

Day 4 Making decisions

Self-Study Guide · About 75 minutes · With answer key

By the end you will be able to

- Write conditions with comparison operators, `===`, `&&`, `||`, and `!`, and predict whether each is true or false.
- Translate a set of rules written in English into an `if`, `else if`, and `else` chain.
- Predict which values are truthy or falsy, and pick between an `if` chain, the conditional operator, and `switch`.
- Read the checker's TS2367 message and explain why the comparison it points at can never be true.

What to remember from earlier

- A boolean is true or false, written in small letters without quotes. ([Day 2: Values, variables and types, 2.4 Numbers, Strings, and Booleans](#))
- Hovering over a `const` in VS Code shows its value as its type, such as `const city: "Lima"`, because a constant can never hold anything else. ([Day 2: Values, variables and types, 2.5 Type Annotations and Inference](#))
- `null` means deliberately empty, and `typeof null` is `"object"`, a quirk that can never be fixed. ([Day 2: Values, variables and types, 2.7 Null and Undefined](#))
- The remainder operator `%` gives what is left over, so an even number has remainder 0 when divided by 2. ([Day 2: Values, variables and types, 2.8 Remainders, Powers, and Rounding](#))
- `Number()` turns text into a number, and text that is not a number becomes `NaN`, which nobody reports. ([Day 2: Values, variables and types, 2.10 Converting with Number and String](#))
- `trim()` removes spaces from the ends of typed text, and `toLowerCase()` turns it into small letters. ([Day 3: Strings, 3.6 Changing Case and Trimming](#))
- `indexOf` gives `-1` when the text is not found, and `===` checks whether two strings are exactly the same. ([Day 3: Strings, 3.8 Searching a String](#))

4.1 Comparison Operators

Before you start, set up a new project folder named `day04` exactly as you set up `day01` in 1.2: open the folder in VS Code, run `npm init -y` and `npm install --save-dev typescript` in its terminal, and type the same `tsconfig.json`. Every program today goes in that folder.

Every program so far has done the same thing every time it ran. On Day 1 you learned that statements run one at a time, from top to bottom, and none was ever skipped. But real programs make decisions: show "Sold out" only when there are no tickets left, charge less for children, warn when a password is too short. Today you learn how a program asks a yes-or-no question and then chooses what to do with the answer.

The answer to a yes-or-no question is a **boolean**: Day 2's type with only two values, `true` and `false`. The first way to make one is to compare two values. Is the score above the pass mark? Is

the price at most the budget? The symbols that ask these questions are **comparison operators**. Each one sits between two values, compares them, and gives back `true` or `false`.

Operator	Means	7 ? 5 gives	5 ? 5 gives
>	is greater than	true	false
<	is less than	false	false
>=	is greater than or equal to	true	true
<=	is less than or equal to	false	true
===	is equal to	false	true
!==	is not equal to	true	false

The two right-hand columns put each operator in place of the `?`. Notice the second column: 5 is not greater than 5, but it **is** greater than or equal to 5. The `=` in `>=` and `<=` is what lets the two sides be equal. Both are written as two characters with no space, and the `=` always comes second: `>=`, never `=>`. Equality is three characters, `===`, which 4.2 explains; a single `=` still means "store a value," as it has since Day 2.

WORKED EXAMPLE Did the student pass?

Read each line as a question, and answer it before you look at the output.

comparisons.ts

```

1  const score = 72;
2  const passMark = 60;
3  console.log(score > passMark);
4  console.log(score < passMark);
5  console.log(score >= 72);
6  console.log(score <= 71);
7  console.log(score === 72);
8  console.log(score !== 72);
9  const isPassing = score >= passMark;
10 console.log(isPassing, typeof isPassing);

```

```

true
false
true
false
true
false
true boolean

```

Line	Comparison	Question it asks	Result
3	<code>score > passMark</code>	Is 72 more than 60?	true
4	<code>score < passMark</code>	Is 72 less than 60?	false
5	<code>score >= 72</code>	Is 72 at least 72?	true
6	<code>score <= 71</code>	Is 72 at most 71?	false
7	<code>score === 72</code>	Is 72 equal to 72?	true
8	<code>score !== 72</code>	Is 72 different to 72?	false

A comparison is an expression, so its result is a value you can store, just like the result of `4 + 5`. Line 9 stores the answer to "is the score at least the pass mark?" in `isPassing`, and line 10 shows that it is `true`, of type `boolean`. The checker knows this too: hover over `isPassing` and VS Code shows `const isPassing: boolean`.

Now vary what is compared. A comparison can have a calculation on either side, and it can compare strings:

`compare-more.ts`

```
1 const price = 4.5;
2 const quantity = 3;
3 const budget = 15;
4 console.log(price * quantity <= budget);
5 console.log(price * quantity + 2 <= budget);
6 console.log("apple" < "banana");
7 console.log("Zebra" < "apple");
8 console.log("cat" < "catalog");
```

```
true
false
true
true
true
```

Line 4 asks whether 4.5×3 fits the budget. The comparison operators come **after** arithmetic in the precedence table from Day 2, so `price * quantity` is worked out to 13.5 first, and then `13.5 <= 15` is `true`. On line 5, `price * quantity + 2` is 15.5, which is more than 15. You never need parentheses around the arithmetic in a comparison, though you may add them to make it easier to read.

Lines 6 to 8 compare strings. Two strings are compared one character at a time from the start, and the first character that differs decides; this is close to dictionary order, which is why `"apple" < "banana"` is `true`. But it is not exactly dictionary order: each character has a code number inside the computer, and every capital letter's number is smaller than every small letter's. So line 7, `"Zebra" < "apple"`, is `true`: Z comes before a. On line 8, `"cat"` runs out of characters while it still matches `"catalog"`, and a string that is the start of a longer one comes first.

KEY IDEA

The comparison operators `>`, `<`, `>=`, `<=`, `===`, and `!==` compare two values and give a boolean. Arithmetic is worked out before the comparison. Strings are compared character by character, and capital letters come before small ones.

WATCH OUT

Comparing numbers that are stored as text. Character-by-character comparison gives a surprise when the strings hold digits:

text-numbers.ts

```
1 const firstScore = "10";
2 const secondScore = "9";
3 console.log(firstScore < secondScore);
4 console.log(Number(firstScore) < Number(secondScore));
```

true
false

As numbers, 10 is more than 9. As strings, line 3 compares the first characters, "1" and "9", and "1" comes first, so "10" < "9" is `true`. Nobody reports anything: comparing two strings is allowed. Line 4 converts both with Day 2's `Number()` first, and gets the right answer. When the values are numbers, make sure they are stored as numbers.

The checker does stop you from comparing a number with a string, because the answer would make no sense:

age-vs-text.ts

```
1 const age = 16;
2 console.log(age < "18");
```

age-vs-text.ts:2:13 - error TS2365: Operator '<' cannot be applied to types 'number' and 'string'.

The message points at the start of the comparison (column 13) and names the two types it was given. Fix it with a number on both sides, `age < 18`, or delete the file, before your next check.

PAUSE AND TRY

Question 1 Tracing

Predict exactly what this program shows.

predict-compare.ts

```
1 const height = 135;
2 const minimum = 120;
3 console.log(height >= minimum);
4 console.log(height - 20 > minimum);
5 console.log(height === minimum + 15);
6 console.log("Ana" < "Ben");
7 console.log("ana" < "Ben");
```

Question 2 Multiple choice

What does `console.log("100" < "25");` show?

- A. `true`
- B. `false`
- C. Nothing: `npx tsc` reports TS2365

D. NaN

4.2 Strict Equality

The most common question a program asks is "are these two the same?" Is the typed password the stored one? Is the command "help"? Day 3 previewed the operator for this, `===`; now you can learn exactly what it does, and why TypeScript programmers always use it instead of a shorter operator that looks similar.

`a === b` is `true` when `a` and `b` have the **same type and the same value**, and `false` otherwise. Its opposite, `a !== b`, is `true` when they differ in type or in value. This kind of equality, which never converts either side to another type, is called **strict equality**. Two numbers are equal when they are the same number. Two strings are equal when they have exactly the same characters in the same order, so case counts. Two booleans are equal when both are `true` or both are `false`.

WORKED EXAMPLE Asking whether two values are the same

equality.ts

```
1 const answer = 6 * 7;
2 console.log(answer === 42);
3 console.log(answer !== 42);
4 const city = "lima";
5 console.log(city === "lima");
6 console.log(city.toUpperCase() === "LIMA");
7 const total = 0.1 + 0.2;
8 console.log(total === 0.3);
9 console.log(total.toFixed(2) === "0.30");
```

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

Line	Comparison	Worked out	Result
2	<code>answer === 42</code>	<code>6 * 7</code> is 42, the same number	<code>true</code>
3	<code>answer !== 42</code>	They are the same, so "not equal" is false	<code>false</code>
5	<code>city === "lima"</code>	Same characters, same order	<code>true</code>
6	<code>city.toUpperCase() === "LIMA"</code>	The new string "LIMA" matches	<code>true</code>
8	<code>total === 0.3</code>	<code>total</code> is 0.30000000000000004, not 0.3	<code>false</code>
9	<code>total.toFixed(2) === "0.30"</code>	Both are the string "0.30"	<code>true</code>

Line 8 is the surprise. As Day 2 showed in 2.11, `0.1 + 0.2` is stored as a binary float a tiny bit above 0.3, and `===` compares the stored values exactly. Never compare the result of decimal

arithmetic with `===`. Line 9 shows one safe way: round both to the number of places you care about with `toFixed`, and compare the strings.

