

# Day 2 Values, variables and types

Self-Study Guide · About 70 minutes

## By the end you will be able to

- Store values in variables with `const` and `let`, choose clear camelCase names, and trace how values change.
- Name the type of any value (number, string, boolean, null, undefined) and say whether the type checker will accept an assignment.
- Predict the value of any arithmetic expression, including remainder, powers, `Math.floor`, `Math.round`, and precedence.
- Convert between text and numbers with `Number()` and `String()`, and explain NaN and why `0.1 + 0.2` is not exactly `0.3`.

## What to remember from earlier

- `console.log` shows several values on one line, separated by spaces, and works out an expression before showing it. ([Day 1: Setup and your first program, 1.4 Printing Text and Numbers](#))
- Multiplication and division happen before addition and subtraction, and parentheses go first. ([Day 1: Setup and your first program, 1.4 Printing Text and Numbers](#))
- A program's statements run one at a time, from the top line to the bottom. ([Day 1: Setup and your first program, 1.5 Programs Run Top to Bottom](#))
- `npx tsc` checks types without running anything and prints nothing when all is well; `node` runs the file without checking types. ([Day 1: Setup and your first program, 1.7 Two Steps: Check, Then Run](#))
- TS2322 reads: Type 'what you gave' is not assignable to type 'what was required'. ([Day 1: Setup and your first program, 1.8 Reading a Type Error](#))
- A word outside quotes is a name, and the checker reports a name it does not know with TS2304, or TS2552 when it can suggest a close match. ([Day 1: Setup and your first program, 1.9 Names the Checker Does Not Know](#))
- A runtime error stops the program at that line, after earlier lines have shown their output. ([Day 1: Setup and your first program, 1.10 Reading a Runtime Error](#))

## 2.1 Variables: Giving a Value a Name

Before you start, set up a new project folder named `day02` 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.

On Day 1, every number you used was typed straight into an expression, like `30 - 12`. That works once. But imagine a program that uses a ticket price in five places: if the price changes, you must find and edit all five, and missing one gives a wrong answer. Programs need a way to work a value out once, remember it, and refer to it by name.

A **variable** is a name that holds a value. You create one with a **variable declaration**. The simplest kind starts with the word `let`:

```
let name = expression;
```

| Piece      | What it is                                                                                  |
|------------|---------------------------------------------------------------------------------------------|
| let        | "Make a new variable." It is a <b>keyword</b> : a word with a fixed meaning in the language |
| name       | The name you choose for the variable                                                        |
| =          | The <b>assignment operator</b> : store the value on the right in the name on the left       |
| expression | Any expression, worked out first                                                            |
| ;          | The end of the statement, as on Day 1                                                       |

The program runs a declaration in two steps. First it works out the expression on the **right**. Then it stores that value under the name on the **left**. From then on, wherever you write the name, the program uses the value stored there. Storing a value in a variable is called **assigning** it. The `=` sign does not mean "is equal to" as it does in math; read `let price = 4;` as "let price be 4" or "price gets 4".

Think of a variable as a labeled box: the name is the label, and the box holds one value at a time.

### WORKED EXAMPLE A ticket total

Watch what each name holds after each line.

ticket-total.ts

```
1 let ticketPrice = 12;
2 let ticketsSold = 35;
3 let total = ticketPrice * ticketsSold;
4 console.log("Tickets sold:", ticketsSold);
5 console.log("Total:", total);
```

Tickets sold: 35  
Total: 420

| Line | ticketPrice | ticketsSold | total |
|------|-------------|-------------|-------|
| 1    | 12          |             |       |
| 2    | 12          | 35          |       |
| 3    | 12          | 35          | 420   |
| 4    | 12          | 35          | 420   |
| 5    | 12          | 35          | 420   |

Lines 1 and 2 each store a number under a name. Line 3 looks up both names, works out `12 * 35`, and stores 420 under `total`. Lines 4 and 5 look up a name and show its value; they change nothing.

Line 3 never mentions `12` or `35`; it uses the names, and the program looks up their values. If the price changes, you edit line 1 and every line after it uses the new price.

