

2.6 Comparing Boolean Expressions

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Decide whether two Boolean expressions are equivalent, prove it with a truth table, or disprove it with a single counterexample.
- Use De Morgan's laws to rewrite a negated and-expression or or-expression without the outer not.
- Predict the result of `==` and `!=` when they compare two object references, including two variables that share one object.
- Check whether a reference is null before calling a method on it.
- Explain when to use `equals` instead of `==` to compare two objects, such as two strings.

What to remember from earlier

- Relational operators give a boolean; `==` compares values for primitives and references for objects. ([Topic 2.2: Boolean Expressions](#), [2.2.5 Comparing Strings and Other Objects](#))
- `!` flips a boolean, `&&` needs both parts true, and `||` needs at least one true. ([Topic 2.5: Compound Boolean Expressions](#))
- Short-circuit evaluation skips the right side of `&&` or `||` once the left side decides the result. ([Topic 2.5: Compound Boolean Expressions](#), [2.5.5 Short-Circuit Evaluation](#))
- A reference variable holds the address of an object, or null for no object. ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.2 Reference Variables and null](#))
- Calling a method on null throws a `NullPointerException`. ([Topic 1.14: Calling Instance Methods](#), [1.14.5 Calling a Method on null](#))
- `equals` returns true when two strings have exactly the same characters. ([Topic 1.15: String Manipulation](#), [1.15.9 Comparing Strings with `equals` and `compareTo`](#))

2.6.1 Equivalent Boolean Expressions

Two students write the "did you pass?" check for a quiz app. Jordan writes `!(score < 60)`, which reads "it is not true that the score is below 60." Priya writes `score >= 60`, which reads "the score is at least 60." Their teammate asks: can we use either one, or will they sometimes disagree? Before you can swap one condition for another, you need a way to answer that question for sure.

In plain words, two conditions are interchangeable when they always give the same answer, no matter what values the variables hold. Not just for the score you happened to try, but for every score that could ever show up. Two Boolean expressions that evaluate to the same value in every possible case are called **equivalent Boolean expressions**. If you can swap one for the other anywhere in a program and nothing could ever change, they are equivalent.

That "every possible case" part has two consequences:

- **One mismatch is enough to prove two expressions are not equivalent.** A case where one expression is `true` and the other is `false` is called a **counterexample**. Find one, and you are done.

- **Matching on a few cases proves nothing.** Two expressions can agree on the values you tried and still disagree on a value you did not try. To show equivalence, you need a reason that covers every case.

KEY IDEA

Two Boolean expressions are equivalent only if they give the same value in every case. One counterexample proves they are not.

WORKED EXAMPLE Testing Jordan's and Priya's conditions

The program below evaluates both conditions for three scores and prints the two results side by side. Look at the scores chosen: one just below 60, 60 itself, and one just above.

```
1 int score = 59;
2 System.out.println(!(score < 60) + " " + (score >= 60));
3 score = 60;
4 System.out.println(!(score < 60) + " " + (score >= 60));
5 score = 61;
6 System.out.println(!(score < 60) + " " + (score >= 60));
```

```
false false
true true
true true
```

In each `println`, the left result is Jordan's `!(score < 60)` and the right result is Priya's `score >= 60`. The parentheses around `(score >= 60)` make Java finish the comparison before joining the result into the printed text with `+` (Topic 1.15).

score	score < 60	!(score < 60)	score >= 60
59	true	false	false
60	false	true	true
61	false	true	true

Three matching rows are not a proof on their own. The reason that covers every score is this: for any whole number, either it is less than 60 or it is not. "Not less than 60" means "60 or more," which is exactly `score >= 60`. So the two expressions agree for every possible `score`, and they are equivalent. The three tested scores were chosen to check the boundary at 60, which is where a mistake would show up.

This reasoning gives you a handy pattern. Putting `!` in front of a single comparison is the same as using the opposite comparison:

Expression	Equivalent without !
<code>!(a < b)</code>	<code>a >= b</code>
<code>!(a > b)</code>	<code>a <= b</code>
<code>!(a <= b)</code>	<code>a > b</code>
<code>!(a >= b)</code>	<code>a < b</code>

<code>!(a == b)</code>	<code>a != b</code>
<code>!(a != b)</code>	<code>a == b</code>

Notice that the opposite of `<` is `>=`, not `>`. The opposite of "below" has to include "equal."

Now vary the operator. Is `temp > 32 || temp == 32` equivalent to `temp >= 32`? Check around the boundary again:

```
1 int temp = 31;
2 System.out.println((temp > 32 || temp == 32) + " " + (temp >= 32));
3 temp = 32;
4 System.out.println((temp > 32 || temp == 32) + " " + (temp >= 32));
5 temp = 33;
6 System.out.println((temp > 32 || temp == 32) + " " + (temp >= 32));
```

```
false false
true true
true true
```

Both columns match at 31, 32, and 33. The reason they match everywhere: `||` is true when either part is true, so the left expression is true when `temp` is greater than 32 or equal to 32. "Greater than or equal to" is exactly what `>=` means. They are equivalent.

WATCH OUT

Flipping < to >. Students often rewrite `!(score < 60)` as `score > 60`, because "not less" sounds like "greater." Test the boundary value:

```
1 int score = 60;
2 System.out.println(!(score < 60));
3 System.out.println(score > 60);
```

```
true
false
```

At `score = 60`, the first expression is `true` (60 is not less than 60) but `score > 60` is `false`. That single counterexample proves the two are not equivalent: a student with exactly 60 would be told they failed. The fix is `score >= 60`. Whenever you rewrite a comparison, test the value right on the boundary.

PAUSE AND TRY

Question 1 Multiple choice

Which expression is equivalent to `!(age > 17)` for every `int` value of `age`?

- A. `age < 17`
- B. `age <= 17`
- C. `age >= 17`
- D. `age != 17`

Question 2 Short answer

Maya compares `!(points < 10)` with `points > 10`. She tries `points = 4` (both are false) and `points = 15` (both are true), and concludes they are equivalent. Is she right? Explain.