JavaScript has a second, older equality operator, `==`, with its opposite `!=`. It is called **loose equality**: when the two sides have different types, `==` first **converts** one or both of them, and only then compares. The conversion rules are long and full of surprises. Look what `npx tsc` says about two loose comparisons:

loose-equals.ts

```
1 console.log("5" == 5);
2 console.log(0 == "");
```

```
loose-equals.ts:1:13 - error TS2367: This comparison appears to be unintentional because the types 'string' and 'number' have no overlap.
loose-equals.ts:2:13 - error TS2367: This comparison appears to be unintentional because the types 'number' and 'string' have no overlap.
```

The checker rejects both lines, because comparing a string with a number is almost certainly a mistake; 4.3 explains this message in full. But remember from Day 1 that `node` runs a file without checking its types. Run `node loose-equals.ts` anyway, and it shows:

```
true
true
```

On line 1, `==` converted `"5"` to the number 5, so the string and the number count as equal. On line 2, it converted the empty string to the number 0, so "no text at all" counts as equal to zero. Neither is what a reader expects. With `===`, both lines would be `false`, because a string is never strictly equal to a number. Fix or delete `loose-equals.ts` now, before your next check.

So why does `==` exist, and why avoid it? It is from the first version of JavaScript, and, like `typeof null`, it can never be removed without breaking old websites. When both sides already have the same type, `==` gives the same answer as `===`; it only behaves differently when the types differ, and that is exactly where its answers surprise people. This course, like most TypeScript code, uses only `===` and `!==`.

Operator	Name	What it does	Example
<code>=</code>	Assignment	Stores a value in a variable	<code>score = 10;</code>
<code>===</code>	Strict equality	Same type and same value? Gives a boolean	<code>score === 10</code>
<code>!==</code>	Strict inequality	Different type or value? Gives a boolean	<code>score !== 10</code>
<code>==</code>	Loose equality (do not use)	Converts the types first, then compares	<code>"5" == 5</code> is true

KEY IDEA

`a === b` is true only when `a` and `b` have the same type and the same value, and `a !== b` is its opposite. `==` converts types before comparing, with surprising results, so always use `===` and `!==`. Never use `===` on the result of decimal arithmetic.

WATCH OUT

One = where you meant ===. A single `=` stores a value; it does not compare. In a comparison, it is easy to type one `=` too few:

single-equals.ts

```
1 const total = 12;
2 console.log(total = 12);
```

```
single-equals.ts:2:13 - error TS2588: Cannot assign to 'total' because it is a
constant.
```

The writer meant "is `total` equal to 12?", but `total = 12` tries to store 12 in `total`, and a `const` can never be assigned again. Here the checker catches the slip only because `total` is a constant; 4.8 shows what happens with a `let`. Fix it with `total === 12`, or delete the file.

PAUSE AND TRY**Question 3** Tracing

Predict exactly what this program shows.

predict-equality.ts

```
1 const word = "Tea";
2 const count = 2 + 2;
3 console.log(count === 4);
4 console.log(count !== 4);
5 console.log(word.toLowerCase() === "tea");
6 console.log(word === "Tea");
7 console.log(word.length === 3);
```

Question 4 Multiple choice

Which line does `npm tsc` accept **and** shows `true` when run?

- A. `console.log("7" == 7);`
- B. `console.log(0.1 + 0.2 === 0.3);`
- C. `console.log((0.1 + 0.2).toFixed(1) === "0.3");`
- D. `console.log("7" === 7);`

4.3 Impossible Comparisons

You have now seen TS2367 twice. It is one of the most useful messages the checker gives, because it catches a comparison that can **never** be true. A condition like that is always a bug: either the code was meant to compare something else, or a value has the wrong type.

Here is the most common cause. A program holds an age typed as text, and compares it with a number:

age-text.ts

```
1 const typedAge = " 16 ".trim();
2 const minimumAge = 16;
3 console.log(typedAge === minimumAge);
```

age-text.ts:3:13 - error TS2367: This comparison appears to be unintentional because the types 'string' and 'number' have no overlap.

Take the message apart:

Part of the message	What it means
age-text.ts:3:13	The comparison starts on line 3, column 13
This comparison appears to be unintentional	The checker thinks you did not mean it, because of the next part
the types 'string' and 'number'	The type of the left side, then the type of the right side
have no overlap	No value has both types, so the two sides can never be equal

Two types **overlap** when some value belongs to both of them. No value is both a string and a number, so `typedAge === minimumAge` is `false` every time it runs, whatever the age. The checker reports it, instead of letting your `if` quietly never run. The fix is the one from Day 2: convert the text first.

age-text-fixed.ts

```
1 const typedAge = " 16 ".trim();
2 const minimumAge = 16;
3 console.log(Number(typedAge) === minimumAge);
```

true

`Number(typedAge)` is the number 16, so both sides are numbers and the comparison can be true. If you typed `age-text.ts`, fix or delete it now.

WORKED EXAMPLE A constant that can never be Sunday

Now vary it: both sides are strings, so the types are the same. Will the checker accept this?

today.ts

```
1 const today = "Saturday";
2 console.log(today === "Sunday");
```

today.ts:2:13 - error TS2367: This comparison appears to be unintentional because the types '"Saturday"' and '"Sunday"' have no overlap.

Look at the types in the message: not 'string', but '"Saturday"' and '"Sunday"'. In 2.5 you saw that hovering over a `const` shows the value itself as its type, such as `const city: "Lima"`, because a constant can never hold anything else. So here, the checker knows that `today` is always exactly `"Saturday"`, and it can never be equal to `"Sunday"`. (Types made of one exact

value are a preview of Day 9, which teaches them fully.) The message shows exact values like this only when both sides are the same kind of value, as here, where both are strings. When the two sides are different kinds, such as text and a number, the message names just the kinds, 'string' and 'number', even for constants, as in `loose-equals.ts`.

The checker is right: in this program, the comparison is always `false`. But it is still a problem for you, because programs in this course stand in for real ones. Until Day 13, a program cannot read what a person types, so the value in a variable like `today` stands for "whatever day it is when the program runs." To tell the checker that, give the constant a type annotation, as in Day 2:

`today-fixed.ts`

```
1 const today: string = "Saturday";
2 console.log(today === "Sunday");
3 console.log(today === "Saturday");
```

```
false
true
```

`const today: string` says "today is some string," so the checker no longer assumes it is always "Saturday", and both comparisons are accepted. A `let` has the same effect without an annotation: hover over `let day = "Saturday";` and VS Code shows `let day: string`, because a `let` can be given another string later. If you typed `today.ts`, fix or delete it now.

From here on, whenever a program's starting value stands for something that could be different each time it runs, such as a typed command, a score, or a day, this course writes an annotation on it: `const score: number = 72;`. A value that comes out of a method, such as `typedDay.trim()`, needs none: the checker gives a method's result the plain type `string`, as it did for `typedAge` above.

KEY IDEA

TS2367 means the two sides of `===` or `!==` have types with no overlap, so the comparison can never be true. Convert a value that has the wrong type. When a constant stands for a value that could vary, annotate it, as in `const today: string = "Saturday";`.

WATCH OUT

Comparing a result with the wrong kind of value. Day 3's `indexOf` gives a **number**, so comparing it with text can never match:

`at-sign.ts`

```
1 const email = "ana.lee@example.com";
2 console.log(email.indexOf("@") === "-1");
```

```
at-sign.ts:2:13 - error TS2367: This comparison appears to be unintentional because the types 'number' and 'string' have no overlap.
```

"-1" in quotes is a string. The fix is `email.indexOf("@") === -1`, without quotes. When you see TS2367, ask which side has the wrong type, and fix that side. Fix or delete the file before your next check.

PAUSE AND TRY

Question 5 Tracing

Which lines does `npx tsc` reject? Write each message, and say what each accepted line shows when the program is fixed and run.

predict-overlap.ts

```
1 const city: string = "Lima";
2 const visitors = 120;
3 const capital = "Lima";
4 console.log(city === "Paris");
5 console.log(visitors === "120");
6 console.log(capital === "Paris");
7 console.log(String(visitors) === "120");
```

4.4 Logical Operators: And, Or, Not

Many rules have more than one part. A ride lets you on if you are tall enough **and** old enough. A shop is busy on Saturday **or** Sunday. You may go out if it is **not** raining. Each part is a comparison you already know how to write; what you need is a way to combine them.

The **logical operators** combine booleans into a new boolean:

- `a && b` (read "a and b") is `true` only when **both** sides are true.
- `a || b` (read "a or b") is `true` when **at least one** side is true. The `|` key is usually above Enter, typed with Shift.
- `!a` (read "not a") flips one boolean: `!true` is `false`, and `!false` is `true`.

A table of every possible combination is called a **truth table**:

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

Notice that `||` is true when both sides are true. In everyday speech, "tea or coffee?" usually means one or the other, but in code, "or" includes both.

WORKED EXAMPLE Rules with more than one part

logic.ts

```

1 const height: number = 135;
2 const age: number = 7;
3 console.log(height >= 120 && age >= 8);
4 console.log(height >= 120 || age >= 8);
5 const day: string = "Sunday";
6 console.log(day === "Saturday" || day === "Sunday");
7 const isRaining: boolean = true;
8 console.log(!isRaining);
9 console.log(!(age >= 8));

```

```

false
true
true
false
true

```

Line	Expression	Worked out	Result
3	height >= 120 && age >= 8	true && false	false
4	height >= 120 age >= 8	true false	true
6	day === "Saturday" day === "Sunday"	false true	true
8	!isRaining	!true	false
9	!(age >= 8)	!false	true

Each comparison is worked out first, then the logical operator combines the two booleans. On line 3, the child is tall enough but not old enough, and && needs both. On line 6, each side compares `day` with one value; you must write `day ===` on **both** sides, because `day === "Saturday" || "Sunday"` would not compare `day` with `"Sunday"` at all. On line 9, the parentheses make `age >= 8` work out first, and `!` then flips its answer.

Where do the logical operators go in the precedence order? `!` belongs to the value right after it, so it is applied before anything else: that is why line 9 needs parentheses. Then come arithmetic, then the comparisons, then &&, and last `||`.

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

Because && comes before ||, mixing them gives different answers depending on the parentheses. A park lets you in free if you are a senior, or if you are a member on a holiday:

park-precedence.ts

```
1 const age: number = 70;
2 const isMember: boolean = false;
3 const isHoliday: boolean = false;
4 console.log(age >= 65 || isMember && isHoliday);
5 console.log((age >= 65 || isMember) && isHoliday);
```

```
true
false
```

On line 4, `&&` goes first: `isMember && isHoliday` is `false && false`, which is `false`, and then `true || false` is `true`. The 70-year-old gets in. On line 5, the parentheses make `||` go first: `true || false` is `true`, and then `true && isHoliday` is `false`. Now the rule says "a senior or a member, and only on a holiday," which is a different rule. When you mix `&&` and `||`, add parentheses so that the code says exactly what the English rule says.

