

2.2 Boolean Expressions

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

By the end you will be able to

- Use `==` and `!=` to ask whether two values are the same, and predict the true or false result.
- Use `<`, `>`, `<=`, and `>=` to compare numbers, including values that sit exactly on the boundary.
- Work out a comparison that contains arithmetic, knowing that the arithmetic happens first.
- Store a true or false result in a boolean variable and print it, with or without a label.
- Explain why `==` on two String objects compares references, and use `equals` and `compareTo` to compare text instead.

What to remember from earlier

- `boolean` is a primitive type with exactly two values, `true` and `false`. ([Topic 1.2: Variables and Data Types](#), [1.2.3 The Three Primitive Types: `int`, `double`, and `boolean`](#))
- Multiplication, division, and remainder happen before addition and subtraction, and parentheses change that order. ([Topic 1.3: Expressions and Output](#), [1.3.8 Operator Precedence](#))
- An assignment works out its right side first and then stores the result in the variable on the left. ([Topic 1.4: Assignment Statements and Input](#), [1.4.2 The Right Side Is Evaluated First](#))
- A `double` can carry a tiny round-off error, so `0.1 + 0.2` is not exactly `0.3`. ([Topic 1.5: Casting and Range of Variables](#), [1.5.7 Round-Off Error with double Values](#))
- A reference variable holds a reference to an object, and each use of `new` creates a brand-new object. ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.1 Constructors and the `new` Keyword](#))
- `a.equals(b)` is true when two strings have exactly the same characters. ([Topic 1.15: String Manipulation](#), [1.15.9 Comparing Strings with `equals` and `compareTo`](#))

2.2.1 Asking Whether Two Values Are Equal

In Topic 2.1 you described algorithms that make decisions: "If the guess matches the secret number, say You win." Before a program can make that choice, it has to answer the question inside it: does the guess match? Every program you have written so far computes numbers and text. Now you need a way to compute an answer to a yes-or-no question.

Java already has a type for yes-or-no answers. Recall from Topic 1.2 that `boolean` is a primitive type with exactly two values, `true` and `false`. So what is needed is a way to write a question whose answer Java works out as one of those two values.

That is what a comparison does. You write two values with a symbol between them, and Java checks how the two values relate. Think of it like asking a referee a question that can only be answered "yes" or "no": Java never answers "maybe" or "7". The answer is always `true` or `false`.

A **relational operator** is a symbol that compares two values and produces a `boolean` result. The first two relational operators ask about sameness:

Operator	Read it as	true when	Example
<code>==</code>	"is equal to"	the two values are the same	<code>a == b</code>
<code>!=</code>	"is not equal to"	the two values are not the same	<code>a != b</code>

Both are two characters typed with no space between them. `==` is two equals signs; `!=` is an exclamation mark followed by an equals sign. Java uses two equals signs because a single `=` already has a job: it is the assignment operator from Topic 1.4, which stores a value. `==` stores nothing; it only asks.

An expression that Java evaluates to `true` or `false` is called a **Boolean expression**.

`guess == secret` is a Boolean expression. So is `guess != secret`. Every expression built with a relational operator is a Boolean expression, because the operator's result is always a `boolean` value. (Boolean is capitalized here because, like the type `boolean`, it is named after the mathematician George Boole.)

KEY IDEA

A relational operator compares two values and produces a `boolean`. `a == b` is `true` when the values are the same; `a != b` is `true` when they are different. For primitive types, Java compares the actual values.

WORKED EXAMPLE Checking a guess

A guessing game compares the player's guess with a secret number. Watch the result of each comparison, and notice that the comparisons themselves never change either variable.

```

1 int guess = 7;
2 int secret = 7;
3 System.out.println(guess == secret);
4 System.out.println(guess != secret);
5 guess = 4;
6 System.out.println(guess == secret);
7 System.out.println(guess != secret);

```

Line	What Java does	guess	secret	Printed
1	Stores 7.	7		
2	Stores 7.	7	7	
3	<code>7 == 7</code> : the values are the same.	7	7	<code>true</code>
4	<code>7 != 7</code> : they are not different.	7	7	<code>false</code>
5	Assignment: replaces 7 with 4.	4	7	
6	<code>4 == 7</code> : the values are not the same.	4	7	<code>false</code>
7	<code>4 != 7</code> : they are different.	4	7	<code>true</code>

Output:

```
true
false
false
true
```

Two things to notice. First, `println` prints a `boolean` result exactly as it prints a `boolean` literal from Topic 1.3: the word `true` or `false`, with no quotation marks. Second, for the same two values, `==` and `!=` always give opposite answers. If one is `true`, the other is `false`.

The values on each side of `==` are primitive values here, so Java compares those values directly: is the `int` in `guess` the same number as the `int` in `secret`? Now vary the types. The next example compares an `int` with a `double`, compares two `boolean` values, and stores a result in a `boolean` variable instead of printing it right away.