2.6.2 Proving Equivalence with Truth Tables

In 2.6.1 you proved equivalence by reasoning about what the comparison means. That works, but it can get slippery when an expression combines several `boolean` variables with `&&`, `||`, and `!`. You need a method that checks every case mechanically, with no clever insight required.

Here is the good news: a `boolean` variable has only two possible values, `true` and `false`. So an expression built from a couple of `boolean` variables has only a few possible situations. With one variable there are 2. With two variables, each of the 2 values for the first can pair with each of the 2 values for the second, giving $2 \times 2 = 4$. With three variables there are $2 \times 2 \times 2 = 8$. You can simply write down every situation and check them all.

A **truth table** is a table with one row for every possible combination of `true` and `false` for the variables, and a column showing the value of each expression in that row. If two expressions have identical columns in every row, they are equivalent, because the table covers every case there is. If any single row differs, that row is a counterexample.

KEY IDEA

A truth table lists every combination of values, so matching columns in every row proves two expressions equivalent, and one differing row proves they are not.

A reliable way to list the rows for two variables `a` and `b` is to start with `a` true for the first two rows and false for the last two, and let `b` alternate true, false, true, false. That order guarantees no combination is missed or repeated.

WORKED EXAMPLE A coupon rule that simplifies

A store's checkout code gives a discount when `member || (member && coupon)` is true, where `member` and `coupon` are `boolean` variables. Is that the same as just `member`? Build the table one column at a time. Work out the parentheses first, as always.

<code>member</code>	<code>coupon</code>	<code>member && coupon</code>	<code>member (member && coupon)</code>	<code>member</code>
true	true	true	true	true
true	false	false	true	true
false	true	false	false	false
false	false	false	false	false

Row by row:

- **Rows 1 and 2:** `member` is true, so the `||` is true no matter what the right side is (one true part is enough).
- **Row 3:** `member && coupon` is false && true, which is false. Then `false || false` is false.

- **Row 4:** both parts are `false`, so the result is `false`.

The two right columns are identical in all four rows, so the expressions are equivalent. The coupon never matters: anyone who is a member already gets the discount, and a non-member cannot satisfy `member && coupon`. The simpler `member` is easier to read and does the same job.

You can have Java confirm the table. This program sets each of the four combinations in turn and prints both expressions:

```
1 boolean member = true;
2 boolean coupon = true;
3 System.out.println((member || (member && coupon)) + " " + member);
4 coupon = false;
5 System.out.println((member || (member && coupon)) + " " + member);
6 member = false;
7 coupon = true;
8 System.out.println((member || (member && coupon)) + " " + member);
9 coupon = false;
10 System.out.println((member || (member && coupon)) + " " + member);
```

```
true true
true true
false false
false false
```

Each printed line is one row of the table, in the same order.

Now vary the expressions. Is `!(a && b)` equivalent to `!a && !b`? Many students guess yes, because it looks like the `!` was just "handed out" to each part. Build the table:

a	b	a && b	!(a && b)	!a	!b	!a && !b
true	true	true	false	false	false	false
true	false	false	true	false	true	false
false	true	false	true	true	false	false
false	false	false	true	true	true	true

Compare the `!(a && b)` column with the `!a && !b` column. Rows 2 and 3 differ. For example, in row 2, `a && b` is false, so `!(a && b)` is true; but `!a` is false, so `!a && !b` is false. They are **not** equivalent. Java agrees:

```
1 boolean a = true;
2 boolean b = true;
3 System.out.println(!(a && b) + " " + (!a && !b));
4 b = false;
5 System.out.println(!(a && b) + " " + (!a && !b));
6 a = false;
7 b = true;
8 System.out.println(!(a && b) + " " + (!a && !b));
9 b = false;
10 System.out.println(!(a && b) + " " + (!a && !b));
```

```
false false
true  false
true  false
true  true
```

The second and third lines show the two rows where the expressions disagree. You will see the correct way to move a `!` inside parentheses in 2.6.3.

WATCH OUT

Checking only some rows. A student checks `!(a && b)` and `!a && !b` using only rows 1 and 4, where both columns match, and declares them equivalent. A truth table is a proof only when it is complete: every combination, 4 rows for two variables. Skipping rows turns it back into "I tried a few cases," which proves nothing. Count your rows before you decide: 2 for one variable, 4 for two, 8 for three.

PAUSE AND TRY

Question 3 Tracing

Complete a truth table for `a || b` and `a != b`, where `a` and `b` are `boolean` variables. (Remember from Topic 2.2 that `!=` on two `boolean` values is `true` when they are different.) Are the expressions equivalent?

2.6.3 De Morgan's Laws

Programs often need the opposite of a compound condition. A movie theater charges a teen price when `age >= 13 && age <= 19`. To charge everyone else the regular price, you could write `!(age >= 13 && age <= 19)`, but a `!` wrapped around a whole compound expression is hard to read and easy to get wrong, as the truth table at the end of 2.6.2 showed. You need a dependable rule for moving that `!` inside.

Think about what the words mean. "It is **not** true that I have **both** a ticket **and** a snack" means "I am missing the ticket, **or** I am missing the snack (or both)." And "It is **not** true that it is Saturday **or** Sunday" means "it is **not** Saturday **and** it is **not** Sunday." When the "not" moves inside, two things happen: each part gets its own "not," and the joining word flips between "and" and "or."

These two rules are called **De Morgan's laws**:

Original	Equivalent expression
<code>!(a && b)</code>	<code>!a !b</code>
<code>!(a b)</code>	<code>!a && !b</code>

To apply either law: negate each part, and change `&&` to `||` or `||` to `&&`. Both steps are required; doing only one of them gives a different expression.

KEY IDEA

De Morgan's laws: $!(a \ \&\& \ b)$ is equivalent to $!a \ || \ !b$, and $!(a \ || \ b)$ is equivalent to $!a \ \&\& \ !b$. Negate each part and flip the operator.

Why should you believe these rules? Because a truth table proves them, using the method from 2.6.2.