KEY IDEA

`a && b` is true only when both sides are true, `a || b` when at least one is, and `!a` flips a boolean. `!` is applied first, comparisons before `&&`, and `&&` before `||`; use parentheses whenever you mix `&&` and `||`.

WATCH OUT

Writing a range the way you would on paper. In math, $0 \leq \text{score} \leq 100$ means "between 0 and 100." In TypeScript, it does not:

chain-wrong.ts

```
1 const score: number = 105;
2 console.log(0 <= score <= 100);
```

```
chain-wrong.ts:2:13 - error TS2365: Operator '<=' cannot be applied to types
'boolean' and 'number'.
```

Comparisons are worked out left to right, one pair at a time, as Day 2 taught for operators on the same row. So `0 <= score` is worked out first, giving the boolean `true`, and then the second `<=` would compare `true <= 100`, a boolean with a number. Run with `node` anyway, it shows `true` even though 105 is not in the range, because JavaScript converts `true` to 1. Fix or delete `chain-wrong.ts`, and write a range as two comparisons joined with `&&`:

score-range.ts

```
1 const score: number = 105;
2 console.log(0 <= score && score <= 100);
```

```
false
```

`0 <= 105` is true, `105 <= 100` is false, and `true && false` is false.

PAUSE AND TRY**Question 6** Tracing

Predict exactly what this program shows.

predict-logic.ts

```
1 const temperature: number = 22;
2 const isSunny: boolean = false;
3 console.log(temperature > 20 && isSunny);
4 console.log(temperature > 20 || isSunny);
5 console.log(!isSunny && temperature < 25);
6 console.log(!(temperature > 20 && temperature < 25));
```

Question 7 Multiple choice

A room feels comfortable when its temperature is from 18 to 25 degrees, including 18 and 25. Which condition is `true` exactly when the room is comfortable?

- A. `18 <= temperature <= 25`
- B. `temperature >= 18 || temperature <= 25`
- C. `temperature >= 18 && temperature <= 25`
- D. `temperature > 18 && temperature < 25`

4.5 Short-Circuit Evaluation

Look again at the truth table. If the left side of `&&` is `false`, the answer is `false` whatever the right side is. If the left side of `||` is `true`, the answer is `true` whatever the right side is. JavaScript uses this: it works out the left side first, and when that already decides the answer, it **does not work out the right side at all**. Skipping the right side like this is called **short-circuit evaluation**.

That sounds like a small speed-up, but it matters for correctness. In 3.4 you saw that reading a character past the end of a string gives `undefined`, and that calling a method on `undefined` crashes the program. Short-circuiting lets the left side check that it is safe to work out the right side:

WORKED EXAMPLE Checking before you look

The program asks whether a nickname starts with A. The nickname is empty here, as if the person left the box blank.

short-circuit.ts

```
1 const nickname: string = "";
2 console.log(nickname.length > 0 && nickname[0].toUpperCase() === "A");
3 console.log(nickname.length === 0 || nickname[0].toUpperCase() === "A");
```

false
true

Line	Left side	Right side worked out?	Result
2	<code>nickname.length > 0</code> is false	No: <code>&&</code> is already false	false

3	nickname.length === 0 is true	No: is already true	true
---	-------------------------------	------------------------	------

On line 2, `nickname[0]` would be `undefined`, and `undefined.toUpperCase()` would crash. But the left side is `false`, so `&&` stops there and the risky right side never runs. Line 3 asks the opposite question with `||`, "is it empty, or does it start with A?", and stops as soon as the left side is `true`.

Now vary the order. Put the risky part first:

short-circuit-crash.ts

```
1 const nickname: string = "";
2 console.log(nickname[0].toUpperCase() === "A" && nickname.length > 0);
```

`npx tsc` reports no errors for this file, but `node short-circuit-crash.ts` stops with:

```
TypeError: Cannot read properties of undefined (reading 'toUpperCase')
```

The left side runs first, `nickname[0]` is `undefined`, and the program crashes before `&&` ever gets to the length check. The checker cannot help here: as 3.4 explained, it treats `nickname[0]` as a `string` whatever the index. So with `&&` and `||`, the order of the two sides matters, even though the truth table gives the same answer either way. Put the safety check **first**.

The checker uses short-circuit reasoning too. It knows the right side of `||` runs only when the left side was `false`, and it can use that to spot a condition that is always true:

weekday-wrong.ts

```
1 const day: string = "Sunday";
2 console.log(day !== "Saturday" || day !== "Sunday");
```

```
weekday-wrong.ts:2:35 - error TS2367: This comparison appears to be unintentional
because the types '"Saturday"' and '"Sunday"' have no overlap.
```

The writer meant "is it a weekday?", that is, "not Saturday and not Sunday." Column 35 is the **second** comparison. The checker reasons like this: the right side of `||` only runs when `day !== "Saturday"` was `false`, which means `day` **is** `"Saturday"`, so comparing it with `"Sunday"` cannot tell anything new. In fact the whole condition is true for every day, because no day is both Saturday and Sunday. (The checker following your conditions to learn more about a value is a preview of Day 9, which calls it narrowing.) Fix or delete `weekday-wrong.ts`. The English "not Saturday and not Sunday" uses "and":

weekday.ts

```
1 const day: string = "Sunday";
2 console.log(day !== "Saturday" && day !== "Sunday");
```

```
false
```

For Sunday, `day !== "Saturday"` is true, but `day !== "Sunday"` is false, so `&&` gives `false`: Sunday is not a weekday.

KEY IDEA

`&&` and `||` work out the left side first and skip the right side when the left side already decides the answer. Put a safety check, such as `text.length > 0`, on the left of the part it protects.

WATCH OUT

Translating "not this or not that" word by word. `x !== a || x !== b` is true for every value of `x`, because no value equals both `a` and `b`. When you mean "neither of these," write `x !== a && x !== b`. When both are strings, the checker usually catches the mistake with TS2367, as above.

PAUSE AND TRY**Question 8** Tracing

Predict what this program shows. Does it reach line 5?

predict-short.ts

```
1 const code: string = "";
2 console.log(code.length >= 2 && code[1].toUpperCase() === "B");
3 console.log(code.length < 2 || code[1].toUpperCase() === "B");
4 console.log(code[1].toUpperCase() === "B" || code.length < 2);
5 console.log("Done");
```

4.6 if Statements and Blocks

A boolean on its own only shows `true` or `false`. To make the program **do** something different, you need a statement that runs some lines only when a condition is true. Choosing which code runs based on a condition is called **selection**, and the basic tool for it is the `if` statement.

```
if (condition) {
  statements that run only when the condition is true
}
```

Piece	What it is
<code>if</code>	The keyword that starts the statement
<code>(condition)</code>	Any expression that gives true or false, such as a comparison. The parentheses are required
<code>{ and }</code>	Curly braces that mark the block : the statements that belong to the <code>if</code>

A **condition** is the expression that decides whether the block runs. When the program reaches the `if`, it works out the condition. If it is `true`, the statements in the block run in order, and then the program carries on after the closing `}`. If it is `false`, the whole block is skipped and the program goes straight on after the `}`. Either way, the lines after the block always run.

WORKED EXAMPLE A hot day

hot-day.ts

```
1 const temperature: number = 31;
2 console.log(`It is ${temperature} degrees.`);
3 if (temperature > 30) {
4   console.log("It is hot today.");
5   console.log("Drink plenty of water.");
6 }
7 console.log("Have a good day!");
```

```
It is 31 degrees.
It is hot today.
Drink plenty of water.
Have a good day!
```

Line	What happens
2	Shows the temperature
3	Works out <code>31 > 30</code> , which is <code>true</code> , so the block runs
4–5	The two lines in the block run, in order
7	After the block's <code>}</code> , the program carries on and shows the last line

The two statements inside the braces are indented by two spaces. TypeScript does not need the indentation (only the braces matter to it), but people do: it shows at a glance which lines belong to the `if`. VS Code adds it for you when you press Enter after `{`. There is no semicolon after the closing `}`, because the braces already mark where the statement ends.

Now vary the temperature, and change nothing else:

cool-day.ts

```
1 const temperature: number = 24;
2 console.log(`It is ${temperature} degrees.`);
3 if (temperature > 30) {
4   console.log("It is hot today.");
5   console.log("Drink plenty of water.");
6 }
7 console.log("Have a good day!");
```

```
It is 24 degrees.
Have a good day!
```

`24 > 30` is `false`, so the program skips lines 4 and 5 entirely, and goes from line 3 straight to line 7. This is the first program you have seen where some lines do not run. The code is identical apart from one value; only the condition's answer differs.

You may meet code that leaves out the braces when the block is a single line, such as `if (temperature > 30) console.log("Hot");`. That works, but only for exactly one statement, and adding a second line later silently leaves it outside the `if`. This course always writes the braces.

KEY IDEA

`if (condition) { ... }` runs the block only when the condition is true; otherwise the program skips it. Either way, the program carries on after the closing `}`. Always write the braces, and indent the block.

WATCH OUT

Using a variable outside the block that made it. A variable declared inside a block exists only inside that block:

advice-scope.ts

```
1 const temperature: number = 31;
2 if (temperature > 30) {
3   const advice = "Drink plenty of water.";
4 }
5 console.log(advice);
```

advice-scope.ts:5:13 - error TS2304: Cannot find name 'advice'.

The part of a program where a name can be used is called its **scope**. A `const` or `let` declared inside braces has the block as its scope: it is made when the block runs and is gone after the `}`. So on line 5, outside the block, the name `advice` does not exist, and you get Day 1's TS2304. The rule makes sense: if the condition were false, `advice` would never have been made, so there would be nothing to show. Use the variable inside the block, or 4.7 shows how to declare it before the `if` so that every path gives it a value. Fix or delete the file before your next check.