WORKED EXAMPLE Mixing types and storing the answer

```
1 int boxes = 3;
2 double weight = 3.0;
3 boolean sameNumber = boxes == weight;
4 System.out.println(sameNumber);
5 boolean doorOpen = true;
6 System.out.println(doorOpen != false);
```

Line 3 has two operators, `=` and `==`. As in every assignment (Topic 1.4), Java finishes the right side before it stores anything:

Step	What Java does	Result
1	Look up <code>boxes</code> (3) and <code>weight</code> (3.0).	
2	One side is a <code>double</code> , so the <code>int</code> 3 is widened to 3.0, as in Topic 1.5.	
3	<code>3.0 == 3.0</code> is the same value.	<code>true</code>
4	Store <code>true</code> in the <code>boolean</code> variable <code>sameNumber</code> .	<code>true</code>

Line 4 prints `true`. Line 6 compares two `boolean` values: `doorOpen` holds `true`, and `true != false` asks whether those differ. They do, so it prints `true` too.

```
true
true
```

The variable on line 3 must be declared `boolean`, because the value being stored is `true` or `false`, not a number.

WATCH OUT

Writing `=` when you mean `==`. The two symbols look alike, and in math class one `=` means "is equal to." In Java, one `=` means "store." A student who wants to check a guess writes:

```
1 int guess = 7;
2 boolean correct = guess = 7;
3 System.out.println(correct);
```

This does not compile:

```
error: incompatible types: int cannot be converted to boolean
```

Java reads line 2 as two assignments: store 7 in `guess`, and then store the result of that assignment (the `int` 7) in `correct`. A `boolean` variable cannot hold the number 7, so Java refuses. The fix is to ask instead of store: `boolean correct = guess == 7;`. (Exact error wording can vary between Java versions.)

PAUSE AND TRY

Question 1 Tracing

What does this code print? Track `lives` as you go.

```
1 int lives = 3;
2 int maxLives = 5;
3 System.out.println(lives == maxLives);
4 System.out.println(lives != maxLives);
5 lives = 5;
6 System.out.println(lives == maxLives);
7 boolean roomForMore = lives != maxLives;
8 System.out.println(roomForMore);
```

Question 2 Debugging

This code should print whether the entered PIN matches the saved PIN. It does not compile. Find the error, explain what Java sees, and correct the line.

```
1 int pin = 1234;
2 int entered = 1243;
3 boolean match = entered = pin;
4 System.out.println(match);
```

2.2.2 Comparing Sizes: Less Than and Greater Than

Many decisions do not ask "the same or not?" They ask "how big?" A test is passed with a score of at least 70. A ride lets you on if you are taller than a line on the sign. A coupon works if the total is under a limit. `==` and `!=` cannot answer these, because a score of 85 is not equal to 70 and yet it passes.

For questions about which number is larger, Java has four more relational operators. You know the ideas from math; only the symbols for "or equal" are typed differently, because a keyboard has no `≤` or `≥` key, so Java spells them with two characters:

Operator	Read it as	true when
<	"is less than"	the left value is smaller
>	"is greater than"	the left value is larger
<=	"is less than or equal to"	the left is smaller or the same
>=	"is greater than or equal to"	the left is larger or the same

For <= and >=, the angle bracket comes first and the equals sign second, written together with no space: say "less than or equal," in that order, and type it in that order. These four operators work on numbers (`int` and `double`), because only numbers have a "larger" and "smaller." Like `==` and `!=`, each one produces a `boolean`.

KEY IDEA

<, >, <=, and >= compare two numbers and produce `true` or `false`. The "or equal" versions are also `true` when the two values are the same.

WORKED EXAMPLE Exactly on the passing line

A class passes students with a score of at least 70. The most important value to test is the boundary itself, a score of exactly 70, because that is where the four operators disagree.

```

1 int score = 70;
2 System.out.println(score > 70);
3 System.out.println(score >= 70);
4 System.out.println(score < 70);
5 System.out.println(score <= 70);

```

Line	Question Java answers	Result
2	Is 70 greater than 70?	false
3	Is 70 greater than or equal to 70?	true
4	Is 70 less than 70?	false
5	Is 70 less than or equal to 70?	true

```

false
true
false
true

```

A number is never greater than or less than itself, so > and < are `false` here. The "or equal" operators let the boundary value count, so >= and <= are `true`. Because "at least 70" includes 70 itself, `score >= 70` is the right test; `score > 70` would fail a student who scored exactly 70.

Everyday words map onto the operators, and the boundary decides which one you need:

Words	Includes the boundary?	Operator
more than, over, above	No	>
at least, no fewer than	Yes	>=

less than, under, below	No	<
at most, no more than	Yes	<=

Now vary the values: a `double` compared with an `int`, and negative numbers.

WORKED EXAMPLE A price and two cold mornings

```

1 double price = 19.99;
2 int budget = 20;
3 System.out.println(price < budget);
4 int lowToday = -5;
5 int lowYesterday = -12;
6 System.out.println(lowToday > lowYesterday);
7 System.out.println(lowToday <= lowYesterday);

```

Line	Comparison	Reasoning	Result
3	19.99 < 20	The <code>int</code> 20 is widened to 20.0; 19.99 is smaller.	true
6	-5 > -12	On a number line, -5 is to the right of -12, so it is greater.	true
7	-5 <= -12	-5 is neither smaller than nor equal to -12.	false

```

true
true
false

```

Line 6 is the one to study. It is tempting to think 12 is "bigger" than 5, so -12 must be bigger than -5. But the operators compare positions on the number line, not the size of the digits. A low of -5 degrees is warmer, and greater, than a low of -12.