WORKED EXAMPLE Proving both laws with one truth table

Each pair of columns below should match in every row.

a	b	!(a && b)	!a !b	!(a b)	!a && !b
true	true	false	false	false	false
true	false	true	true	false	false
false	true	true	true	false	false
false	false	true	true	true	true

Take row 2 as a sample. $a \ \&\& \ b$ is $\text{true} \ \&\& \ \text{false}$, which is false , so $!(a \ \&\& \ b)$ is true . On the other side, $!a$ is false and $!b$ is true , and $\text{false} \ || \ \text{true}$ is true . They match. For the second law in the same row, $a \ || \ b$ is true , so $!(a \ || \ b)$ is false ; and $!a \ \&\& \ !b$ is $\text{false} \ \&\& \ \text{true}$, which is false . They match too. Every row matches, so both laws hold in every case.

This program evaluates all four expressions for each row, printing the first law's pair, a gap, and then the second law's pair:

```

1  boolean a = true;
2  boolean b = true;
3  System.out.println(!(a && b) + " " + (!a || !b) + " " + !(a || b) + " " + (!a
   && !b));
4  b = false;
5  System.out.println(!(a && b) + " " + (!a || !b) + " " + !(a || b) + " " + (!a
   && !b));
6  a = false;
7  b = true;
8  System.out.println(!(a && b) + " " + (!a || !b) + " " + !(a || b) + " " + (!a
   && !b));
9  b = false;
10 System.out.println(!(a && b) + " " + (!a || !b) + " " + !(a || b) + " " + (!a
   && !b));

```

```

false false    false false
true true     false false
true true     false false
true true     true  true

```

In real code, the parts are usually comparisons. Negating a part means using the opposite comparison from the table in 2.6.1.

WORKED EXAMPLE Who pays the regular price?

Apply the first law to `!(age >= 13 && age <= 19)`:

1. The parts are `age >= 13` and `age <= 19`, joined by `&&`.
2. Negate each part: `!(age >= 13)` becomes `age < 13`, and `!(age <= 19)` becomes `age > 19`.
3. Flip `&&` to `||`.

The result is `age < 13 || age > 19`: younger than a teen or older than a teen. Test both forms on each side of both boundaries:

```
1 int age = 12;
2 System.out.println(!(age >= 13 && age <= 19) + " " + (age < 13 || age > 19));
3 age = 13;
4 System.out.println(!(age >= 13 && age <= 19) + " " + (age < 13 || age > 19));
5 age = 19;
6 System.out.println(!(age >= 13 && age <= 19) + " " + (age < 13 || age > 19));
7 age = 20;
8 System.out.println(!(age >= 13 && age <= 19) + " " + (age < 13 || age > 19));
```

```
true true
false false
false false
true true
```

age	Is a teen?	<code>!(age >= 13 && age <= 19)</code>	<code>age < 13 age > 19</code>
12	No	true	true
13	Yes	false	false
19	Yes	false	false
20	No	true	true

The two columns agree, including on the boundary values 13 and 19, just as the law promises.

Now vary the operator that is negated. Days of the week are numbered 1 to 7, with 6 and 7 for Saturday and Sunday. A weekday check written as `!(day == 6 || day == 7)` uses `||`, so the second law applies: negate each part (`day != 6`, `day != 7`) and flip `||` to `&&`, giving `day != 6 && day != 7`.

```
1 int day = 3;
2 System.out.println(!(day == 6 || day == 7) + " " + (day != 6 && day != 7));
3 day = 6;
4 System.out.println(!(day == 6 || day == 7) + " " + (day != 6 && day != 7));
5 day = 7;
6 System.out.println(!(day == 6 || day == 7) + " " + (day != 6 && day != 7));
```

```
true true
false false
false false
```

For `day = 3`, both are `true`; for 6 and 7, both are `false`. Notice why `&&` is right here: a weekday must be not Saturday **and** not Sunday. With `||`, every day would pass, since any day is either not 6 or not 7.

WATCH OUT

Negating the parts but forgetting to flip the operator. A student rewrites

`!(age >= 13 && age <= 19)` as `age < 13 && age > 19`:

```
1 int age = 12;
2 System.out.println(!(age >= 13 && age <= 19));
3 System.out.println(age < 13 && age > 19);
```

```
true
false
```

For a 12-year-old, the original is `true` but the rewrite is `false`. In fact, `age < 13 && age > 19` is `false` for every age, since no number is both below 13 and above 19. Students make this slip because negating the comparisons feels like the whole job. The fix is to flip `&&` to `||`:

`age < 13 || age > 19`.

A second trap is leaving out the parentheses. `!` is applied before anything else (Topic 2.5), so `!age >= 13` would try to apply `!` to the `int` variable `age` alone:

```
1 int age = 15;
2 boolean notTeen = !age >= 13;
```

Java refuses to compile it, with a message like `bad operand type int for unary operator '!'` (the exact wording can vary between Java versions). `!` only works on a `boolean`, so the comparison must be in parentheses: `!(age >= 13)`.

PAUSE AND TRY

Question 4 Multiple choice

Which expression is equivalent to `!(x > 0 || y > 0)`, where `x` and `y` are `int` variables?

- A. `x <= 0 && y <= 0`
- B. `x <= 0 || y <= 0`
- C. `x < 0 && y < 0`
- D. `!x > 0 && !y > 0`

Question 5 Debugging

A store is open from hour 9 through hour 17. The programmer wanted the opposite of `hour >= 9 && hour <= 17` to print `Closed`, but this code prints `Open` when `hour` is 20. Find and fix the error.

```
1 int hour = 20;
2 if (hour < 9 && hour > 17)
3 {
4     System.out.println("Closed");
5 }
6 else
7 {
8     System.out.println("Open");
9 }
```

2.6.4 Aliases and Reference Comparison

So far every comparison was between numbers or `boolean` values. Programs also compare objects: is this the same `Scanner` I was using, or is this the same `String` object? Topic 2.2 told you that `==` on reference types compares references, not contents. This subsection shows exactly what that means.

Recall from Topic 1.13 that a reference variable does not hold an object. It holds a **reference**, which you can picture as the object's address. Picture it as a sticky note with an address written on it. Assigning one reference variable to another, as in `second = first;`, copies the address onto a second sticky note. It does not build a second object. Both notes now lead to the same object. When two variables hold references to the same object, they are called **aliases** of that object, the way one person can go by two names.