Now vary one thing: assign a new value to a variable that already exists. You write the name, `=`, and the new value, **without** `let`, because the variable is already made. The old value is thrown

away and the new one takes its place. This lets you **update** a variable, even using its own old value on the right side.

score-update.ts

```
1 let score = 10;
2 console.log("Start:", score);
3 score = score + 5;
4 console.log("After bonus:", score);
5 score = 0;
6 console.log("After reset:", score);
```

```
Start: 10
After bonus: 15
After reset: 0
```

| Line | Right side worked out | score after the line |
|------|-----------------------|----------------------|
| 1    | 10                    | 10                   |
| 3    | score + 5 is 10 + 5   | 15                   |
| 5    | 0                     | 0                    |

Line 3, `score = score + 5;`, would be nonsense as a math equation, but as an assignment it makes sense: the right side is worked out first, using the **current** value 10, and only then is the result, 15, stored back in `score`. The `let` appears only once, on line 1, where the variable is made.

#### KEY IDEA

`let name = expression;` makes a variable: the right side is worked out first, then stored under the name. Later, `name = expression;` (no `let`) replaces the old value with a new one.

#### WATCH OUT

**Using a variable before the line that makes it.** A program runs top to bottom, so a variable exists only from its declaration onward:

use-before-declare.ts

```
1 console.log("Total:", total);
2 let total = 5 * 3;
```

```
use-before-declare.ts:1:23 - error TS2448: Block-scoped variable 'total' used before its declaration.
use-before-declare.ts:1:23 - error TS2454: Variable 'total' is used before being assigned.
```

The type checker reports two messages about the same spot, line 1, column 23, where `total` is used. Both say the same thing in different words: line 1 uses `total`, but the declaration is on line 2. ("Block-scoped" describes variables made with `let` and `const`; you can ignore the word for now.) Move the declaration above the `console.log`. If you typed this file, fix or delete it now.

**PAUSE AND TRY****Question 1** Tracing

What does this program show? Make a table of `apples` and `pears` after each line.

fruit-trace.ts

```
1 let apples = 6;
2 apples = apples + 4;
3 let pears = apples - 3;
4 apples = 2;
5 console.log(apples, pears);
```

**Question 2** Debugging

This program should show the old price, then change it to 5 and show the new price. `npx tsc` reports the messages below. Explain the mistake and write the fix.

price-change.ts

```
1 let price = 4;
2 console.log("Old price:", price);
3 let price = 5;
4 console.log("New price:", price);
```

```
price-change.ts:1:5 - error TS2451: Cannot redeclare block-scoped variable 'price'.
price-change.ts:3:5 - error TS2451: Cannot redeclare block-scoped variable 'price'.
```

## 2.2 Constants: Names That Never Change

Look back at `ticket-total.ts` in 2.1. None of its three variables ever changed after it was made. Most values in a program are like that: a price, a name, a total worked out once. If such a value changed by accident, the program would quietly give wrong answers.

TypeScript has a second kind of declaration for values that must stay the same. It starts with `const` (short for "constant") instead of `let`:

```
const name = expression;
```

A variable declared with `const` is a **constant**: it gets its value once, when it is made, and can never be assigned a new one. A variable declared with `let` can be assigned again as often as you like. You met `const` once already, in Day 1's preview `const muffinsLeft: number = 12; (1.8)`.

| Declaration        | Can it be assigned again? | Use it for                       |
|--------------------|---------------------------|----------------------------------|
| <code>const</code> | No                        | Values that never change (most)  |
| <code>let</code>   | Yes                       | Values that change, like a score |

### WORKED EXAMPLE Constants and one variable that changes

Here is the concert program again, with one value that does change: the number of seats left.

concert-total.ts

```
1 const ticketPrice = 12;
2 const ticketsSold = 35;
3 const total = ticketPrice * ticketsSold;
4 let seatsLeft = 40;
5 seatsLeft = seatsLeft - ticketsSold;
6 console.log("Total:", total);
7 console.log("Seats left:", seatsLeft);
```

```
Total: 420
Seats left: 5
```

The price, the number sold, and the total never change, so they are `const`. `seatsLeft` starts at 40 and is updated on line 5 to 40 - 35, which is 5, so it must be `let`.