PAUSE AND TRY**Question 9** Tracing

Predict exactly what this program shows. Work out each condition on its own; the three `if` statements are separate.

predict-if.ts

```
1 const balance: number = 40;
2 const price: number = 25;
3 if (balance >= price) {
4   console.log("Purchase approved.");
5 }
6 if (balance - price < 20) {
7   console.log("Your balance is low.");
8 }
9 if (price > balance) {
10  console.log("Not enough money.");
11 }
12 console.log(`Balance: ${balance}`);
```

4.7 else and else if

An `if` alone handles "do this, or do nothing." Often you need "do this, **otherwise** do that": take an umbrella, otherwise leave it at home. And some decisions have more than two outcomes: free shipping, cheap shipping, or full price.

Adding `else` and a second block after an `if` gives the program two paths. When the condition is true, the `if` block runs; when it is false, the `else` block runs. Exactly one of the two always runs, never both and never neither.

umbrella.ts

```
1 const chanceOfRain: number = 70;
2 if (chanceOfRain >= 50) {
3   console.log("Take an umbrella.");
4 } else {
5   console.log("Leave the umbrella at home.");
6 }
```

Take an umbrella.

`70 >= 50` is true, so the first block runs and the `else` block is skipped. With `chanceOfRain` set to 20, the condition would be false, and only the `else` block would run. Notice the layout: `} else {` on one line, which closes the first block and opens the second.

Day 3 promised a way to deal with `indexOf` giving -1. Here it is: test for -1, and do something sensible in each case.

full-name.ts

```
1 const fullName: string = "Cher";
2 const space = fullName.indexOf(" ");
3 if (space === -1) {
4   console.log(`One name: ${fullName}`);
5 } else {
6   console.log(`First: ${fullName.slice(0, space)}`);
7   console.log(`Last: ${fullName.slice(space + 1)}`);
8 }
```

One name: Cher

"Cher" has no space, so `space` is -1, the condition is true, and the program says so instead of slicing the name into nonsense, as it did in 3.8. With "Ada Lovelace", `space` would be 3, the condition false, and the `else` block would show the two parts.

For more than two outcomes, put another `if` straight after `else`. `else if (condition)` adds another question that is asked only when every condition above it was false. An `if`, any number of `else ifs`, and an optional final `else` together make an **if chain**. The program asks the conditions from the top, one at a time; the **first** one that is true has its block run, and the rest of the chain is skipped. If none is true, the `else` block runs.

WORKED EXAMPLE Shipping cost in three bands

A shop charges nothing for shipping on orders of \$50 or more, \$4.99 on orders of \$25 or more, and \$7.99 otherwise.

shipping.ts

```
1 const orderTotal: number = 42.5;
2 let shipping: number;
3 if (orderTotal >= 50) {
4   shipping = 0;
5 } else if (orderTotal >= 25) {
6   shipping = 4.99;
7 } else {
8   shipping = 7.99;
9 }
10 console.log(`Order: ${orderTotal.toFixed(2)}`);
11 console.log(`Shipping: ${shipping.toFixed(2)}`);
```

Order: \$42.50
Shipping: \$4.99

Line	Condition	Worked out	What happens
3	orderTotal >= 50	42.5 >= 50	false: ask the next condition
5	orderTotal >= 25	42.5 >= 25	true: run this block
6	(block)	shipping = 4.99	The rest of the chain is skipped
10	after the chain		Shows both amounts

The `else if` on line 5 does not need to say "and less than 50": the program only gets there when the first condition was `false`, so it already knows the order is under \$50. That is why the bands go from the highest to the lowest.

Look at line 2: `let shipping: number;` declares the variable **before** the chain, with an annotation and no starting value, as you saw in 2.7. Each block then assigns it. Because it was declared outside the blocks, it is still in scope on line 11, unlike `advice` in 4.6. And because the chain ends with `else`, exactly one block always runs, so the checker can see that `shipping` always has a value by line 11.

KEY IDEA

An if chain (`if`, then any `else ifs`, then an optional `else`) runs the block of the first condition that is true and skips the rest; `else` runs when none is true. Order the conditions so that each one only has to handle what the ones above it did not.

WATCH OUT

Putting the widest condition first. The first true condition wins, so the order of the rungs matters:

grade-wrong-order.ts

```
1 const score: number = 95;
2 if (score >= 60) {
3   console.log("Grade: D");
4 } else if (score >= 70) {
5   console.log("Grade: C");
6 } else if (score >= 90) {
7   console.log("Grade: A");
8 } else {
9   console.log("Grade: F");
10 }
```

Grade: D

A score of 95 should be an A, but `95 >= 60` is `true`, so the first block runs and the chain stops before it ever asks `score >= 90`. In fact this chain gives a D to every score of 60 or more.

Neither `npx tsc` nor `node` reports anything. Put the narrowest condition first: `score >= 90`, then `score >= 80`, and so on down.

Leaving out the `else`, then using the variable. Without a final `else`, there is a path where no block runs. The checker follows every path:

shipping-no-else.ts

```
1 const orderTotal: number = 42.5;
2 let shipping: number;
3 if (orderTotal >= 50) {
4   shipping = 0;
5 } else if (orderTotal >= 25) {
6   shipping = 4.99;
7 }
8 console.log(`Shipping: ${shipping.toFixed(2)}`);
```

shipping-no-else.ts:8:27 - error TS2454: Variable 'shipping' is used before being assigned.

This is Day 2's TS2454 (column 27 is `shipping` inside the placeholder). For an order under \$25, neither condition is true, nothing assigns `shipping`, and line 8 would read a variable that holds nothing. Adding the `else` block, as in `shipping.ts`, fixes it. Fix or delete this file before your next check.

PAUSE AND TRY

Question 10 Tracing

Predict exactly what this program shows.

predict-chain.ts

```
1 const speed: number = 64;
2 const limit: number = 50;
3 const over = speed - limit;
4 if (over <= 0) {
5   console.log("No ticket");
6 } else if (over <= 10) {
7   console.log("Warning");
8 } else if (over <= 20) {
9   console.log("Small fine");
10 } else {
11   console.log("Large fine");
12 }
13 console.log(`Over by ${over}`);
```

Question 11 Debugging

This program should show a library fine: \$5 for more than a week late, 50 cents a day for 1 to 7 days, and nothing otherwise. `npm tsc` rejects it. Predict the message, explain it, and write the fix.

library-fine.ts

```
1 const daysLate: number = 3;
2 let fine: number;
3 if (daysLate > 7) {
4   fine = 5;
5 } else if (daysLate > 0) {
6   fine = daysLate * 0.5;
7 }
8 console.log(`Fine: ${fine.toFixed(2)}`);
```

4.8 Truthy and Falsy Values

So far every condition has been a boolean. But a program often wants to ask a simpler question: "is there anything here?" Did the person type a nickname, or leave it empty? JavaScript lets you put **any** value in a condition, and decides whether it counts as true or false.

A value that counts as `false` in a condition is **falsy**. There are exactly six to remember:

Falsy value	What it is
<code>false</code>	The boolean
<code>0</code>	The number zero
<code>""</code>	The empty string
<code>null</code>	Deliberately empty
<code>undefined</code>	Not assigned yet
<code>NaN</code>	The failed number from 2.10

(JavaScript has a few rarer falsy values, such as `-0`, that this course does not use.) **Every other value is truthy**: it counts as `true`. That includes every number except zero, so `-1` is truthy, and every string with at least one character, so `"0"`, `" "`, and even `"false"` are truthy.

Why these six? Each one means "nothing" or "no answer" in its own type: no number, no text, no value. So `if (nickname)` reads as "if there is a nickname." To see how a value will count, use `Boolean(value)`, which converts any value to `true` or `false` the same way a condition does, just as Day 2's `Number()` and `String()` convert to numbers and strings.

WORKED EXAMPLE Which values count as true?

truthy-falsy.ts

```
1 console.log(Boolean(false), Boolean(0), Boolean(""));
2 console.log(Boolean(null), Boolean(undefined), Boolean(NaN));
3 console.log(Boolean(42), Boolean(-1), Boolean("0"));
4 console.log(Boolean(" "), Boolean("false"), Boolean(true));
```

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

Lines 1 and 2 are the six falsy values. Lines 3 and 4 are all truthy: `42` and `-1` are numbers other than zero, and `"0"`, `" "`, and `"false"` are strings with at least one character, whatever those characters are. A string is falsy only when it is completely empty.

Now put a string straight into a condition, and into `||`:

nickname.ts

```
1 const nickname: string = "";
2 if (nickname) {
3   console.log(`Hi, ${nickname}!`);
4 } else {
5   console.log("Hi, guest!");
6 }
7 const shownName = nickname || "guest";
8 console.log(shownName);
9 console.log(`[${nickname} && nickname.toUpperCase()]`);
10 const friend: string = "Kim";
11 console.log(friend || "guest");
12 console.log(friend && friend.toUpperCase());
```

```
Hi, guest!
guest
[]
Kim
KIM
```

The empty nickname is falsy, so line 2's condition counts as false and the `else` block runs. Lines 7 to 12 show something new about `&&` and `||`: they do not always give a boolean. They give back **one of their two sides**, whichever one decided the answer:

Expression	Rule	Here gives
<code>a b</code>	a if a is truthy, otherwise b	
<code>nickname "guest"</code>	<code>""</code> is falsy, so the right side	<code>"guest"</code>
<code>friend "guest"</code>	<code>"Kim"</code> is truthy, so the left side	<code>"Kim"</code>
<code>a && b</code>	a if a is falsy, otherwise b	
<code>nickname && ...</code>	<code>""</code> is falsy, so <code>""</code> itself (shown as <code>[]</code>)	<code>""</code>
<code>friend && friend.toUpperCase()</code>	<code>"Kim"</code> is truthy, so the right side	<code>"KIM"</code>

With booleans on both sides, this is exactly the truth table from 4.4. With other values, `name || "guest"` is a short way to say "the name, or "guest" if it is empty," and the result has the type `string`.