When `==` compares two reference variables, it asks: **do these two variables hold the same address?** In other words, do they refer to the very same object? `!=` asks the opposite: do they refer to different objects? Neither operator looks inside the objects at all.

KEY IDEA

For references, `a == b` is true only when `a` and `b` refer to the same object. `a != b` is true when they refer to different objects, even if those objects hold identical data.

WORKED EXAMPLE Same object or just the same text?

Count the `new`s (Topic 1.13) to count the objects, then predict each comparison.

```
1 String first = new String("Lion");
2 String second = first;
3 String third = new String("Lion");
4 System.out.println(first == second);
5 System.out.println(first == third);
6 System.out.println(first != third);
```

Line 1 builds a `String` object holding `Lion`; call its address A. Line 2 has no `new`, so it copies address A into `second`. Line 3 has `new`, so it builds a second, separate object, also holding `Lion`, at a different address B.

Variable	Holds	Object at that address
first	address A	String "Lion"
second	address A	(the same object)
third	address B	another String "Lion"

```
true
false
true
```

- Line 4: `first` and `second` both hold address A, so `first == second` is `true`. They are aliases.
- Line 5: `first` holds A and `third` holds B. The text inside is identical, but the addresses differ, so `first == third` is `false`.
- Line 6: since they refer to different objects, `first != third` is `true`.

Being aliases is not permanent. Assigning a new reference to one of the variables changes what it refers to:

```
1 String first = new String("Lion");
2 String second = first;
3 System.out.println(first == second);
4 second = new String("Lion");
5 System.out.println(first == second);
6 System.out.println(second);
```

```
true
false
Lion
```

After line 2, `first == second` is `true`. Line 4 builds a brand-new object and stores its address in `second`, so `second` no longer refers to the object that `first` refers to. Line 5 prints `false`, even though line 6 shows that `second` still holds the text `Lion`.

Aliases matter because whatever you do through one variable happens to the one shared object. `String` objects cannot be changed (Topic 1.15), so with strings you rarely notice. A `Scanner` makes it visible, since reading a value moves the `Scanner` forward through its input:

SharedScanner.java

```
1 import java.util.Scanner;
2
3 public class SharedScanner
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner("7 8 9");
8         Scanner sameReader = reader;
9         System.out.println(reader == sameReader);
10        System.out.println(reader.nextInt());
11        System.out.println(sameReader.nextInt());
12    }
13 }
```

```
true
7
8
```

Line 8 makes `sameReader` an alias of `reader`, so line 9 prints `true`. Line 10 reads 7 through `reader`. Line 11 reads through `sameReader`, but it is the same `Scanner` object, which has already moved past the 7, so it reads 8. There is only one `Scanner`, with one position in its input.

WATCH OUT

Using `==` to compare the text of two strings. A student checks a password like this:

```
1 String saved = new String("pass");
2 String typed = new String("pass");
3 if (typed == saved)
4 {
5     System.out.println("Welcome");
6 }
7 else
8 {
9     System.out.println("Wrong password");
10 }
```

```
Wrong password
```

Both strings hold `pass`, so the student expects `Welcome`. But each `new` built a separate object, so `typed` and `saved` hold different addresses, and `typed == saved` is `false`. Students make this mistake because `==` works for comparing numbers, so it seems natural for text too. For objects, `==` answers "same object?", not "same contents?". The right tool for comparing contents is the `equals` method, coming up in 2.6.6.

PAUSE AND TRY

Question 6 Tracing

What does this code print?

```
1 String p = new String("pear");
2 String q = p;
3 String r = new String("pear");
4 System.out.println(q == p);
5 System.out.println(r == p);
6 System.out.println(q != r);
7 p = r;
8 System.out.println(p == r);
9 System.out.println(q == p);
```

Question 7 Multiple choice

After `Scanner left = new Scanner("1 2 3");` and then `Scanner right = left;`, which statement is true?

- A. There are two `Scanner` objects, and `left == right` is false.
- B. There is one `Scanner` object, and `left == right` is true.
- C. There is one `Scanner` object, but `left == right` is false because they are different variables.
- D. The second line does not compile, because two variables cannot refer to the same object.

2.6.5 Checking for null

In Topic 1.14 you saw that calling a method on a variable holding `null` crashes the program with a `NullPointerException`. In real programs a variable is sometimes `null` on purpose: a player has not picked a nickname yet, or a search found nothing. How can a program find out, before calling a method, whether a variable refers to an object at all?

Recall that `null` is a special value meaning "this variable refers to no object" (Topic 1.13). Since `==` and `!=` compare what reference variables hold, they can compare a reference with `null` too:

Expression	true when...
<code>name == null</code>	name refers to no object
<code>name != null</code>	name refers to an actual object

Comparing with `null` never calls a method, so it is always safe, even when the variable is `null`. That makes it the right first step before using an object.

KEY IDEA

Use `== null` or `!= null` to check whether a reference variable refers to an object before calling a method on it.

WORKED EXAMPLE A nickname that may not exist yet

The same if-else statement runs twice, once with `nickname` holding `null` and once after it is given an object.

```
1 String nickname = null;
2 if (nickname == null)
3 {
4     System.out.println("No nickname yet");
5 }
6 else
7 {
8     System.out.println(nickname.length());
9 }
10 nickname = "Sky";
11 if (nickname == null)
12 {
13     System.out.println("No nickname yet");
14 }
15 else
16 {
17     System.out.println(nickname.length());
18 }
```

```
No nickname yet
3
```

- Line 2: `nickname == null` is true, so the `if` body prints `No nickname yet`. The `else` body, which calls `length()`, is skipped, so there is no crash.
- Line 10: `nickname` now refers to the `String` `"Sky"`.
- Line 11: `nickname == null` is false, so the `else` body runs and prints the length, `3`.