Now vary it: try to assign a new value to a constant.

const-reassign.ts

```
1 const busSeats = 40;
2 console.log("Seats:", busSeats);
3 busSeats = 45;
```

```
const-reassign.ts:3:1 - error TS2588: Cannot assign to 'busSeats' because it is a constant.
```

The checker points at line 3, column 1, where the assignment starts, and the message says exactly why: `busSeats` was made with `const`. Line 2 is fine; reading a constant is always allowed. Only assigning to it is not. If the number of seats really does change, declare it with `let`; if it should not change, delete line 3. Fix or delete the file before you go on.

Why is `const` the default choice? Because it is a promise the type checker enforces. A reader who sees `const` knows the value is the same on every line after it, without searching the program for changes. And if you assign to it by mistake, you find out from `npx tsc` before the program ever runs. Good practice is to write `const` first and change it to `let` only when the value really needs to change. (Older JavaScript code uses a third keyword, `var`, which you may see online. This course never uses it.)

**KEY IDEA**

`const` makes a variable that can never be assigned again; `let` makes one that can. Use `const` unless the value must change, so the type checker catches accidental changes.

**WATCH OUT**

**Using `let` for everything.** A program full of `let` runs fine, and no error appears. But you lose the protection: if you change a value by accident, nothing warns you. Ask of each variable, "does this ever change?" Only the ones that do get `let`.

**PAUSE AND TRY****Question 3** Multiple choice

Which line does `npx tsc` reject?

lunch-sales.ts

```
1 const lunchPrice = 6;
2 let lunchesSold = 0;
3 lunchesSold = lunchesSold + 3;
4 const total = lunchPrice * lunchesSold;
5 lunchPrice = 7;
```

- A. Line 3, because it uses `lunchesSold` on both sides of `=`
- B. Line 4, because it uses two variables at once
- C. Line 5, because `lunchPrice` is a constant
- D. None; the program type-checks

## 2.3 Naming Rules and camelCase

You choose every variable name, so two questions follow: which names does TypeScript accept, and which names make a program easy to read?

A name you choose is called an **identifier**. The rules are short:

1. An identifier may contain letters, digits, the underscore `_`, and the dollar sign `$`, and nothing else: no spaces, hyphens, or other symbols.
2. It cannot start with a digit.
3. Capital and small letters count as different (names are **case-sensitive**, as Day 1 showed), so `score` and `Score` are two different names.
4. It cannot be a **reserved word**, one of the keywords the language keeps for its own grammar, such as `let`, `const`, `class`, `true`, `false`, and `null`.

Why these rules? The type checker has to tell names apart from everything else on a line. A name starting with a digit would look like a number, a space would make one name look like two, and a hyphen looks like a minus sign.

On top of the rules there is a **convention**, a habit every TypeScript programmer follows even though the checker does not enforce it. Variable names use **camelCase**: the first word all

lowercase, and every later word starting with a capital letter, with no spaces or underscores, like `ticketPrice` or `maxSpeedKmh`. The capitals in the middle stick up like the humps of a camel. Choose names that say what the value means: `ticketsSold` tells a reader far more than `t` or `x`.

### WORKED EXAMPLE Names that work

good-names.ts

```
1 const playerName = "Ana";
2 const player2Score = 18;
3 const maxSpeedKmh = 120;
4 console.log(playerName, player2Score, maxSpeedKmh);
```

Ana 18 120

All three names follow the rules and the convention. `player2Score` contains a digit, which is fine because it does not **start** with one. `maxSpeedKmh` adds the unit, a good habit that prevents mix-ups between kilometres and miles.

Now vary the names so they break the rules. Each of these is a syntax error, so `npm tsc` reports it before anything runs:

name-with-space.ts

```
1 let total cost = 5;
```

name-with-space.ts:1:11 - error TS1005: ',' expected.

name-starts-with-digit.ts

```
1 let 2ndPlace = "Lee";
```

name-starts-with-digit.ts:1:1 - error TS1440: Variable declaration not allowed at this location.  
name-starts-with-digit.ts:1:7 - error TS1005: ';' expected.  
name-starts-with-digit.ts:1:7 - error TS1351: An identifier or keyword cannot immediately follow a numeric literal.

name-is-keyword.ts

```
1 const class = "7B";
```

name-is-keyword.ts:1:7 - error TS1389: 'class' is not allowed as a variable declaration name.  
name-is-keyword.ts:1:13 - error TS1005: '{' expected.