WATCH OUT

Typing the symbols in the wrong order. Because people say "equal or greater" as often as "greater or equal," students sometimes type `=>` or `=<`:

```

1 int score = 70;
2 System.out.println(score => 70);

```

```
error: illegal start of expression
```

Java does not have a `=>` comparison operator, so it cannot make sense of the line. The same happens with a space in the middle, as in `> =`. Write the angle bracket first and the equals sign right after it: `score >= 70`. (Exact error wording can vary between Java versions.)

PAUSE AND TRY

Question 3 Multiple choice

A roller coaster lets riders on if they are at least 48 inches tall. The `int` variable `height` holds a rider's height in inches. Which Boolean expression is `true` exactly when the rider may ride?

- A. `height > 48`
- B. `height >= 48`
- C. `height == 48`
- D. `48 >= height`

Question 4 Tracing

What does this code print?

```
1 double speed = 54.5;
2 int limit = 55;
3 System.out.println(speed < limit);
4 System.out.println(speed >= limit);
5 int depth = -30;
6 System.out.println(depth > -20);
7 System.out.println(limit <= 55);
```

2.2.3 Arithmetic Happens Before Comparison

Real questions usually involve a calculation. "Can 14 apples be shared equally among 4 students?" is really "is the remainder of 14 divided by 4 equal to 0?" "Will two shirts at \$20.75 cost more than \$40?" is "is 20.75 times 2 greater than 40?" So a comparison often has arithmetic on one or both sides.

Recall from Topic 1.3 that Java works out an expression in a fixed order: `*`, `/`, and `%` before `+` and `-`. Relational operators come **after** all of the arithmetic operators. In `apples % students == 0`, Java first computes `apples % students`, and only then compares the result with `0`. This order makes sense: a comparison needs two finished numbers, so Java finishes each side first. You can add parentheses for readability, as in `(apples % students) == 0`, but they do not change the answer.

KEY IDEA

In an expression that mixes arithmetic and comparison, Java does all the arithmetic first, then compares the two results. The whole expression still produces a single `boolean`.

WORKED EXAMPLE Sharing apples and buying shirts

```
1 int apples = 14;
2 int students = 4;
3 System.out.println(apples % students == 0);
4 System.out.println(apples / students >= 3);
5 double total = 12.5 + 8.25;
6 System.out.println(total * 2 > 40);
```

Each comparison is worked out in two steps: arithmetic, then the comparison.

Line	Arithmetic first	Then compare	Result
3	14 % 4 is 2	2 == 0	false

4	14 / 4 is 3 (integer division)	3 >= 3	true
5	12.5 + 8.25 is 20.75, stored in total		
6	20.75 * 2 is 41.5	41.5 > 40	true

```
false
true
true
```

Line 3 says the apples cannot be shared equally: 4 students get 3 apples each and 2 are left over. Line 4 uses integer division from Topic 1.3, so `14 / 4` is 3, not 3.5, before the comparison happens.

The remainder operator is especially useful in comparisons. A whole number is **even** exactly when dividing it by 2 leaves no remainder, so `number % 2 == 0` is a Boolean expression that is `true` for even numbers and `false` for odd ones. Here it is with different values, plus a comparison that adds two variables first:

```
1 int tickets = 18;
2 int chairs = 25;
3 System.out.println(tickets % 2 == 0);
4 System.out.println(chairs % 2 == 0);
5 System.out.println(chairs % 2 != 0);
6 System.out.println(tickets + chairs > 40);
```

```
true
false
true
true
```

`18 % 2` is 0, so line 3 is `true`. `25 % 2` is 1, so line 4 (`1 == 0`) is `false` and line 5 (`1 != 0`) is `true`: the same remainder, opposite questions. Line 6 adds first, `18 + 25` is 43, then checks `43 > 40`.

WATCH OUT

Comparing double results with `==`. Topic 1.5 showed that `0.1 + 0.2` is stored as `0.30000000000000004`. That tiny round-off error matters for `==`, which compares the actual stored values:

```
1 double sum = 0.1 + 0.2;
2 System.out.println(sum);
3 System.out.println(sum == 0.3);
```

```
0.30000000000000004
false
```

On paper the sum is exactly 0.3, so students expect `true`. But Java compares the value it actually stored, and that value is not exactly 0.3. Two common fixes: ask whether the two `double` values are very close, using `Math.abs` from Topic 1.11 to measure the distance between

them, or keep amounts that must be exact (such as money in cents) in `int` variables, which have no round-off error.

```
1 double sum = 0.1 + 0.2;
2 System.out.println(Math.abs(sum - 0.3) < 0.000001);
3 int cents = 10 + 20;
4 System.out.println(cents == 30);
```

```
true
true
```

Line 2 asks "is the gap between `sum` and 0.3 smaller than one millionth?" It is, so the answer is `true`. Line 4 compares two `int` values, which are always exact.