Often a program wants to check for `null` and test the object in a single condition. Short-circuit evaluation from Topic 2.5 makes that safe: with `&&`, if the left side is `false`, Java never evaluates the right side.

```
1 String nickname = null;
2 if (nickname != null && nickname.length() > 3)
3 {
4     System.out.println("Long nickname");
5 }
6 else
7 {
8     System.out.println("No long nickname");
9 }
10 nickname = "Skyler";
11 if (nickname != null && nickname.length() > 3)
12 {
13     System.out.println("Long nickname");
14 }
15 else
16 {
17     System.out.println("No long nickname");
18 }
```

```
No long nickname
Long nickname
```

On line 2, `nickname` is `null`, so `nickname != null` is `false`. Because the left side of `&&` is `false`, the whole condition must be `false`, and Java skips `nickname.length() > 3` entirely. The method is never called on `null`, and the `else` prints `No long nickname`. Line 10 stores `"Skyler"`. On line 11 the null check is `true`, so Java evaluates `nickname.length() > 3`, which is `6 > 3`, `true`. Placing the null check on the left of `&&` like this is called a **guard**.

One more case to keep straight: an empty string is not `null`. `""` is a real `String` object that happens to have no characters.

```
1 String blank = "";
2 System.out.println(blank == null);
3 System.out.println(blank.length());
```

```
false
0
```

`blank` refers to an object, so `blank == null` is `false` and `blank.length()` safely returns `0`.

WATCH OUT

Putting the guard on the wrong side. Short-circuiting only helps if the null check comes **first**:

```
1 String nickname = null;
2 System.out.println("Checking");
3 if (nickname.length() > 3 && nickname != null)
4 {
5     System.out.println("Long nickname");
6 }
```

```
Checking
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because ...
```

Java evaluates the left side of `&&` first, so it calls `nickname.length()` while `nickname` is `null`, and the program crashes before it ever reaches `nickname != null`. (The exact message varies between Java versions.) Students write it this way because in math class, `p` and `q` means the same as `q` and `p`. In Java the two sides of `&&` mean the same thing logically, but the order decides what gets evaluated. The fix is to swap them:
`nickname != null && nickname.length() > 3`.

PAUSE AND TRY

Question 8 Multiple choice

After `String title = null;` and `String blank = "";`, which line prints `true`?

- A. `System.out.println(title == null);`
- B. `System.out.println(blank == null);`
- C. `System.out.println(title != null);`
- D. `System.out.println(title.length() == 0);`

Question 9 Debugging

This code is meant to print `No city` when `city` is `null`. Instead it throws a `NullPointerException`. Find and fix the error.

```
1 String city = null;
2 if (city != null || city.length() > 0)
3 {
4     System.out.println("City entered");
5 }
6 else
7 {
8     System.out.println("No city");
9 }
```

2.6.6 Comparing Objects with equals

The password check in 2.6.4 failed because `==` asks "same object?" when the program needed "same contents?". For two `String` objects, "same contents" means the same characters. For another kind of object, it might mean something else: two dates might count as the same if their day, month, and year match. Each class gets to decide what "equal" means for its own objects.

A class decides this by providing an **equals method**: a method that returns `true` when two objects should count as equivalent, according to rules the class sets. Those rules almost always compare the objects' attributes, the data each object stores (Topic 1.12). `String` has an `equals` method (Topic 1.15) whose rule is: two strings are equal when they have exactly the same characters in the same order, with the same capitalization. So for strings:

Comparison	Question it answers	true when...
<code>a == b</code>	Same object?	<code>a</code> and <code>b</code> hold the same reference
<code>a.equals(b)</code>	Same contents?	the two strings have the same characters

Writing an `equals` method for a class of your own is not part of this course. You only need to call `equals` methods that classes already provide, mainly the one in `String`.

KEY IDEA

Use `==` to ask whether two references refer to the same object. Use `equals` to ask whether two objects hold equivalent data, as the class defines it. To compare the text of two strings, always use `equals`.

WORKED EXAMPLE Three variables, two objects

`saved` and `typed` are separate objects holding the same characters, and `copy` is an alias of `saved`.

```

1 String saved = new String("Tiger42");
2 String typed = new String("Tiger42");
3 String copy = saved;
4 System.out.println(typed == saved);
5 System.out.println(typed.equals(saved));
6 System.out.println(copy == saved);
7 System.out.println(copy.equals(saved));
8 System.out.println(typed.equals("tiger42"));

```

```

false
true
true
true
false

```

Line	Comparison	Reasoning	Result
4	<code>typed == saved</code>	Two news: different objects	false
5	<code>typed.equals(saved)</code>	Same characters	true
6	<code>copy == saved</code>	Aliases: same object	true
7	<code>copy.equals(saved)</code>	Same object, so certainly the same characters	true
8	<code>typed.equals("tiger42")</code>	T and t differ	false

Lines 6 and 7 show that when two variables are aliases, `==` and `equals` both say `true`. Lines 4 and 5 show where they part ways: different objects with the same data.

Using `equals` fixes the password check from 2.6.4. Only the condition changes:

```

1 String saved = new String("pass");
2 String typed = new String("pass");
3 if (typed.equals(saved))
4 {
5     System.out.println("Welcome");
6 }
7 else
8 {
9     System.out.println("Wrong password");
10 }

```

```
Welcome
```

Now vary how the string was made. A string built while the program runs, for example with `+`, is compared by its characters just the same:

```

1 String start = new String("ice");
2 String word = start + "cream";
3 System.out.println(word);
4 System.out.println(word.equals("icecream"));
5 System.out.println(word.equals("ice cream"));

```

```
icecream  
true  
false
```

`equals` does not care how or where the string was made, only what characters it holds. `"ice cream"` has a space, so it does not match.

What if the argument is `null`? `saved.equals(typed)` with `typed` holding `null` simply returns `false`, because there is no object to match. The method is called on `saved`, which is a real object, so nothing crashes:

```
1 String saved = new String("Tiger42");  
2 String typed = null;  
3 System.out.println(saved.equals(typed));
```

```
false
```

You may notice that `==` sometimes gives `true` for two separate variables assigned the same literal, such as `String first = "hi";` and `String second = "hi";`. That happens because Java may reuse one object for identical literals in a program, so the two variables end up as aliases. It is a detail of how Java stores literals, not a comparison of text, and a string built with `new` or put together while the program runs will not necessarily be reused. Never rely on it: compare text with `equals`.