In the first, the checker reads `total` as the whole name and then finds a second word, `cost`, where it expected the declaration to go on (column 11 is where `cost` starts). In the second, it reads `2` as the start of a number and then finds letters glued to it; a **numeric literal** is a number typed directly into the code, and no number can contain letters. In the third, `class` is a reserved word, and the first message says so plainly. As on Day 1, read the **first** message of each group; the others follow from it. The fixes are `totalCost`, `secondPlace`, and a name like `className`. If you typed these files, delete them now.

| Name       | Allowed? | Why                                           |
|------------|----------|-----------------------------------------------|
| totalCost  | Yes      | Letters only; good camelCase                  |
| player2    | Yes      | A digit is fine after the first character     |
| 2ndPlace   | No       | Starts with a digit                           |
| total cost | No       | Contains a space                              |
| total-cost | No       | The hyphen is read as minus                   |
| class      | No       | A reserved word                               |
| total_cost | Yes      | Allowed, but not the convention for variables |

**KEY IDEA**

An identifier uses letters, digits, `_`, and `$`, does not start with a digit, and is not a reserved word. By convention, variable names are descriptive camelCase, like `ticketsSold`.

**WATCH OUT**

**Two names that differ only in capitals.** Because names are case-sensitive, `Score` is a completely different name from `score`. Typing it by mistake is caught, because TypeScript only lets you assign to a variable that has been declared:

wrong-capitals.ts

```
1 let score = 10;
2 Score = score + 5;
3 console.log(score);
```

```
wrong-capitals.ts:2:1 - error TS2552: Cannot find name 'Score'. Did you mean 'score'?
```

This is the Day 1 message for an unknown name, with a suggestion because `Score` is close to `score`. Sticking to camelCase (always a small first letter) makes this mistake rare. Fix the capital before you go on.

**PAUSE AND TRY****Question 4** Multiple choice

Which is a valid variable name that also follows the camelCase convention?

- A. `final score`
- B. `finalScore`
- C. `final_score`
- D. `2finalScore`

## 2.4 Numbers, Strings, and Booleans

On Day 1 you learned that every value has a **type**, the kind of value it is, and you met two: `number` and `string` (1.7). The type decides what you can do with a value: you can multiply numbers, but multiplying two pieces of text makes no sense. Today you meet three everyday types, and in 2.7 two more.

- `number` is the type of every number: whole numbers like `42`, `0`, and `-7`, and decimals like `3.5` and `0.45`. Many languages have separate types for whole numbers and decimals; JavaScript has just one. That is why Day 1's `8 / 2` showed `4` and `7 / 2` showed `3.5`: both are simply numbers.
- `string` is the type of text in quotes, like `"Hatchet"`.
- `boolean` is the type of a value that is either `true` or `false`, and nothing else. Those two words are written in small letters and without quotes. A boolean answers a yes-or-no question: is the shop open? Is the book a hardcover? On Day 4 you will make booleans by comparing values; today you store and show them.

To find out a value's type while the program runs, put the keyword `typeof` in front of it. `typeof` gives back the type's name as a string: `typeof 42` is `"number"`. This is JavaScript speaking: `typeof` runs inside `node` and reports what the value is at that moment.

### WORKED EXAMPLE The types in a book record

basic-types.ts

```
1 const pages = 320;
2 const weightKg = 0.45;
3 const title = "Hatchet";
4 const isHardcover = false;
5 console.log(pages, typeof pages);
6 console.log(weightKg, typeof weightKg);
7 console.log(title, typeof title);
8 console.log(isHardcover, typeof isHardcover);
```

```
320 number
0.45 number
Hatchet string
false boolean
```

| Name        | Value     | Type    | Meaning          |
|-------------|-----------|---------|------------------|
| pages       | 320       | number  | A count          |
| weightKg    | 0.45      | number  | A measurement    |
| title       | "Hatchet" | string  | Text             |
| isHardcover | false     | boolean | A yes-or-no fact |

`typeof` works on a variable too: it reports the type of the value stored there. A boolean's name often starts with `is` or `has`, so that it reads like a yes-or-no question.