WATCH OUT

Writing a range the way math does. In math, $60 < t < 80$ means "t is between 60 and 80." Java does not read it that way:

```
1 int temperature = 72;
2 System.out.println(60 < temperature < 80);
```

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

Java works left to right. `60 < temperature` becomes `true`, and then Java is asked to work out `true < 80`. A `boolean` is not a number, so it cannot be less than 80, and the message says exactly that: the first value is a `boolean`, the second an `int`. A comparison produces a `boolean`, and a `boolean` cannot be compared by size. Checking two conditions at once needs a new kind of operator that joins Boolean expressions; that is Topic 2.5. For now, write each comparison on its own: `60 < temperature` and `temperature < 80`. (Exact wording can vary between Java versions.)

PAUSE AND TRY

Question 5 Tracing

A movie runs for 135 minutes. What does this code print? Show the arithmetic step for each line.

```
1 int minutes = 135;
2 System.out.println(minutes / 60 == 2);
3 System.out.println(minutes % 60 > 15);
4 System.out.println(minutes % 2 != 0);
5 System.out.println(minutes - 35 <= 100);
```

Question 6 Multiple choice

What does `System.out.println(0.1 * 3 == 0.3);` print?

- A. `true`
- B. `false`
- C. `0.3`
- D. Nothing; you cannot use `==` with `double` values.

2.2.4 Storing and Printing Boolean Results

A program often needs the answer to a question more than once, or wants to give it a meaningful name. "Is this person a teenager?" might be needed in several places, and `isTeen` says more than `age >= 13` does each time it appears. Topic 1.4 showed that a variable can hold the result of any expression of its type. A Boolean expression has type `boolean`, so its result goes in a `boolean` variable, and a good name for that variable reads like a yes-or-no question or statement: `isTeen`, `passed`, `canAfford`.

There is one thing to understand clearly. When Java runs `boolean isTeen = age >= 13;`, it evaluates the comparison **once**, at that moment, and stores the answer, `true` or `false`. The variable holds the answer, not the question. If `age` changes later, `isTeen` does not change by itself. It changes only if you assign to it again.

KEY IDEA

A Boolean expression has type `boolean`, so its result can be stored in a `boolean` variable. The variable stores the answer at the moment of assignment; it does not update when the values that produced it change.

WORKED EXAMPLE The answer, not the question

```

1 int age = 12;
2 boolean isTeen = age >= 13;
3 System.out.println(isTeen);
4 age = 14;
5 System.out.println(isTeen);
6 isTeen = age >= 13;
7 System.out.println(isTeen);

```

Line	What Java does	age	isTeen	Printed
1	Stores 12.	12		
2	<code>12 >= 13</code> is false; stores false.	12	false	
3	Prints <code>isTeen</code> .	12	false	false
4	Replaces 12 with 14. <code>isTeen</code> is untouched.	14	false	
5	Prints <code>isTeen</code> .	14	false	false
6	<code>14 >= 13</code> is true; stores true.	14	true	

7	Prints isTeen.	14	true	true
---	----------------	----	------	------

Line 5 surprises many students: the person is now 14, yet `isTeen` still prints `false`. Nothing re-asked the question until line 6.

Because a Boolean expression has type `boolean`, it cannot be stored in a variable of another type. Java checks this before the program runs:

```
1 int answer = 5 > 3;
2 System.out.println(answer);
```

```
error: incompatible types: boolean cannot be converted to int
```

`5 > 3` is `true`, a `boolean`, and an `int` variable holds only whole numbers. There is no automatic conversion between `boolean` and the number types in Java. (Exact wording can vary between Java versions.)

Printing a result with a label uses string concatenation from Topic 1.15: `+` with a `String` on one side turns a `boolean` into the text `true` or `false`. But concatenation brings an order-of-operations trap.

WATCH OUT

Forgetting parentheses around a comparison in a label. A student wants to print Even: `true`:

```
1 int count = 6;
2 System.out.println("Even: " + count % 2 == 0);
```

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

Here is the order Java follows. `%` comes first: `count % 2` is `0`. Then `+`, which comes before `==`: `"Even: " + 0` is the `String` `"Even: 0"`. Last, `==` is asked to compare `"Even: 0"` with the `int` `0`. A `String` and an `int` cannot be compared, so the code does not compile; the error message names exactly those two types. Students expect the comparison to "go with" the label because that is how they read it, but Java follows operator order, not intent.

The fix is parentheses, which make Java finish the comparison before joining it to the label:

```
1 int count = 6;
2 System.out.println("Even: " + (count % 2 == 0));
3 System.out.println("Remainder: " + count % 2);
```

```
Even: true
Remainder: 0
```

Line 3 needs no extra parentheses, because `%` already happens before `+`. Only a relational operator needs them, since relational operators come after `+`.

PAUSE AND TRY**Question 7** Tracing

A club needs more than half of its 50 members to vote yes. What does this code print?

```
1 int votes = 26;
2 int members = 50;
3 boolean majority = votes * 2 > members;
4 System.out.println("Majority: " + majority);
5 System.out.println("Tie: " + (votes * 2 == members));
6 votes = 25;
7 System.out.println("Majority: " + majority);
8 System.out.println("Tie: " + (votes * 2 == members));
```

Question 8 Debugging

This code should print whether a score of 82 passes. It does not compile. Find the error, explain it, and correct the line.

```
1 int score = 82;
2 String passed = score >= 70;
3 System.out.println(passed);
```

2.2.5 Comparing Strings and Other Objects

Every comparison so far used primitive values. Programs also need to compare text: does the typed password match the saved one? Does this name come before that one alphabetically? It is natural to reach for `==` and `<`. Neither does what you want with a `String`.