WATCH OUT

Calling `equals` on a variable that holds `null`. `equals` is a method, so the rule from 2.6.5 applies to the variable before the dot:

```
1 String saved = null;  
2 System.out.println("Comparing");  
3 System.out.println(saved.equals("Tiger42"));
```

```
Comparing  
Exception in thread "main" java.lang.NullPointerException: Cannot invoke  
"String.equals(Object)" because ...
```

`saved` is `null`, so there is no object to run `equals` on, and line 3 throws a `NullPointerException` (the exact message varies between Java versions). Students expect `false`, because the strings are obviously not equal, but Java never gets as far as comparing. The fix is a guard: `saved != null && saved.equals("Tiger42")`. Or, when you know which variable is never `null`, call `equals` on that one: `"Tiger42".equals(saved)` returns `false` safely.

PAUSE AND TRY

Question 10 Multiple choice

After `String first = new String("Ada");` and `String second = new String("Ada");`, what are the values of `first == second` and `first.equals(second)`?

- A. `true` and `true`
- B. `false` and `true`
- C. `true` and `false`
- D. `false` and `false`

Question 11 Short answer

Lin says, "`==` and `equals` do the same thing for strings. I compared two aliases and both gave `true`." Explain what is wrong with Lin's conclusion, and describe a situation where the two give different results.

Putting it together

A door lock app stores a secret code and checks codes people enter. A code can also be missing (`null`) if nothing was entered. The program accepts a code when `code != null && code.equals(savedCode)`. The programmer wants an if statement that tests for **rejection** first, so the rejection message appears at the top. The rejection condition is the negation:

```
!(code != null && code.equals(savedCode))
```

Apply De Morgan's first law: negate each part and flip `&&` to `||`.

Part	Negated
<code>code != null</code>	<code>code == null</code>
<code>code.equals(savedCode)</code>	<code>!code.equals(savedCode)</code>

The rejection condition becomes `code == null || !code.equals(savedCode)`. That condition is also safe to evaluate when `code` is `null`: the left side `code == null` is then `true`, so `||` short-circuits and never calls `equals` on `null`.

WORKED EXAMPLE Checking three entered codes

The program compares references and contents, then runs the rejection test on three codes: one that matches, one with different capitalization, and one that is missing.

```

1 String savedCode = new String("RX42");
2 String typedCode = new String("RX42");
3 String wrongCode = new String("rx42");
4 String missingCode = null;
5 System.out.println(typedCode == savedCode);
6 System.out.println(typedCode.equals(savedCode));
7 if (typedCode == null || !typedCode.equals(savedCode))
8 {
9     System.out.println("typedCode rejected");
10 }
11 else
12 {
13     System.out.println("typedCode accepted");
14 }
15 if (wrongCode == null || !wrongCode.equals(savedCode))
16 {
17     System.out.println("wrongCode rejected");
18 }
19 else
20 {
21     System.out.println("wrongCode accepted");
22 }
23 if (missingCode == null || !missingCode.equals(savedCode))
24 {
25     System.out.println("missingCode rejected");
26 }
27 else
28 {
29     System.out.println("missingCode accepted");
30 }

```

```

false
true
typedCode accepted
wrongCode rejected
missingCode rejected

```

Lines	What happens	Printed
1-4	Three separate String objects are built; missingCode holds null	
5	typedCode and savedCode are different objects	false
6	Same characters, RX42	true
7	typedCode == null is false, so check the right side: equals is true, !true is false. false false is false	typedCode accepted
15	wrongCode == null is false; rx42 vs RX42 differ in case, so equals is false and !false is true. Result true	wrongCode rejected
23	missingCode == null is true, so short-circuits: equals is never called	missingCode rejected

Every idea from the chapter shows up: `==` gave `false` for two objects with the same text, `equals` compared characters, the `null` check prevented a crash, and De Morgan's law turned the accept condition into an equivalent reject condition.

Vocabulary

Term	Definition	Example
equivalent Boolean expressions	Two Boolean expressions that evaluate to the same value in every possible case	<code>!(score < 60)</code> and <code>score >= 60</code>
counterexample	One case where two expressions give different values, proving they are not equivalent	<code>score = 60</code> for <code>!(score < 60)</code> vs <code>score > 60</code>
truth table	A table with one row for every combination of values, showing each expression's value	4 rows for <code>a</code> and <code>b</code>
De Morgan's laws	Rules for moving <code>!</code> inside parentheses: negate each part and flip <code>&&</code> and <code> </code>	<code>!(a && b)</code> is <code>!a !b</code>
alias	One of two or more variables that hold references to the same object	<code>String b = a;</code>
null	A value meaning a reference variable refers to no object	<code>nickname == null</code>
guard	A null check on the left of <code>&&</code> that stops a method call on <code>null</code>	<code>name != null && name.length() > 3</code>
equals method	A method a class provides to decide whether two of its objects count as equivalent	<code>typed.equals(saved)</code>

Chapter summary

- Two Boolean expressions are equivalent when they give the same value in every possible case. One counterexample proves they are not; agreeing on a few cases proves nothing.
- A truth table lists every combination of values (4 rows for two variables) and proves or disproves equivalence completely.
- De Morgan's laws: `!(a && b)` is equivalent to `!a || !b`, and `!(a || b)` is equivalent to `!a && !b`. Negate each part, flip the operator, and remember the opposite of `<` is `>=`.
- For references, `==` and `!=` ask whether two variables refer to the same object. Aliases are `==`; separate objects are not, even with identical data.
- Compare a reference with `null` using `==` or `!=` before calling a method on it, and put that guard on the left of `&&`.

- Classes define an `equals` method to say when two objects are equivalent, usually by comparing attributes. Compare the text of strings with `equals`, never `==`.

Review questions

Question 12 Multiple choice

A thermostat should report "comfortable" exactly when `!(temp < 50 || temp > 90)` is true. Which expression is equivalent?