Now vary the quotes. `console.log` shows a string without its quotes, so a number and a string of digits look the same in the terminal. `typeof` tells them apart:

## typeof-quotes.ts

```
1 console.log(42, typeof 42);
2 console.log("42", typeof "42");
3 console.log(true, typeof true);
4 console.log("true", typeof "true");
5 console.log(typeof (6 * 7));
```

```
42 number
42 string
true boolean
true string
number
```

Lines 1 and 2 both show `42`, and lines 3 and 4 both show `true`, but the types differ: the quotes make `"42"` and `"true"` strings, pieces of text that happen to contain digits or a word. The last line shows that `typeof` works on an expression too: `6 * 7` is worked out to `42`, a number. The parentheses matter there: without them, `typeof` would apply only to the `6`.

## KEY IDEA

`number` covers whole numbers and decimals alike, `string` is text in quotes, and `boolean` is `true` or `false` (small letters, no quotes). `typeof value` gives the type's name while the program runs.

## WATCH OUT

**Writing `True` with a capital.** Some other languages spell the boolean values with a capital letter, and students carry the habit over:

## true-capital.ts

```
1 const isMember = True;
2 console.log("Member:", isMember);
```

```
true-capital.ts:1:18 - error TS2304: Cannot find name 'True'.
```

With a capital, `True` is not the boolean value; it is just an unknown name, so you get Day 1's TS2304. Write `true`. And do not put quotes around it: `"true"` is a string, as `typeof-quotes.ts` showed. Fix or delete the file before you go on.

## PAUSE AND TRY

## Question 5 Multiple choice

Which value has the type `boolean`?

- A. `"false"`
- B. `false`
- C. `False`
- D. `0`

## 2.5 Type Annotations and Inference

`typeof` tells you a type while the program runs. But the type checker works **before** anything runs, so it needs to know each variable's type from the code alone. There are two ways it finds out.

The first is that you tell it. A **type annotation** is a colon and a type name written after a variable's name: `let visitors: number = 0;` reads "visitors is a variable of type `number`, starting at 0." You saw one on Day 1 in `const muffinsLeft: number = 12;`.

```
let name: type = expression;
```

| Piece                     | What it is                                              |
|---------------------------|---------------------------------------------------------|
| <code>: type</code>       | The annotation: the type this variable must always hold |
| <code>= expression</code> | The starting value, which must have that type           |

The type names in annotations are the ones `typeof` reports: `number`, `string`, `boolean`. An annotation is TypeScript only; when Node runs the file, it strips the annotation away, as Day 1 explained.

### WORKED EXAMPLE Annotations on every variable

annotated.ts

```
1 let visitors: number = 0;
2 const city: string = "Lima";
3 const isOpen: boolean = true;
4 visitors = visitors + 25;
5 console.log(city, isOpen, visitors);
```

```
Lima true 25
```

Each variable is labeled with its type, and each starting value has that type, so `npm tsc` reports no errors for this file. Line 4 adds 25 to `visitors`; a number plus a number is a number, so the checker accepts storing it back.

Now vary it: take every annotation away.

inferred.ts

```
1 let visitors = 0;
2 const city = "Lima";
3 const isOpen = true;
4 visitors = visitors + 25;
5 console.log(city, isOpen, visitors);
```

```
Lima true 25
```

The output is the same, and `npm tsc` still reports no errors for it. The checker did not need the annotations, because it can see the starting values: 0 is a number, so `visitors` must be a number. Working out a type from the code without being told is called **type inference**. In VS Code, hold the mouse pointer over `visitors` and a small box shows `let visitors: number`: the type the checker

inferred. (Over a `const`, the box shows the value itself, such as `const city: "Lima"`, because a constant can never hold anything else; Day 9 explains that kind of type.)

So when should you write an annotation? When the checker cannot see a starting value to infer from (2.7 shows that case), or when you want to state your intention so the checker can hold you to it. When a variable gets an obvious starting value, most programmers let inference do the work, and this course does the same from here on.