Recall from Topics 1.12 and 1.13 that a variable of a reference type, such as `String`, does not hold the object itself. It holds a reference to the object, which you can think of as the object's address in memory. And each use of `new` creates a brand-new object at a new address.

`==` always compares what is stored in the two variables. For an `int`, that is the number. For a `String` variable, that is the reference. So for reference types, `first == second` asks "do these two variables refer to the very same object?" It does not look inside the objects at all. Two different objects can hold exactly the same characters, the way two copies of the same book are still two books.

To ask whether two strings hold the same text, use `equals` from Topic 1.15.

KEY IDEA

With reference types, `==` and `!=` compare references: whether two variables refer to the same object. To compare the text inside two strings, use `equals`.

WORKED EXAMPLE Two objects, same text

```

1 String first = new String("Tiger42");
2 String second = new String("Tiger42");
3 System.out.println(first.equals(second));
4 System.out.println(first == second);
5 System.out.println(first != second);

```

Lines 1 and 2 each use `new`, so there are two separate `String` objects. Memory looks like this (the arrows stand for references):

```

first  → [ "Tiger42" ]  object 1
second → [ "Tiger42" ]  object 2

```

Line	Question	Result
3	Same characters? (<code>equals</code> looks inside)	true
4	Same object? (<code>==</code> compares references)	false
5	Different objects?	true

```

true
false
true

```

Line 4 is the surprise. The text matches, but the arrows point to two different objects, so `==` says `false`. If a login program used `==` to check a password, a correct password could be rejected. Never rely on `==` to compare the text of two strings; its answer depends on how the strings were created, not on what they say. Topic 2.6 explores comparing references in more detail.

The size operators `<`, `>`, `<=`, and `>=` do not work on strings at all. To ask which string comes first alphabetically, use `compareTo` from Topic 1.15, which returns an `int`, and then compare that `int` with `0` using a relational operator. That turns `compareTo` into a yes-or-no question.

WORKED EXAMPLE Which word comes first?

```

1 String a = "apple";
2 String b = "banana";
3 System.out.println(a.compareTo(b) < 0);
4 System.out.println(a.compareTo(b) == 0);
5 System.out.println(b.compareTo(a) > 0);
6 System.out.println(a.length() < b.length());

```

Line	int from the method	Then compare	Result
3	"apple".compareTo("banana"): negative	<code>< 0</code>	true
4	the same negative number	<code>== 0</code>	false
5	"banana".compareTo("apple"): positive	<code>> 0</code>	true
6	lengths 5 and 6	<code>5 < 6</code>	true

```
true  
false  
true  
true
```

Read line 3 as "apple compared to banana is less than zero," which means "apple comes before banana." You never need the exact number `compareTo` returns, only whether it is negative, zero, or positive. Line 6 is different: `length()` returns an `int`, so `<` compares two ordinary numbers.

WATCH OUT

Using `<` directly on strings. A student who wants to know whether "apple" comes before "banana" writes:

```
1 String a = "apple";  
2 String b = "banana";  
3 System.out.println(a < b);
```

```
error: bad operand types for binary operator '<'  
  first type:  String  
  second type: String
```

`<` works only on numbers, and a `String` is not a number, so the code does not compile. The fix is `a.compareTo(b) < 0`. (Exact wording can vary between Java versions.)

PAUSE AND TRY

Question 9 Multiple choice

What does this code print?

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

- A. true then true
- B. false then true
- C. true then false
- D. false then false

Question 10 Tracing

What does this code print?

```

1 String first = "Lopez";
2 String second = "Lee";
3 System.out.println(first.compareTo(second) < 0);
4 System.out.println(second.compareTo(first) < 0);
5 System.out.println(first.equals("lopez"));
6 System.out.println(first.length() > second.length());

```

Putting it together

A theater's ticket kiosk answers several yes-or-no questions about a customer before selling a ticket. This program uses every idea in the chapter. Try predicting all six lines before reading the trace.

WORKED EXAMPLE A ticket kiosk

TicketCheck.java

```

1 public class TicketCheck
2 {
3     public static void main(String[] args)
4     {
5         int age = 15;
6         double cash = 12.0;
7         double ticketPrice = 11.5;
8         int seatsLeft = 40;
9         int groupSize = 6;
10        String rating = new String("PG-13");
11        boolean oldEnough = age >= 13;
12        boolean canAfford = cash >= ticketPrice;
13        System.out.println("Old enough: " + oldEnough);
14        System.out.println("Can afford: " + canAfford);
15        System.out.println("Exact cash: " + (cash == ticketPrice));
16        System.out.println("Room for group: " + (seatsLeft - groupSize >= 0));
17        System.out.println("Even group: " + (groupSize % 2 == 0));
18        System.out.println("Rated PG-13: " + rating.equals("PG-13"));
19    }
20 }

```

Lines 5–10 set up the values: a 15-year-old with \$12.00, a ticket price of \$11.50, 40 seats left, a group of 6, and a rating string made with `new`.