- A. `temp >= 50 || temp <= 90`
- B. `temp > 50 && temp < 90`
- C. `temp >= 50 && temp <= 90`
- D. `temp < 50 && temp > 90`

Question 13 Tracing

Use a truth table to decide whether `!(a && !b)` is equivalent to `!a || b`, where `a` and `b` are `boolean` variables.

Question 14 Tracing

What does this code print?

```
1 String a = new String("map");
2 String b = a;
3 String c = new String("map");
4 b = c;
5 System.out.println(a == b);
6 System.out.println(b == c);
7 System.out.println(a.equals(c));
8 System.out.println(a != c);
```

Question 15 Debugging

This code should print `Access granted` when `entered` has the same characters as `stored`, but it prints `Access denied`. Find and fix the error.

```
1 String stored = new String("blue");
2 String entered = new String("blue");
3 if (entered != stored)
4 {
5     System.out.println("Access denied");
6 }
7 else
8 {
9     System.out.println("Access granted");
10 }
```

Question 16 Multiple choice

After `String a = new String("go");` and `String d = null;`, evaluating which expression throws a `NullPointerException`?

- A. `d == null || d.equals(a)`

- B. `d != null && d.equals(a)`
- C. `a.equals(d)`
- D. `d.equals(a)`

Question 17 Short answer

Sam claims `!(x == 1 && y == 2)` is equivalent to `x != 1 && y != 2`, where `x` and `y` are `int` variables. Give a counterexample that proves Sam wrong, show both values, and state the correct equivalent expression.

Reflection

☐ I can explain what makes two Boolean expressions equivalent and use one counterexample to show two are not. Reread: 2.6.1 Equivalent Boolean Expressions

☐ I can build a complete truth table and use it to prove or disprove equivalence. Reread: 2.6.2

Proving Equivalence with Truth Tables

☐ I can apply De Morgan's laws to rewrite a negated `&&` or `||` expression, including comparisons.

Reread: 2.6.3 De Morgan's Laws

☐ I can predict `==` and `!=` for two references, including aliases and separate objects with the same data. Reread: 2.6.4 Aliases and Reference Comparison

☐ I can check a reference against null and order a guard so no method is called on null. Reread: 2.6.5 Checking for null

☐ I can explain when to use `equals` instead of `==` and what a class `equals` method decides.

Reread: 2.6.6 Comparing Objects with `equals`

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. `!(age > 17)` is true when `age` is not greater than 17, which means `age` is 17 or less: `age <= 17`. The opposite of `>` is `<=`.

- **A** is wrong because it leaves out 17. At `age = 17`, `!(age > 17)` is true but `age < 17` is false. This is the "opposite of greater is less" slip.
- **C** is wrong because it reverses the direction. At `age = 18`, the original is false but `age >= 17` is true.
- **D** is wrong because at `age = 18` the original is false but `age != 17` is true. It only rules out one value.

Reread 2.6.1 *Equivalent Boolean Expressions* if you chose differently.

Question 2

Model answer: No. Matching on two values does not prove equivalence, because the expressions must agree in every case. At `points = 10`, `!(points < 10)` is true (10 is not less than 10), but `points > 10` is false. That one counterexample proves they are not equivalent. The equivalent expression is `points >= 10`.

A complete answer:

- says agreeing on some values is not enough, since equivalence means every case;
- gives the counterexample `points = 10` with both results.

Reread 2.6.1 *Equivalent Boolean Expressions* if your reasoning differed.

Question 3

a	b	a b	a != b
true	true	true	false
true	false	true	true
false	true	true	true
false	false	false	false

Not equivalent. Row 1 differs: when both are true, `a || b` is true (both parts count), but `a != b` is false because the two values are the same. One differing row is a counterexample. The other three rows match, which is exactly why checking only some rows can fool you.

Reread 2.6.2 *Proving Equivalence with Truth Tables* if your table differed.

Question 4

A. By De Morgan's second law, `!(p || q)` is `!p && !q`. Negating `x > 0` gives `x <= 0`, negating `y > 0` gives `y <= 0`, and `||` flips to `&&`.

- **B** is wrong because it negates the parts but keeps `||`. For `x = 5`, `y = 0`, the original is false (since `x > 0` is true) but B is true.
- **C** is wrong because the opposite of `>` is `<=`, not `<`. For `x = 0`, `y = 0`, the original is true but C is false.

- **D** does not compile. Without parentheses, `!` applies to the `int` variable `x` alone, and `!` works only on `boolean` values.

Reread 2.6.3 *De Morgan's Laws* if you chose differently.

Question 5

Error: line 2 negated each comparison but kept `&&`. The condition `hour < 9 && hour > 17` asks for an hour that is both before 9 and after 17, which no hour can be, so it is always `false` and the `else` always runs. For 20, it prints `open`.

Fix: apply De Morgan's law fully by also flipping `&&` to `||`:

```
1 if (hour < 9 || hour > 17)
```

Now for `hour = 20`, `hour > 17` is `true`, so the condition is `true` and the program prints `closed`.

Reread 2.6.3 *De Morgan's Laws* if you did not spot the operator.

Question 6

```
true
false
true
true
false
```

After line	p	q	r	Printed
1	address A			
2	address A	address A		
3	address A	address A	address B	
4	address A	address A	address B	true
5	address A	address A	address B	false
6	address A	address A	address B	true
7	address B	address A	address B	
8	address B	address A	address B	true
9	address B	address A	address B	false

Line 7 copies `r`'s address into `p`, so now `p` and `r` are aliases. `q` still refers to the first object, so on line 9, `q == p` is `false`. Both objects hold `pear` the whole time, which `==` never looks at.

Reread 2.6.4 *Aliases and Reference Comparison* if your trace differed.

Question 7

B. Only one `new` runs, so there is one `Scanner` object. The second line copies its reference into `right`, making the two variables aliases, and `==` is `true` because they hold the same reference.

- **A** is wrong because assignment copies a reference; it never builds an object. Only `new` does.
- **C** is wrong because `==` compares what the variables hold (the references), not the variable names.
- **D** is wrong because aliases are allowed and common; this is ordinary assignment.

Reread 2.6.4 *Aliases and Reference Comparison* if you chose differently.