Day 2 also promised ways to check for two special values. `NaN` is falsy, but so is `0`, so `if (!seats)` cannot tell a failed conversion from a real zero. And `NaN` is the one value that is not equal to anything, not even itself, so `=== NaN` never works; the checker reports it:

nan-equals.ts

```
1 const seats = Number("forty");
2 console.log(seats === NaN);
```

nan-equals.ts:2:13 - error TS2845: This condition will always return 'false'.

The full report even adds a hint below the code: Did you mean `'Number.isNaN(seats)'`?

`Number.isNaN(value)` is a function that gives `true` only when `value` is `NaN`. For `null`, `typeof` cannot help (`typeof null` is `"object"`), but `===` can, because `null` is equal to itself. Fix or delete `nan-equals.ts`, then:

nan-check.ts

```
1 const seats = Number("forty");
2 if (Number.isNaN(seats)) {
3   console.log("That is not a number.");
4 } else {
5   console.log(`Seats: ${seats}`);
6 }
7 const middleName = null;
8 console.log(middleName === null);
9 console.log(typeof middleName);
```

```
That is not a number.
true
object
```

`Number("forty")` is `NaN`, so `Number.isNaN(seats)` is `true`, and the program reports the bad text instead of doing arithmetic with it. Line 8 checks for `null` directly with `===`, and line 9 shows why `typeof` would not have worked. A variable that holds a number **or** `null` needs a type from Day 9, which also shows how to check it; for now, know that `=== null` is the way.

KEY IDEA

In a condition, `false`, `0`, `""`, `null`, `undefined`, and `NaN` are falsy; every other value is truthy, including `"0"` and `" "`. `a || b` gives `a` if it is truthy, otherwise `b`. Check for `NaN` with `Number.isNaN(value)` and for `null` with `=== null`.

WATCH OUT

Using `||` for a default when `0` is a real value. A music app should start at the saved volume, or at 50 if there is none:

volume.ts

```
1 const volume: number = 0;
2 const level = volume || 50;
3 console.log(`Volume: ${level}`);
```

Volume: 50

The person had muted the app: the saved volume is 0. But 0 is falsy, so `volume || 50` gives 50, and the app starts loud. `||` cannot tell "nothing" from zero or from an empty string. Use it for a default only when zero or `""` really means "nothing here." (Day 9 teaches `??`, which gives a default only for `null` and `undefined`.)

= instead of `===` in a condition. With a `let`, the checker cannot save you as it did in 4.2:

perfect-score.ts

```
1 let score: number = 75;
2 if (score = 100) {
3   console.log("Perfect score!");
4 }
5 console.log(score);
```

Perfect score!
100

`score = 100` **assigns** 100 to `score`, and an assignment is an expression whose value is the value assigned. So the condition is 100, which is truthy, and the block runs for any score. Line 5 shows the damage: the score itself was changed. Neither `npx tsc` nor `node` reports anything. When an `if` runs every time, check its condition for a single `=`.

PAUSE AND TRY**Question 12** Tracing

Predict exactly what this program shows.

predict-truthy.ts

```
1 const items: number = 0;
2 const note: string = " ";
3 const label: string = "";
4 if (items) {
5   console.log("Items in cart");
6 }
7 if (note) {
8   console.log("Note added");
9 }
10 if (!label) {
11   console.log("No label");
12 }
13 console.log(label || "untitled");
14 console.log(Boolean("0"));
```

Question 13 Multiple choice

Which of these values is truthy?

- A. 0
- B. ""
- C. "0"
- D. NaN

4.9 The Conditional Operator

A shopping cart should say 1 item but 3 items. With an if chain, choosing one of two words takes six lines:

plural-if.ts

```
1 const itemCount: number = 3;
2 let word: string;
3 if (itemCount === 1) {
4   word = "item";
5 } else {
6   word = "items";
7 }
8 console.log(`${itemCount} ${word} in your cart`);
```

3 items in your cart

That works, but all it does is pick one of two values. For exactly this job, JavaScript has an operator that picks a value in one expression: the **conditional operator**, `condition ? a : b`. It works out the condition; if it is truthy, the whole expression gives `a`, otherwise it gives `b`. (It is also called the **ternary operator**, because it is the only operator with three parts.)

`condition ? valueIfTrue : valueIfFalse`

Piece	What it is
condition	Any condition, as in an <code>if</code> ; no parentheses are needed
<code>?</code>	Separates the condition from the two choices
<code>valueIfTrue</code>	The value the expression gives when the condition is truthy
<code>:</code>	Separates the two choices
<code>valueIfFalse</code>	The value the expression gives when the condition is falsy

WORKED EXAMPLE Picking a word

plural.ts

```
1 const itemCount: number = 1;
2 const word = itemCount === 1 ? "item" : "items";
3 console.log(`${itemCount} ${word} in your cart`);
4 const points: number = 7;
5 console.log(`${points} is ${points % 2 === 0 ? "even" : "odd"}`);
```

```
1 item in your cart
7 is odd
```

On line 2, `itemCount === 1` is `true`, so the expression gives `"item"`, and that is stored in `word`. Because the conditional operator gives a value, `word` can be a `const`, and the checker works out its type from the two choices. (Hover over `word` and VS Code shows both words, `"item" | "items"`; Day 9 explains that kind of type.) In `plural-if.ts`, `word` had to be a `let` declared with an annotation, because an `if` is a statement: it runs blocks, but it does not give back a value.

Line 5 puts a conditional operator straight inside a template literal's placeholder. `7 % 2` is 1, Day 2's remainder, so `7 % 2 === 0` is `false`, and the placeholder becomes `"odd"`.

Use the conditional operator when you are choosing between two **values**. When each choice needs to **do** several things, such as show two lines or change two variables, use an `if` chain. And although you can put one conditional operator inside another, the result is hard to read; three or more choices belong in an `if` chain.

KEY IDEA

`condition ? a : b` is an expression that gives `a` when the condition is truthy and `b` otherwise. Use it to pick one of two values; use an `if` chain when the choices need statements or there are more than two.

WATCH OUT

Forgetting that `?` comes almost last in the precedence order. The conditional operator is applied after everything else, even `+`:

cart-message-wrong.ts

```
1 const count: number = 1;
2 console.log("You have " + count === 1 ? "one item" : "several items");
```

```
cart-message-wrong.ts:2:13 - error TS2367: This comparison appears to be unintentional because the types 'string' and 'number' have no overlap.
```

The writer meant "You have" followed by one of two phrases. But `+` comes before `===`, so the program would first join "You have " + `count` into the string "You have 1", and then compare that string with the number 1: TS2367 from 4.3. (Run anyway, it shows just `several items`.) Fix or delete this file. Put the whole conditional in parentheses:

cart-message.ts

```
1 const count: number = 1;
2 console.log("You have " + (count === 1 ? "one item" : "several items"));
```

You have one item

PAUSE AND TRY

Question 14 Tracing

Predict exactly what this program shows.

predict-conditional.ts

```
1 const age: number = 12;
2 const ticket = age < 13 ? "child" : "adult";
3 const price = age < 13 ? 6 : 10;
4 console.log(`${ticket}: ${price}`);
5 const temperature: number = -3;
6 console.log(temperature < 0 ? "Freezing" : "Above freezing");
7 console.log(`${age} >= 18 ? "Can" : "Cannot" vote`);
```

4.10 switch and break

A traffic light program checks one value, `light`, against three fixed choices: "green", "yellow", and "red". An if chain works, but it repeats `light ===` on every rung. When a program compares **one** value with a list of fixed values, the `switch` statement says it more directly.

```
switch (value) {
  case choice1:
    statements
    break;
  case choice2:
    statements
    break;
  default:
    statements
}
```

Piece

What it is

<code>switch (value)</code>	The value to check, worked out once
<code>case choice:</code>	A fixed value to compare with, using <code>===</code> , followed by a colon
<code>statements</code>	What to do when that case matches
<code>break;</code>	Leaves the <code>switch</code> and carries on after its closing <code>}</code>
<code>default:</code>	Runs when no case matches, like <code>else</code> . Optional; usually last, and needs no <code>break</code>

WORKED EXAMPLE A traffic light

traffic-light.ts

```
1 const light: string = "yellow";
2 switch (light) {
3   case "green":
4     console.log("Go");
5     break;
6   case "yellow":
7     console.log("Slow down");
8     break;
9   case "red":
10    console.log("Stop");
11    break;
12  default:
13    console.log("Unknown light");
14 }
15 console.log("Done");
```

Slow down
Done

Step by step: line 2 works out `light`, which is `"yellow"`. The program compares it with each case from the top, using `===`. `"yellow" === "green"` is `false`, so it moves on;

`"yellow" === "yellow"` is `true`, so it starts running the statements after that case, line 7. Line 8's `break` jumps out of the whole `switch`, to line 15, which shows `Done`. The `red` and `default` cases are never looked at. If `light` were `"blue"`, no case would match and the `default` statements would run.

Notice the `: string` on line 1. `switch` compares with `===`, so 4.3's rule applies: without the annotation, the checker would know `light` is always `"yellow"` and reject the other cases (you will see the message below).

What does `break` really do? A matching case is only a **starting point**: the program runs every statement from there on, down through the following cases, until it meets a `break` or the end of the `switch`. Running on into the next case is called **fall-through**. Usually you stop it with `break`, but you can use it on purpose to give several cases the same statements:

day-type.ts

```
1 const day: string = "sunday";
2 switch (day) {
3   case "saturday":
4   case "sunday":
5     console.log("Weekend");
6     break;
7   default:
8     console.log("Weekday");
9 }
```

Weekend

"sunday" matches line 4. Line 3's case "saturday": has no statements of its own, so it falls through to the same lines: both days run line 5, then break. Stacking cases like this is the switch way of writing `||`.

KEY IDEA

`switch (value)` compares one value with each `case` using `===` and runs the statements from the first match until a `break`. `default` runs when nothing matches. Without a `break`, the program falls through into the next case.