#### KEY IDEA

A type annotation (`let visitors: number = 0;`) tells the checker a variable's type. Without one, the checker infers the type from the starting value. Either way, the variable has a type before the program runs.

#### WATCH OUT

**An annotation that does not match the value.** An annotation is a rule, and the starting value must follow it:

annotation-mismatch.ts

```
1 const isOpen: boolean = "true";
2 console.log("Open:", isOpen);
```

```
annotation-mismatch.ts:1:7 - error TS2322: Type 'string' is not assignable to type 'boolean'.
```

Read it as Day 1 taught: Type *what you gave* is not assignable to type *what was required*. "true" is in quotes, so it is a string, but the annotation requires a boolean. Remove the quotes. Fix or delete the file before you go on.

#### PAUSE AND TRY

##### Question 6 Short answer

For each variable, name the type the checker gives it, and say whether it came from an annotation or from inference. Which annotation could you remove without changing anything?

inferred-types.ts

```
1 let lives = 3;
2 const nickname = "Ace";
3 let isPaused = false;
4 let speed: number = 2.5;
5 console.log(lives, nickname, isPaused, speed);
```

## 2.6 Static Typing: A Variable Keeps Its Type

In 2.1 you assigned a new number to `score`. Could you assign a string to it instead? In plain JavaScript, yes: a JavaScript variable holds whatever you put in it, and a number can be replaced by text without complaint. That freedom causes bugs, because later lines still expect the number. TypeScript closes that door.

In TypeScript, a variable's type is fixed when it is declared, by its annotation or by inference, and it never changes. Every later assignment must give a value of that type. This is called **static typing**: "static" means fixed, settled before the program runs. The type checker reads every assignment and rejects any that does not fit.

### WORKED EXAMPLE A count that must stay a number

count-five.ts

```
1 let count: number = 0;
2 count = "five";
3 console.log(count);
```

```
count-five.ts:2:1 - error TS2322: Type 'string' is not assignable to type 'number'.
```

Line 1 declares `count` with the type `number`. Line 2 tries to store the string `"five"` in it. The checker points at line 2, column 1, where the assignment starts: a string is not allowed where a number is required. `let` lets you change the **value**, but never the **type**.

Now vary it: leave out the annotation, and add a line that stores a different number.

count-five-inferred.ts

```
1 let count = 0;
2 count = 5;
3 count = "five";
4 console.log(count);
```

```
count-five-inferred.ts:3:1 - error TS2322: Type 'string' is not assignable to type 'number'.
```

The same message, now for line 3. The checker inferred `number` from the starting value `0`, and an inferred type is just as fixed as an annotated one. Line 2 is accepted: `5` is a different value, but the same type. Only line 3 breaks the rule.

| Line                         | Value's type        | Variable's type                | Accepted?                     |
|------------------------------|---------------------|--------------------------------|-------------------------------|
| <code>let count = 0;</code>  | <code>number</code> | <code>number</code> (inferred) | Yes, it is declared here      |
| <code>count = 5;</code>      | <code>number</code> | <code>number</code>            | Yes                           |
| <code>count = 2.5;</code>    | <code>number</code> | <code>number</code>            | Yes, decimals are numbers too |
| <code>count = "five";</code> | <code>string</code> | <code>number</code>            | No, TS2322                    |

If you typed these two files, fix or delete them now. The fix depends on what you meant: if the count is a number, store `5`; if you need the word, make a separate `const countText = "five";`.

**KEY IDEA**

A variable's type is set when it is declared and never changes. The type checker rejects any assignment of a different type, such as `count = "five"` for a `number`, before the program runs.

**WATCH OUT**

**Changing the annotation to make the error go away.** When TS2322 appears, it is tempting to edit the type until the message stops. But the message is usually right: the variable was meant to hold one kind of value, and some line is trying to put another kind in. Ask which line is wrong, the declaration or the assignment, and fix that one.

