is never evaluated (Topic 2.4). In the second segment, Java evaluates the left side of `&&` first. `count > 0` is false, which already makes the whole `&&` expression false, so short-circuit evaluation skips the right side and the

division never runs. Neither segment prints anything. The second does in one condition what the first does with nesting, but only because the guard `count > 0` is on the **left** of `&&`.

A complete answer must:

- say that the division `total / count` is never evaluated in either segment,
- explain that the inner `if` runs only when the outer condition is `true`, and
- name short-circuit evaluation: `&&` skips its right side when its left side is `false`.

Reread 2.5.5 *Short-Circuit Evaluation* if your answer differed.

Question 15

C. `a || b` is `true`, so at least one of them is `true`. `a && b` is `false`, so they are not both `true`. The only combinations left are `true/false` and `false/true`: exactly one is `true`.

- **A** is wrong: if both were `true`, `a && b` would be `true`.
- **B** is wrong: if both were `false`, `a || b` would be `false`.
- **D** is wrong: `a` being `false` and `b` being `true` also gives `false` for `&&` and `true` for `||`. Neither operator cares which side holds the `true`.

Reread 2.5.2 *Requiring Both Conditions with AND* and 2.5.3 *Accepting Either Condition with OR* if you chose differently.