WATCH OUT

Forgetting a break. Fall-through happens whether you want it or not:

traffic-light-no-break.ts

```
1 const light: string = "yellow";
2 switch (light) {
3   case "green":
4     console.log("Go");
5     break;
6   case "yellow":
7     console.log("Slow down");
8   case "red":
9     console.log("Stop");
10    break;
11  default:
12    console.log("Unknown light");
13 }
14 console.log("Done");
```

Slow down
Stop
Done

The yellow case has no `break`, so after showing `Slow down`, the program falls through into the red case's statement and shows `Stop` too, until the `break` on line 10. Neither `npx tsc` nor `node`

reports anything, because fall-through is allowed. End every case with `break` unless you mean to stack it with the next one.

A switch on a constant with no annotation. The checker compares each `case` with the value's type, just as it does for `===`:

light-literal.ts

```
1 const light = "yellow";
2 switch (light) {
3   case "green":
4     console.log("Go");
5     break;
6   case "yellow":
7     console.log("Slow down");
8     break;
9   case "red":
10    console.log("Stop");
11    break;
12 }
```

light-literal.ts:3:8 - error TS2678: Type '"green"' is not comparable to type '"yellow"'.

light-literal.ts:9:8 - error TS2678: Type '"red"' is not comparable to type '"yellow"'.

TS2678 is the `switch` version of TS2367: "not comparable" means the `case` value can never match. `light` is a constant that is always `"yellow"`, so the `green` and `red` cases (column 8 is each case's value) could never run. Annotate it, `const light: string = "yellow";`, as in `traffic-light.ts`, then fix or delete this file before your next check.

PAUSE AND TRY

Question 15 Tracing

Predict exactly what this program shows. Look for every `break`.

predict-switch.ts

```
1 const size: string = "m";
2 switch (size) {
3   case "s":
4     console.log("Small");
5     break;
6   case "m":
7   case "l":
8     console.log("Medium or large");
9   case "xl":
10    console.log("Extra large");
11    break;
12   default:
13     console.log("Unknown size");
14 }
```

Putting it together

A cinema charges \$6 for children under 12, \$7 for people 65 and over, and \$10 for everyone else. On Tuesdays every ticket is \$2 off, and on weekends \$1 more. A member code gives \$1 off, and teenagers from 13 to 17 get free popcorn. Before you type it, make sure `npx tsc` is clean in your `day04` folder: fix or delete any broken file from today.

cinema.ts

```
1  const age: number = 15;
2  const typedDay: string = " Tuesday ";
3  const memberCode: string = "";
4
5  const day = typedDay.trim().toLowerCase();
6  let price: number;
7  if (age < 12) {
8    price = 6;
9  } else if (age >= 65) {
10   price = 7;
11 } else {
12   price = 10;
13 }
14
15 switch (day) {
16   case "tuesday":
17     price = price - 2;
18     break;
19   case "saturday":
20   case "sunday":
21     price = price + 1;
22     break;
23 }
24
25 const discount = memberCode ? 1 : 0;
26 const isTeen = age >= 13 && age <= 17;
27 console.log(`Ticket: ${!(price - discount).toFixed(2)}`);
28 console.log(isTeen ? "Teen ticket: free popcorn!" : "Enjoy the film!");
29 console.log(`Member code: ${memberCode || "none"}`);
```

`npx tsc` reports no errors, and `node cinema.ts` shows:

```
Ticket: $8.00
Teen ticket: free popcorn!
Member code: none
```

Follow the values in the order the lines run:

Line	Worked out	Result	Idea
1-3	Annotated, because these stand for values that change each time	15, " Tuesday ", ""	4.3
5	" Tuesday ".trim().toLowerCase()	"tuesday"	Day 3
7	15 < 12	false	4.7
9	15 >= 65	false	4.7

12	The <code>else</code> block	<code>price</code> is 10	4.7
16–18	<code>"tuesday"</code> matches the first case; <code>price - 2</code> , then <code>break</code>	<code>price</code> is 8	4.10
25	<code>memberCode</code> is <code>""</code> , which is falsy	<code>discount</code> is 0	4.8, 4.9
26	<code>15 >= 13 && 15 <= 17</code>	<code>true</code>	4.4
27	<code>(8 - 0).toFixed(2)</code>	<code>"8.00"</code>	Day 3
28	<code>isTeen</code> is <code>true</code> , so the first choice	<code>"Teen ticket: ..."</code>	4.9
29	<code>""</code> is falsy, so <code> </code> gives <code>"none"</code>	<code>"none"</code>	4.8

Each tool does the job it fits best. The if chain sorts every customer into exactly one price band, with `else` catching everyone not in a narrow band, so the checker knows `price` is always assigned by line 15. The `switch` compares one value, `day`, with fixed choices, and stacks `saturday` and `sunday`; it has no `default`, because on other days the price does not change. The two conditional operators pick one of two values, and line 29's `||` gives a default for an empty code. Change the three values on lines 1 to 3, such as `70`, `" SUNDAY"`, and `"M123"`, and only the output changes.

Common mistakes

Today's messages and silent surprises in one place. The first group comes from `npm tsc`; the second shows no message at all.

Message or symptom	Usual cause	Fix
TS2367: This comparison appears to be unintentional because the types ...	Comparing a string with a number, or a constant with a value it never holds, or <code>x !== a x !== b</code>	Convert the value; annotate the constant; use <code>&&</code>
TS2678: Type <code>"green"</code> is not comparable to type <code>"yellow"</code> .	A <code>switch</code> on a constant with no annotation	<code>const light: string = ...</code>
TS2365: Operator <code><</code> cannot be applied to types ...	<code><</code> between a number and a string, or <code>0 <= x <= 100</code>	<code>Number(); 0 <= x && x <= 100</code>
TS2845: This condition will always return <code>'false'</code> .	<code>=== NaN</code>	<code>Number.isNaN(value)</code>
TS2588: Cannot assign to <code>'total'</code> because it is a constant.	<code>=</code> where <code>===</code> was meant	<code>===</code>
TS2304: Cannot find name <code>'advice'</code> .	Using a variable outside the block that declared it	Declare it before the <code>if</code>
TS2454: Variable <code>'shipping'</code> is used before being assigned.	An if chain with no <code>else</code> assigns the variable on only some paths	Add an <code>else</code>
<code>TypeError: Cannot read properties of undefined (from node)</code>	The risky part of <code>&&</code> came before the safety check	Put the check on the left
A block runs every time	<code>if (score = 100)</code> with a <code>let</code>	<code>===</code>

The first rung always wins	The widest condition is first in the chain	Narrowest condition first
Extra lines from a <code>switch</code>	A missing <code>break</code>	End each case with <code>break</code>
"10" < "9" is true; 0 replaced by a default	Comparing digits as text; <code> </code> treats 0 as "nothing"	Compare numbers; use <code> </code> only for real "nothing"

Vocabulary

Term	Definition	Example
comparison operator	<code>></code> , <code><</code> , <code>>=</code> , <code><=</code> , <code>===</code> , <code>!==</code> : compares two values and gives a boolean	<code>score >= 60</code>
strict equality	<code>===</code> : true only for the same type and the same value	<code>answer === 42</code>
loose equality	<code>==</code> : converts types before comparing; avoid it	<code>"5" == 5</code> is true
overlap	Two types overlap when some value belongs to both	<code>string</code> and <code>number</code> do not
logical operator	<code>&&</code> , <code> </code> , <code>!</code> : combine or flip conditions	<code>height >= 120 && age >= 8</code>
truth table	A table of an operator's result for every combination of inputs	<code>true && false</code> is false
short-circuit evaluation	Skipping the right side of <code>&&</code> or <code> </code> when the left side decides the answer	<code>text.length > 0 && text[0] === "A"</code>
selection	Choosing which code runs based on a condition	An if chain
condition	The expression that decides whether a block runs	<code>temperature > 30</code>
block	Statements inside curly braces that belong together	<code>{ console.log("Hot"); }</code>
scope	The part of a program where a name can be used	A <code>const</code> inside a block
if chain	<code>if</code> , any <code>else if</code> s, and an optional <code>else</code> ; the first true condition runs	Shipping bands
falsy / truthy	Counts as false / as true in a condition	<code>""</code> / <code>"0"</code>
conditional operator	<code>condition ? a : b</code> : gives a or b (also called the ternary operator)	<code>n === 1 ? "item" : "items"</code>
fall-through	Running on from one case into the next when there is no <code>break</code>	Stacked case <code>"saturday"</code> :

Summary

- **Comparison operators** give `true` or `false`, after the arithmetic is worked out. Use **strict equality**, `===` and `!==`; `==` converts types first and surprises people. Strings compare character by character, so `"10" < "9"`.
- TS2367 means the two sides have types with no **overlap**, so the comparison can never be true. Convert the value, or annotate a constant that stands for a value that could vary:
`const day: string = "Sunday";`
- `&&` needs both sides, `||` at least one, and `!` flips. `&&` comes before `||`, and **short-circuit evaluation** skips the right side when the left decides, so put safety checks first.
- An **if chain** runs the block of the first true condition and skips the rest, with `else` catching everything else. A variable declared inside a **block** exists only there.
- `false`, `0`, `""`, `null`, `undefined`, and `NaN` are **falsy**; every other value is **truthy**. `a || b` gives `a` if it is truthy, otherwise `b`. Check `NaN` with `Number.isNaN` and `null` with `=== null`.
- `condition ? a : b` picks one of two values, and `switch` compares one value with fixed cases, falling through until a `break`.

Practice

Do these in your `day04` folder. First make sure `npm tsc` prints nothing there: if any broken file from today is still in the folder (such as `loose-equals.ts`, `today.ts`, `weekday-wrong.ts`, `shipping-no-else.ts`, `light-literal.ts`, or the questions' `predict-overlap.ts` and `library-fine.ts`), fix it or delete it. Then type each program yourself, check it with `npm tsc`, run it with `node`, and only then open the answer. Change the starting values and run them again, including values right on a boundary, such as exactly 90.

Question 16 Short answer

Letter grade. Store a numeric score in a constant (it stands for any score, so annotate it). Show a letter grade: 90 and above is A, 80 up to 90 is B, 70 up to 80 is C, 60 up to 70 is D, and below 60 is F. If the score is below 0 or above 100, show a message instead of a grade.