Question 8

A. `title` holds `null`, so `title == null` is `true`.

- **B** prints `false`: `""` is an actual `String` object with no characters, so `blank` is not `null`.
- **C** prints `false`: `title` refers to no object, so `title != null` is `false`.
- **D** prints nothing; it throws a `NullPointerException`, because it calls `length()` on `null`.

Reread 2.6.5 *Checking for null* if you chose differently.

Question 9

Error: line 2 uses `||`. When `city` is `null`, `city != null` is `false`, and with `||` a `false` left side does not decide the result, so Java goes on to evaluate `city.length() > 0`. That calls a method on `null` and throws a `NullPointerException`.

Fix: use `&&`, so that a `false` null check short-circuits the rest:

```
1 if (city != null && city.length() > 0)
```

Now the left side is `false`, Java skips `city.length()`, and the `else` prints `No city`. A guard only protects the right side when its operator stops at the guard's result: `&&` stops on `false`.

Reread 2.6.5 *Checking for null* if you looked only at the order of the two sides.

Question 10

B. Two `news` build two separate objects, so `first == second` is `false`. Both hold the characters `Ada`, so `first.equals(second)` is `true`.

- **A** is wrong because `==` compares references, and these are different objects, even though their text matches.
- **C** reverses the two operators: it treats `==` as "same text" and `equals` as "same object".
- **D** is wrong because `equals` compares characters, and the characters are identical.

Reread 2.6.6 *Comparing Objects with equals* if you chose differently.

Question 11

Model answer: Lin tested only one kind of case. For aliases, both are `true`, because the variables refer to the same object, and an object certainly has the same characters as itself. But `==` asks whether two references refer to the same object, while `equals` asks whether the strings hold the same characters. They differ for two separate objects with the same text, such as `new String("Ada")` built twice: `==` is `false` and `equals` is `true`.

A complete answer:

- says `==` compares references (same object?) and `equals` compares characters (same contents?);
- notes that matching on one case does not show the two always agree;
- gives two separate objects with identical text as a case where they differ.

Reread 2.6.6 *Comparing Objects with equals* and 2.6.4 *Aliases and Reference Comparison* if your reasoning differed.

Question 12

C. The outer `!` covers an `||`, so De Morgan's second law applies: negate each part and change `||` to `&&`. `!(temp < 50)` is `temp >= 50`, and `!(temp > 90)` is `temp <= 90`.

- **A** negates the parts but keeps `||`. Every temperature is at least 50 or at most 90, so A is always `true`; at `temp = 100` the original is `false`.
- **B** uses the wrong opposites. The opposite of `<` is `>=`, so at `temp = 50`, the original is `true` but B is `false`.
- **D** flips the operator but does not negate the parts. No temperature is both below 50 and above 90, so D is always `false`.

Reread 2.6.3 *De Morgan's Laws* and the opposites table in 2.6.1 *Equivalent Boolean Expressions* if you chose differently.

Question 13

a	b	!b	a && !b	!(a && !b)	!a	!a b
true	true	false	false	true	false	true
true	false	true	true	false	false	false
false	true	false	false	true	true	true
false	false	true	false	true	true	true

Equivalent. The two result columns match in all four rows. This is De Morgan's first law with `!b` as the second part: `!(a && !b)` becomes `!a || !!b`, and `!!b` (not-not b) is just `b`.

Reread 2.6.2 *Proving Equivalence with Truth Tables* and 2.6.3 *De Morgan's Laws* if your table differed.

Question 14

```
false
true
true
true
```

After line	a	b	c
1	address A		
2	address A	address A	
3	address A	address A	address B
4	address A	address B	address B

- Line 5: `a` holds A and `b` now holds B, so `a == b` is `false`. Line 4 ended the alias between `a` and `b`.
- Line 6: `b` and `c` both hold B, so `b == c` is `true`.
- Line 7: `a` and `c` are different objects, but both hold `map`, so `equals` is `true`.
- Line 8: different objects, so `a != c` is `true`.

Reread 2.6.4 *Aliases and Reference Comparison* and 2.6.6 *Comparing Objects with equals* if your trace differed.

Question 15

Error: line 3 uses `!=`, which compares references. Each `new` built a separate object, so entered `!=` stored is `true` even though both hold `blue`, and the program prints `Access denied`.

Fix: compare contents with `equals`, and negate the result to keep the meaning "do not match":

```
1 if (!entered.equals(stored))
```

Now `entered.equals(stored)` is `true`, `!true` is `false`, and the `else` prints `Access granted`. Just as `!(a == b)` is `a != b` for numbers, "not equal contents" for strings is `!a.equals(b)`.

Reread 2.6.6 *Comparing Objects with equals* if you did not spot the comparison.

Question 16

D. It calls `equals` on `d`, which holds `null`. There is no object to run the method on, so Java throws a `NullPointerException`.

- **A** is safe: `d == null` is `true`, so `||` short-circuits and `d.equals(a)` is never evaluated. The result is `true`.
- **B** is safe: `d != null` is `false`, so `&&` short-circuits. The result is `false`.
- **C** is safe: `equals` is called on `a`, a real object, and `null` is only the argument. `equals` returns `false` when given `null`.

Reread 2.6.5 *Checking for null* and 2.6.6 *Comparing Objects with equals* if you chose differently.

Question 17

Model answer: Use `x = 1` and `y = 5`. Then `x == 1 && y == 2` is `true && false`, which is `false`, so `!(x == 1 && y == 2)` is `true`. But `x != 1 && y != 2` is `false && true`, which is `false`. The values differ, so the expressions are not equivalent. By De Morgan's first law the correct form flips `&&` to `||`: `x != 1 || y != 2`, which is `false || true`, `true`, matching the original.

A complete answer:

- gives values where exactly one of the two comparisons is true (such as `x = 1, y = 5`, or `x = 3, y = 2`);
- shows the original is `true` and Sam's version is `false` for those values;
- gives `x != 1 || y != 2` as the correct expression.

Reread 2.6.1 *Equivalent Boolean Expressions* and 2.6.3 *De Morgan's Laws* if your reasoning differed.