Line	Expression	Work	Result
11	<code>age >= 13</code>	<code>15 >= 13</code> ; stored in <code>oldEnough</code>	true
12	<code>cash >= ticketPrice</code>	<code>12.0 >= 11.5</code> ; stored in <code>canAfford</code>	true
15	<code>cash == ticketPrice</code>	<code>12.0 == 11.5</code>	false
16	<code>seatsLeft - groupSize >= 0</code>	arithmetic first: <code>40 - 6</code> is 34; <code>34 >= 0</code>	true
17	<code>groupSize % 2 == 0</code>	<code>6 % 2</code> is 0; <code>0 == 0</code>	true
18	<code>rating.equals("PG-13")</code>	same characters	true

```
Old enough: true
Can afford: true
Exact cash: false
Room for group: true
Even group: true
Rated PG-13: true
```

Look at the details. Lines 13 and 14 print stored `boolean` variables, so no parentheses are needed. Lines 15–17 put a comparison inside a label, so each comparison is wrapped in parentheses; without them, `+` would join the label first and the code would not compile. Line 18 needs no extra parentheses because `equals` is a method call, which Java finishes before it does any `+`. And line 18 uses `equals`, not `==`: `rating` was made with `new`, so `rating == "PG-13"` would compare two different objects.

Vocabulary

Term	Definition	Example
relational operator	A symbol that compares two values and produces a <code>boolean</code> : <code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code>	<code>score >= 70</code>
Boolean expression	An expression that Java evaluates to <code>true</code> or <code>false</code>	<code>count % 2 == 0</code>
even	A whole number that leaves no remainder when divided by 2	<code>18 % 2 == 0</code>

Chapter summary

- A relational operator compares two values and produces a `boolean`, so any expression built with one is a Boolean expression whose value is `true` or `false`.
- `==` asks whether two values are the same and `!=` whether they differ. For primitive types these compare the actual values, which is why `double` round-off error can make `==` give `false`.
- `<`, `>`, `<=`, and `>=` compare numbers. Only the "or equal" versions are `true` when the two values are the same, so the boundary decides which operator a question needs.
- Arithmetic happens before comparison, and `+` happens before comparison too, so a comparison inside a printed label needs parentheses.
- A Boolean result can be stored in a `boolean` variable; it holds the answer at that moment and does not update on its own.
- For reference types, `==` and `!=` compare references, not contents. Compare string text with `equals`, and alphabetical order with `compareTo` and `0`.

Review questions

Question 11 Multiple choice

The variable `points` holds 50. Which expression evaluates to `false`?

- A. `points >= 50`

- B. `points % 10 == 0`
- C. `points / 3 > 16`
- D. `points != 49`

Question 12 Tracing

A gym reads a member's age, account balance, and membership level, then checks a \$10.00 entry fee. Write the exact output.

GymEntry.java

```
1 import java.util.Scanner;
2
3 public class GymEntry
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("17 9.5 Gold");
8         int age = input.nextInt();
9         double balance = input.nextDouble();
10        String level = input.next();
11        double fee = 10.0;
12        boolean adult = age >= 18;
13        System.out.println("Adult: " + adult);
14        System.out.println("Short: " + (fee - balance));
15        System.out.println("Enough money: " + (balance >= fee));
16        System.out.println("Gold member: " + level.equals("Gold"));
17        System.out.println("Odd age: " + (age % 2 == 1));
18    }
19 }
```

Question 13 Debugging

Two players type their team colors. This program should print `Same team: true` for the input `blue blue`, but it prints `Same team: false`. Find the error, explain why it prints `false`, and correct the line.

TeamWrong.java

```
1 import java.util.Scanner;
2
3 public class TeamWrong
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("blue blue");
8         String myTeam = input.next();
9         String yourTeam = input.next();
10        System.out.println("Same team: " + (myTeam == yourTeam));
11    }
12 }
```

Question 14 Short answer

Two students want to test whether the `int` variable `total` is a multiple of 5. One writes `total % 5 == 0`; the other writes `total / 5 == 0`. Give the value of each expression when `total` is 15 and when `total` is 3, and explain which test is correct.

Question 15 Multiple choice

What does this code print?

```
1 String x = "Zoo";
2 String y = "apple";
3 System.out.println(x.compareTo(y) < 0);
4 System.out.println(x.equals("zoo"));
5 System.out.println(x.length() == y.length());
6 System.out.println(y.compareTo(x) < 0);
```

- A. false, false, false, true
- B. true, false, false, false
- C. false, true, false, true
- D. true, true, true, false

Reflection

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

- ☐ I can predict the result of == and != and explain why = cannot be used to compare. Reread: 2.2.1 Asking Whether Two Values Are Equal
- ☐ I can choose between <, >, <=, and >= for a question, paying attention to the boundary value. Reread: 2.2.2 Comparing Sizes: Less Than and Greater Than
- ☐ I can work out a comparison that contains arithmetic, and explain why == on double results is risky. Reread: 2.2.3 Arithmetic Happens Before Comparison
- ☐ I can store a Boolean result in a boolean variable and print it with a label, using parentheses where needed. Reread: 2.2.4 Storing and Printing Boolean Results
- ☐ I can explain what == does with two String objects and use equals and compareTo instead. Reread: 2.2.5 Comparing Strings and Other Objects