Question 17 Short answer

Leap year. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not, except that years divisible by 400 are. Store a year in a constant and show whether it is a leap year. (A year is divisible by 4 when `year % 4 === 0`.)

Question 18 Short answer

Command responder. Store a command, as a person might type it, in a constant. Show a different reply for each of three commands, and for anything else show a help message that names the command it did not know and lists the ones it does. Capital letters and spaces at the ends should not matter.

Review questions

Question 19 Tracing

Predict exactly what this program shows. Track `fee` through both `if` statements.

parking.ts

```
1 const hours: number = 5;
2 const isWeekend: boolean = true;
3 const hasPass: boolean = false;
4 let fee: number;
5 if (hasPass) {
6   fee = 0;
7 } else if (hours <= 1) {
8   fee = 2;
9 } else if (hours <= 4) {
10  fee = hours * 1.5;
11 } else {
12  fee = 8;
13 }
14 if (isWeekend && !hasPass) {
15  fee = fee + 1;
16 }
17 console.log(`Fee: ${fee.toFixed(2)}`);
18 console.log(fee > 5 ? "Pay at the machine" : "Pay at the gate");
```

Question 20 Multiple choice

Which condition is `true` exactly when `n` is from 1 to 10, including 1 and 10, but is not 5?

- A. `1 <= n <= 10 && n !== 5`
- B. `n >= 1 && n <= 10 && n !== 5`
- C. `n >= 1 || n <= 10 && n !== 5`
- D. `n >= 1 && n <= 10 || n !== 5`

Question 21 Debugging

Mia's program should show `Weekend rate` on Saturdays and Sundays. `npx tsc` rejects it. Predict the message, explain it, and write the fix.

weekend-rate.ts

```
1 const day: string = "sunday";
2 if (day === "saturday" && day === "sunday") {
3   console.log("Weekend rate");
4 } else {
5   console.log("Weekday rate");
6 }
```

Question 22 Tracing

Predict exactly what this program shows. Remember what `&&` and `||` give back.

coupon.ts

```
1 const coupon: string = "";
2 const owner: string = "Ravi";
3 console.log(coupon || "no coupon");
4 console.log(owner && owner.toUpperCase());
5 console.log(`[${coupon && coupon.toUpperCase()}]`);
6 console.log(owner.length > 3 && coupon.length > 3);
```

Question 23 Short answer

Lena wants to rewrite the letter-grade chain from Practice question 16 as a `switch`, starting with `switch (score) { case 90: ...`. Explain why that cannot work, and describe the kind of decision a `switch` is good for.

Question 24 Tracing

Predict exactly what this program shows.

day-number.ts

```
1 const dayNumber: number = 9;
2 switch (dayNumber % 7) {
3   case 0:
4     console.log("Sunday");
5     break;
6   case 6:
7     console.log("Saturday");
8     break;
9   default:
10    console.log("Weekday");
11 }
12 console.log(dayNumber % 7 === 2 ? "Tuesday" : "Not Tuesday");
```

Go further

MDN's guide walks through `if...else` and `switch` with more examples, and its page on falsy values lists every falsy value, including the rare ones: MDN: Control flow and error handling (read the "Conditional statements" part) and MDN: Falsy.

Reflection

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

☐ I can compare numbers and strings with `>`, `<`, `>=`, and `<=`, and predict the result. Reread: 4.1

Comparison Operators

☐ I can explain why `===` is used instead of `==`. Reread: 4.2 Strict Equality

☐ I can read a TS2367 message and fix the comparison it points at. Reread: 4.3 Impossible

Comparisons

☐ I can combine conditions with `&&`, `||`, and `!`, using parentheses when I mix them. Reread: 4.4

Logical Operators: And, Or, Not

☐ I can use short-circuit evaluation to keep a risky check from running. Reread: 4.5 Short-Circuit

Evaluation

☐ I can write an `if` statement and say which lines run. Reread: 4.6 `if` Statements and Blocks

☐ I can turn a rule written in English into an `if` chain in the right order. Reread: 4.7 `else` and `else if`

☐ I can name the falsy values and predict what `||` gives back. Reread: 4.8 Truthy and Falsy Values

☐ I can pick one of two values with the conditional operator. Reread: 4.9 The Conditional Operator

☐ I can write a `switch` with `break` and `default`, and explain fall-through. Reread: 4.10 `switch` and `break`

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
true  
false  
true  
true  
false
```

- Line 3: 135 is at least 120: **true**.
- Line 4: arithmetic first, $135 - 20$ is 115, and 115 is not more than 120: **false**.
- Line 5: `minimum + 15` is 135, so this asks whether 135 equals 135: **true**.
- Line 6: A comes before B: **true**.
- Line 7: every capital letter comes before every small letter, so "ana" comes **after** "Ben", and `"ana" < "Ben"` is **false**.

Reread 4.1 *Comparison Operators* if any line differed.

Question 2

A. Both sides are strings, so they are compared character by character. The first characters are "1" and "2", and "1" comes first, so the comparison is `true`, even though 100 is more than 25 as a number.

- **B** is what you would get comparing the **numbers** 100 and 25. The quotes make them text.
- **C** is wrong: TS2365 is for a comparison between a number and a string. Two strings may be compared.
- **D** is wrong: a comparison always gives `true` or `false`, never a number.

Reread the Watch Out in 4.1 *Comparison Operators* if you chose differently.

Question 3

```
true  
false  
true  
true  
true
```

- Line 3: $2 + 2$ is 4: **true**.
- Line 4: they are equal, so "not equal" is **false**.
- Line 5: `"Tea".toLowerCase()` is "tea", the same characters: **true**.
- Line 6: word is exactly "Tea": **true**.
- Line 7: "Tea" has 3 characters, and `3 === 3`: **true**.

Reread 4.2 *Strict Equality* if any line differed.

Question 4

C. `(0.1 + 0.2).toFixed(1)` rounds 0.30000000000000004 to one place and gives the string "0.3", which is strictly equal to "0.3".

- **A** is wrong: the checker rejects comparing a string with a number (TS2367), even with `==`. Run anyway, it would show `true`, which is why `==` is avoided.
- **B** is wrong: it is accepted, but shows `false`, because the stored sum is not exactly 0.3.
- **D** is wrong: TS2367 again. A string can never be strictly equal to a number, and the checker says so.

Reread 4.2 *Strict Equality* if you chose differently.

Question 5

```
predict-overlap.ts:5:13 - error TS2367: This comparison appears to be unintentional
because the types 'number' and 'string' have no overlap.
predict-overlap.ts:6:13 - error TS2367: This comparison appears to be unintentional
because the types '"Lima"' and '"Paris"' have no overlap.
```

Line	Comparison	Types	Result
4	<code>city === "Paris"</code>	string (annotated), "Paris"	false
5	<code>visitors === "120"</code>	120, "120"	TS2367
6	<code>capital === "Paris"</code>	"Lima", "Paris"	TS2367
7	<code>String(visitors) === "120"</code>	string, "120"	true

Lines 4 and 6 hold the same value, but line 4's constant is annotated `string`, so it could be any string, while `capital` can only ever be `"Lima"`. Line 5 compares a number with a string, two different kinds of value, so its message names just the kinds, `'number'` and `'string'`, not the exact values `120` and `"120"`. Line 7 converts the number to text first, so it compares two strings. If you typed this program, fix or delete it before your next check.

Reread 4.3 *Impossible Comparisons* if you picked different lines.

Question 6

```
false
true
true
false
```

- Line 3: `22 > 20` is `true`, and `true && false` is **false**.
- Line 4: `true || false` is **true**.
- Line 5: `!` goes first: `!isSunny` is `true`. Then `22 < 25` is `true`, and `true && true` is **true**.
- Line 6: inside the parentheses, `true && true` is `true`, and `!true` is **false**.

Reread 4.4 *Logical Operators: And, Or, Not* if any line differed.

Question 7

C. A range is two comparisons that must **both** be true, joined with `&&`, and `>=` and `<=` include the ends.

- **A** is wrong: `18 <= temperature` gives a boolean, and comparing it with 25 is rejected with TS2365.
- **B** is wrong: every number is at least 18 **or** at most 25, so it is always true; 40 degrees would count as comfortable.
- **D** is wrong: `>` and `<` leave out exactly 18 and exactly 25.

Reread 4.4 *Logical Operators: And, Or, Not* if you chose differently.

Question 8

The program shows two lines, then stops:

```
false
true
TypeError: Cannot read properties of undefined (reading 'toUpperCase')
```

- Line 2: `code.length >= 2` is `false`, so `&&` stops: **false**.
- Line 3: `code.length < 2` is `true`, so `||` stops: **true**.
- Line 4: the risky part is on the left. `code[1]` is `undefined`, and calling `toUpperCase` on it is a runtime error. The program stops, so line 5 never runs and `Done` is never shown.

Lines 3 and 4 hold the same two parts in a different order, and only the order decides whether the program crashes.

Reread 4.5 *Short-Circuit Evaluation* if your answer differed.

Question 9

```
Purchase approved.
Your balance is low.
Balance: 40
```

- Line 3: `40 >= 25` is `true`, so the first block runs.
- Line 6: `40 - 25` is `15`, and `15 < 20` is `true`, so the second block runs too. Nothing was subtracted from `balance`; the subtraction happens only inside the condition.
- Line 9: `25 > 40` is `false`, so the third block is skipped.
- Line 12 runs whatever the conditions were, and `balance` is still `40`.

Reread 4.6 *if Statements and Blocks* if any line differed.

Question 10

```
Small fine
Over by 14
```

`over` is `64 - 50`, which is `14`.

Line	Condition	Worked out	Result
4	<code>over <= 0</code>	<code>14 <= 0</code>	<code>false</code>
6	<code>over <= 10</code>	<code>14 <= 10</code>	<code>false</code>
8	<code>over <= 20</code>	<code>14 <= 20</code>	<code>true: show Small fine</code>

The `else` block is skipped because an earlier condition was `true`. Line 13 is after the chain, so it always runs.

Reread 4.7 *else and else if* if any line differed.

Question 11