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
false
true
true
false
```

Line	Comparison or action	Result
3	<code>3 == 5</code>	false
4	<code>3 != 5</code>	true
5	<code>lives becomes 5</code>	
6	<code>5 == 5</code>	true
7	<code>5 != 5</code> , stored in <code>roomForMore</code>	false
8	prints <code>roomForMore</code>	false

Line 7 does not print anything; it stores `false` in `roomForMore`, and line 8 prints what was stored. If you got `true` on the last line, you may have used the old value `3`; line 5 replaced it before line 7 ran.

Reread 2.2.1 *Asking Whether Two Values Are Equal* if your output differed.

Question 2

The error is on line 3. `entered = pin` uses the assignment operator, so Java reads it as "store `pin` in `entered`." The result of that assignment is the `int` `1234`, and a `boolean` variable cannot hold an `int`, so Java reports incompatible types: `int` cannot be converted to `boolean`. The programmer wanted to ask a question, not store a value.

Corrected code:

```
1 int pin = 1234;
2 int entered = 1243;
3 boolean match = entered == pin;
4 System.out.println(match);
```

It prints `false`, because `1243 == 1234` compares two different numbers. (Notice that even if the original had compiled, it would have overwritten the `entered` PIN, which is never what a PIN check should do.)

Reread the Watch Out in 2.2.1 *Asking Whether Two Values Are Equal* if you missed it.

Question 3

B. "At least 48" includes 48 itself, so the test must be `true` for 48 and for every taller height. `height >= 48` does exactly that.

- **A** is wrong: `>` leaves out the boundary. A rider who is exactly 48 inches tall gets `false` and is turned away.
- **C** is wrong: `=>` is not a Java operator, so the expression does not compile.
- **D** is wrong: it reverses the question. `48 >= height` is `true` when the rider is 48 inches or shorter.

Reread 2.2.2 *Comparing Sizes: Less Than and Greater Than* if you chose differently.

Question 4

```
true
false
false
true
```

- Line 3: `54.5 < 55` (the `int` `55` is widened to `55.0`). `54.5` is smaller, so `true`.
- Line 4: `54.5 >= 55` is `false`; it is the opposite question from line 3.
- Line 6: `-30 > -20` is `false`. On the number line, `-30` is to the left of `-20`, so it is smaller, even though `30` is more than `20`.
- Line 7: `55 <= 55` is `true`, because the "or equal" part lets the boundary count.

Reread 2.2.2 *Comparing Sizes: Less Than and Greater Than* if your output differed.

Question 5

```
true
false
true
true
```

Line	Arithmetic first	Then compare	Result
2	<code>135 / 60</code> is 2	<code>2 == 2</code>	<code>true</code>
3	<code>135 % 60</code> is 15	<code>15 > 15</code>	<code>false</code>
4	<code>135 % 2</code> is 1	<code>1 != 0</code>	<code>true</code>
5	<code>135 - 35</code> is 100	<code>100 <= 100</code>	<code>true</code>

Line 2 uses integer division (two whole hours fit in 135 minutes). Line 3 is the trap: the remainder is exactly 15, and 15 is not greater than itself. Line 4 says 135 is odd.

Reread 2.2.3 *Arithmetic Happens Before Comparison* if your output differed.

Question 6

B. The multiplication happens first. `0.1` cannot be stored exactly in a `double`, so `0.1 * 3` comes out very slightly different from `0.3` (it prints as `0.30000000000000004`). `==` compares the stored values, which differ, so the result is `false`.

- **A** is wrong: it is the answer on paper, but Java compares the values it actually stored, and round-off error makes them differ.
- **C** is wrong: a Boolean expression produces `true` or `false`, never a number. The multiplication is only a step along the way.
- **D** is wrong: `==` works on `double` values and compiles fine. It is just risky because of round-off error.

Reread the round-off Watch Out in 2.2.3 *Arithmetic Happens Before Comparison* if you chose differently.

Question 7

```
Majority: true
Tie: false
Majority: true
Tie: true
```

Line	What happens	Printed
3	26 * 2 is 52; 52 > 50 is true; stored in majority.	
4	Prints the stored value.	Majority: true
5	52 == 50 is false (worked out now, inside parentheses).	Tie: false
6	votes becomes 25. majority is not re-evaluated.	
7	Prints the stored value, still true.	Majority: true
8	25 * 2 is 50; 50 == 50 is true, worked out now.	Tie: true

Line 7 still says `true` because `majority` holds the answer from line 3. Line 8 differs from line 5 because its comparison is written right there in the `println`, so it is evaluated again with the new value of `votes`.

Reread 2.2.4 *Storing and Printing Boolean Results* if your output differed.

Question 8

The error is on line 2. `score >= 70` is a Boolean expression, so its value is the boolean `true`, not text. A String variable cannot hold a boolean, and Java reports incompatible types: boolean cannot be converted to String. The quotation-free word `true` is a boolean value, not the String `"true"`.

Corrected code:

```
1 int score = 82;
2 boolean passed = score >= 70;
3 System.out.println(passed);
```

It prints `true`, because `82 >= 70`.

Reread 2.2.4 *Storing and Printing Boolean Results* if you missed it.

Question 9

B. Each `new` creates its own String object, so `saved` and `typed` refer to two different objects. `==` compares the references, which differ, so line 3 prints `false`. `equals` compares the characters, which are the same, so line 4 prints `true`.

- **A** is wrong: it treats `==` as if it compared text. For reference types it compares references.
- **C** is wrong: it swaps the two methods' jobs. `equals` is the one that looks at the characters.
- **D** is wrong: the two strings do hold exactly the same characters, so `equals` is `true`.

Reread 2.2.5 *Comparing Strings and Other Objects* if you chose differently.

Question 10

```
false
true
false
true
```

- Line 3: "Lopez" and "Lee" match at L, then differ at o versus e. o comes after e, so "Lopez".compareTo("Lee") is positive, and positive < 0 is false.
- Line 4: the same pair in the other order gives a negative number, so < 0 is true: Lee comes before Lopez.
- Line 5: equals is case-sensitive (Topic 1.15). L and l differ, so false.
- Line 6: 5 > 3 compares two int lengths, so true.

If you said line 3 is true because "Lopez" is longer, remember that compareTo uses dictionary order, not length.

Reread 2.2.5 *Comparing Strings and Other Objects* if your output differed.

Question 11

C. Arithmetic first: 50 / 3 is integer division, so it is 16, not 16.67. Then 16 > 16 is false, because a number is not greater than itself.

- **A** is wrong: 50 >= 50 is true; the "or equal" part includes the boundary.
- **B** is wrong: 50 % 10 is 0, and 0 == 0 is true.
- **D** is wrong: 50 is different from 49, so != gives true.

Reread 2.2.2 *Comparing Sizes: Less Than and Greater Than* and 2.2.3 *Arithmetic Happens Before Comparison* if you chose differently.

Question 12

```
Adult: false
Short: 0.5
Enough money: false
Gold member: true
Odd age: true
```

The Scanner reads 17 into age, 9.5 into balance, and "Gold" into level (Topic 1.4).

Line	Expression	Work	Printed
12	age >= 18	17 >= 18; stored	
13	adult	stored value	Adult: false
14	fee - balance	10.0 - 9.5 (arithmetic only)	Short: 0.5
15	balance >= fee	9.5 >= 10.0	Enough money: false
16	level.equals("Gold")	same characters	Gold member: true
17	age % 2 == 1	17 % 2 is 1; 1 == 1	Odd age: true

Line 14 is not a Boolean expression at all; it prints a double. It shows the parentheses doing the same job as in the comparison lines: finishing a calculation before + joins it to the label.

Reread 2.2.4 *Storing and Printing Boolean Results* and 2.2.5 *Comparing Strings and Other Objects* if your output differed.

Question 13

The error is on line 10. `myTeam == yourTeam` compares references, not text. Each call to `next()` creates a new `String` object for the word it reads, so `myTeam` and `yourTeam` refer to two different objects that happen to hold the same characters. `==` asks "same object?", and the answer is no. The code compiles, so only the wrong output reveals the bug.

Corrected code:

TeamFixed.java

```
1 import java.util.Scanner;
2
3 public class TeamFixed
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("blue blue");
8         String myTeam = input.next();
9         String yourTeam = input.next();
10        System.out.println("Same team: " + myTeam.equals(yourTeam));
11    }
12 }
```

`equals` compares the characters, so it prints `Same team: true`. No extra parentheses are needed around the method call.

Reread 2.2.5 *Comparing Strings and Other Objects* if you missed it.

Question 14

Model answer: When `total` is 15: `15 % 5` is 0, so `total % 5 == 0` is true; `15 / 5` is 3, so `total / 5 == 0` is false. When `total` is 3: `3 % 5` is 3, so the first test is false; `3 / 5` is 0 with integer division, so the second test is true. The `%` test is correct: a number is a multiple of 5 exactly when dividing by 5 leaves no remainder. The `/` test asks a different question, whether the whole-number quotient is zero, so it gives the wrong answer for both values.

A complete answer must:

- give all four results: true and false for 15, false and true for 3,
- say that arithmetic is done before the comparison, including integer division for `3 / 5`, and
- explain that "multiple of 5" means "remainder 0," which is what `%` measures.

Reread 2.2.3 *Arithmetic Happens Before Comparison* if your answer differed.

Question 15

B. Line 3: Topic 1.15 showed that `compareTo` puts every uppercase letter before every lowercase letter. `Z` is uppercase and `a` is lowercase, so `"Zoo"` comes first, `"Zoo".compareTo("apple")` is negative, and `< 0` is true. Line 4: `equals` is case-sensitive, so `"Zoo"` and `"zoo"` differ: false. Line 5: `lengths 3 == 5` is false. Line 6: the reversed call is positive, so `< 0` is false.

- **A** is wrong: it uses everyday alphabetical order, where `z` comes last. `compareTo` ranks uppercase before lowercase.
- **C** is wrong: it also ignores case, both in `compareTo` and in `equals`.

- **D** is wrong: it treats `equals` as ignoring case and compares the lengths incorrectly; 3 is not equal to 5.

Reread *2.2.5 Comparing Strings and Other Objects* if you chose differently.