```
library-fine.ts:8:23 - error TS2454: Variable 'fine' is used before being assigned.
```

The chain has no `else`, so for a book that is not late (`daysLate` of 0), neither block runs and `fine` is never assigned. The checker follows that path to line 8 (column 23 is `fine` in the placeholder) and refuses to read a variable that might hold nothing. The English rule says "nothing otherwise," and "otherwise" is an `else`:

library-fine-fixed.ts

```
1 const daysLate: number = 3;
2 let fine: number;
3 if (daysLate > 7) {
4   fine = 5;
5 } else if (daysLate > 0) {
6   fine = daysLate * 0.5;
7 } else {
8   fine = 0;
9 }
10 console.log(`Fine: ${fine.toFixed(2)}`);
```

Fine: \$1.50

3 days is not more than 7 but is more than 0, so `fine` is $3 \times 0.5 = 1.5$. If you typed the original program, fix or delete it before you go on.

Reread the Watch Out in 4.7 *else and else if* if you missed it.

Question 12

```
Note added
No label
untitled
true
```

- Line 4: `items` is 0, which is falsy, so the block is skipped.
- Line 7: `note` is " ", one space. It is not empty, so it is truthy and the block runs.
- Line 10: `label` is "", falsy, so `!label` is `true` and the block runs.
- Line 13: "" is falsy, so `||` gives its right side, "untitled".
- Line 14: "0" is a string with one character, so it is truthy: **true**.

Reread 4.8 *Truthy and Falsy Values* if any line differed.

Question 13

C. "0" is a string with one character in it, and only the **empty** string is falsy. What the character is does not matter.

- A is wrong: the number zero is falsy.
- B is wrong: the empty string is falsy.
- D is wrong: `NaN` is falsy, even though its type is `number`.

Reread 4.8 *Truthy and Falsy Values* if you chose differently.

Question 14

```
child: $6
Freezing
Cannot vote
```

- Lines 2 and 3: `12 < 13` is true, so ticket is "child" and price is 6.
- Line 4: the \$ before the placeholder is an ordinary character, as on Day 3.
- Line 6: `-3 < 0` is true: **Freezing**.
- Line 7: `12 >= 18` is false, so the placeholder becomes "Cannot".

Reread 4.9 *The Conditional Operator* if any line differed.

Question 15

Medium or large
Extra large

size is "m", so the switch starts at line 6, case "m":. That case has no statements, so it falls through to line 8 and shows Medium or large. There is no break after it, so it falls through again into the xl case and shows Extra large. The break on line 11 finally leaves the switch. The writer probably forgot the break after line 8.

Reread 4.10 *switch and break* if your output differed.

Question 16

One possible solution:

grade.ts

```
1 const score: number = 89.5;
2 if (score < 0 || score > 100) {
3   console.log(`${score} is not a valid score.`);
4 } else if (score >= 90) {
5   console.log("Grade: A");
6 } else if (score >= 80) {
7   console.log("Grade: B");
8 } else if (score >= 70) {
9   console.log("Grade: C");
10 } else if (score >= 60) {
11   console.log("Grade: D");
12 } else {
13   console.log("Grade: F");
14 }
```

Grade: B

The first condition catches the invalid scores with `||`: a score is invalid if it is below 0 **or** above 100. The other rungs go from the highest band down, so each needs only a lower limit: by the time the program asks `score >= 80`, it already knows the score is below 90. 89.5 fails `>= 90` and passes `>= 80`, so it gets a B. The annotation `: number` lets you change the score freely; it stands for any number. Try 90, 60, 59.9, and 101 too.

Reread 4.7 *else and else if* and 4.4 *Logical Operators: And, Or, Not* if your result differed.

Question 17

One possible solution:

leap-year.ts

```
1 const year: number = 1900;
2 let isLeap: boolean;
3 if (year % 400 === 0) {
4   isLeap = true;
5 } else if (year % 100 === 0) {
6   isLeap = false;
7 } else if (year % 4 === 0) {
8   isLeap = true;
9 } else {
10  isLeap = false;
11 }
12 console.log(isLeap ? `${year} is a leap year.` : `${year} is not a leap year.`);
```

1900 is not a leap year.

The English rule has exceptions to exceptions, so the chain checks the **most specific** rule first. 1900 is not divisible by 400, but it is divisible by 100, so the second block sets `isLeap` to `false`. Checking `% 4` first would wrongly call 1900 a leap year, as in the grade Watch Out in 4.7. Every path assigns `isLeap`, so the checker accepts line 12, where a conditional operator picks one of two sentences.

The same rule also fits in one condition, using `&&`, `||`, and parentheses:

leap-rule.ts

```
1 const year: number = 2024;
2 const isLeap = (year % 4 === 0 && year % 100 !== 0) || year % 400 === 0;
3 console.log(`${year}: ${isLeap}`);
```

2024: true

Read it as "divisible by 4 and not by 100, or divisible by 400." For 2024, `2024 % 4 === 0` is `true` and `2024 % 100 !== 0` is `true`, so the left part is `true` and `||` stops there. Either version is correct; the if chain is easier to check against the English rule. Try 1900, 2000, and 2023 in both.

Reread 4.7 *else and else if* and 4.4 *Logical Operators: And, Or, Not* if your result differed.

Question 18

One possible solution, for a café:

commands.ts

```

1  const typedCommand: string = "  Hours ";
2  const command = typedCommand.trim().toLowerCase();
3  switch (command) {
4    case "hours":
5      console.log("Open 8:00 to 18:00, Monday to Saturday.");
6      break;
7    case "price":
8      console.log("A coffee costs $3.50.");
9      break;
10   case "wifi":
11     console.log("The network is CafeGuest.");
12     break;
13   default:
14     console.log(`Unknown command: ${command}`);
15     console.log("Try hours, price, or wifi.");
16 }

```

Open 8:00 to 18:00, Monday to Saturday.

`trim().toLowerCase()` cleans the typed text, as on Day 3, so " Hours " becomes "hours" and matches the first case. Each command is one fixed value, which is what `switch` is for, and `default` catches everything else; it has two statements, which is fine, because a case can run as many lines as it needs. An `if` chain with `command === "hours"` and so on works just as well. Change `typedCommand` to "PRICE", "wifi", and "dance", and run it again each time.

Reread 4.10 *switch and break* if your result differed.

Question 19

Fee: \$9.00
Pay at the machine

Line	Condition or step	Worked out	fee
5	hasPass	false	
7	hours <= 1	5 <= 1, false	
9	hours <= 4	5 <= 4, false	
12	the else block		8
14	isWeekend && !hasPass	true && true	
15	fee + 1		9
18	fee > 5 ? ... : ...	9 > 5, true	

The second `if` is a separate statement, not part of the chain, so its condition is asked whatever the chain did. `!hasPass` is `true` because `hasPass` is `false`.

Reread 4.7 *else and else if* and 4.9 *The Conditional Operator* if your output differed.

Question 20

B. All three parts must be true, so all three are joined with `&&`.

- **A** is wrong: `1 <= n` gives a boolean, and `boolean <= 10` is rejected with TS2365.

- **C** is wrong: `&&` goes before `||`, so this is `n >= 1 || (n <= 10 && n !== 5)`. For 5, `n >= 1` is already `true`, so `||` stops and gives `true`.
- **D** is wrong: this is `(n >= 1 && n <= 10) || n !== 5`. For 50, `n !== 5` is `true`, so the whole condition is `true`.

Reread 4.4 *Logical Operators: And, Or, Not* if you chose differently.

Question 21

```
weekend-rate.ts:2:27 - error TS2367: This comparison appears to be unintentional because the types '"saturday"' and '"sunday"' have no overlap.
```

Column 27 is the second comparison. The right side of `&&` runs only when the left side was `true`, that is, when `day` is `"saturday"`, and `"saturday"` can never equal `"sunday"`. So the condition would be `false` for every day: no day is both Saturday and Sunday. In English, "on Saturdays and Sundays" means "if it is Saturday **or** it is Sunday":

weekend-rate-fixed.ts

```
1 const day: string = "sunday";
2 if (day === "saturday" || day === "sunday") {
3   console.log("Weekend rate");
4 } else {
5   console.log("Weekday rate");
6 }
```

Weekend rate

If you typed the original program, fix or delete it before you go on.

Reread 4.5 *Short-Circuit Evaluation* and 4.4 *Logical Operators: And, Or, Not* if you missed it.

Question 22

```
no coupon
RAVI
[]
false
```

- Line 3: `""` is falsy, so `||` gives its right side: **no coupon**.
- Line 4: `"RAVI"` is truthy, so `&&` gives its right side, `"RAVI"`.
- Line 5: `""` is falsy, so `&&` gives the left side, `""`, and the right side never runs. The brackets show the empty string: `[]`.
- Line 6: both sides are booleans here: `4 > 3` is `true`, `0 > 3` is `false`, so **false**.

Reread 4.8 *Truthy and Falsy Values* and 4.5 *Short-Circuit Evaluation* if any line differed.

Question 23

Model answer: A `switch` compares its value with each `case` using `===`, so `case 90` matches only a score of exactly 90. A score of 89.5 or 95 equals no case, so it would fall to `default`. The grade rule is about **ranges** (90 and above, 80 up to 90), and a range needs a comparison such as `score >= 90`, which only an `if` condition can hold. A `switch` is good when **one** value is compared with a short list of **fixed** values, such as a command, a day, or a light color.

A complete answer must:

- say that `case` matches with `===`, one exact value at a time,
- explain that grade bands are ranges, which need `>=` comparisons in an if chain, and
- name fixed-value choices (commands, days, colors) as the job for `switch`.

Reread 4.10 *switch and break* and 4.7 *else and else if* if your reasoning differed.

Question 24

```
Weekday
Tuesday
```

`dayNumber % 7` is the remainder of $9 \div 7$, which is 2. The `switch` compares 2 with 0 and 6; neither matches, so the `default` statements run: **Weekday**. A `switch` can check a number just as well as a string, and its value can be any expression; it is worked out once. On the last line, `2 === 2` is true, so the conditional operator gives **Tuesday**.

Reread 4.10 *switch and break* and 4.9 *The Conditional Operator* if your output differed.