**PAUSE AND TRY****Question 7** Tracing

Which line does `npx tsc` reject? Write the full message it prints.

thermostat.ts

```
1 let temperature = 21;
2 temperature = 22.5;
3 temperature = temperature - 4;
4 temperature = "cold";
5 console.log(temperature);
```

## 2.7 Null and Undefined

Some values are not known yet. A player with no middle name has none to store; a game that has not been scheduled has no date. JavaScript has two values that mean "nothing here," and TypeScript gives each one a type.

- `null` means "deliberately empty": the program knows there is no value. It is written in small letters, without quotes.
- `undefined` means "no value has been given yet." JavaScript gives a variable this value when nothing has been assigned to it.

Each is the only value of its own type: the value `null` has the type `null`, and the value `undefined` has the type `undefined`.

**WORKED EXAMPLE** Two kinds of nothing

empty-values.ts

```
1 const middleName = null;
2 const nextGame = undefined;
3 console.log(middleName, nextGame);
4 console.log(typeof middleName, typeof nextGame);
```

```
null undefined
object undefined
```

Line 3 shows both values as words. Line 4 holds a surprise: `typeof null` is "object", not "null". That is a mistake from the very first version of JavaScript, and it can never be fixed, because changing it would break old websites that depend on it. Remember it as a quirk: `typeof` cannot tell you that a value is `null`. (Day 4 shows how to check for `null` directly.)

Now vary it: declare a variable with an annotation but no starting value. That is allowed with `let`, and it is the case 2.5 promised, where an annotation is needed because there is no value to infer from. Until you assign something, the variable holds `undefined`, and strict mode will not let you read it:

unassigned.ts

```
1 let finalScore: number;
2 console.log("Final score:", finalScore);
```

```
unassigned.ts:2:29 - error TS2454: Variable 'finalScore' is used before being assigned.
```

The checker points at line 2, column 29, where `finalScore` is read. Nothing has been stored in it, so showing it would only show `undefined`, which is almost certainly a bug. Assign a value first and the checker is satisfied:

assign-later.ts

```
1 let finalScore: number;
2 finalScore = 88;
3 console.log("Final score:", finalScore);
```

```
Final score: 88
```

Strict mode, the `"strict": true` setting from Day 1, also keeps `null` and `undefined` out of other types. A `number` variable must hold a number, and "nothing" is not a number:

null-count.ts

```
1 let booksOut: number = null;
2 console.log("Books out:", booksOut);
```

```
null-count.ts:1:5 - error TS2322: Type 'null' is not assignable to type 'number'.
```

This rule protects you: whenever a variable's type is `number`, you can be sure it holds an actual number, so arithmetic on it will not go wrong. A variable that is sometimes a number and

sometimes `null` needs a type that says so; that is a preview of Day 9, which teaches how to write one. Fix or delete `unassigned.ts` and `null-count.ts` before you go on.

#### KEY IDEA

`null` means deliberately empty and `undefined` means not assigned yet; each is the only value of its own type. Strict mode will not let you read a variable before it is assigned, or store `null` or `undefined` in a `number` or `string`.

#### WATCH OUT

**Trusting `typeof` for `null`.** `typeof null` is `"object"`, so a program that uses `typeof` to look for `null` never finds it. This is one of the few places where `typeof` misleads you; for every other value today, it reports the type you expect.

#### PAUSE AND TRY

##### Question 8 Multiple choice

Which line does `npx tsc` accept?

- A. `let winner: number = null;`
- B. `const nickname = null;`
- C. `let goals: number; followed by console.log(goals);`
- D. `const total: number = undefined;`

## 2.8 Remainders, Powers, and Rounding

A movie is 135 minutes long. How many hours and minutes is that? `135 / 60` gives 2.25, which is true but not what you want. You want "2 hours and 15 minutes": a whole number of hours, and the minutes left over. Day 1 gave you `+`, `-`, `*`, and `/`. Today adds two more operators and two rounding tools.

- The **remainder** operator `%` gives what is left over after dividing as many whole times as possible. `135 % 60` is 15, because two 60s make 120, and `135 - 120` is 15. The `%` sign here has nothing to do with percentages.
- The **exponent** operator `**` raises a number to a power. `2 ** 10` means 2 multiplied by itself 10 times, which is 1024.
- `Math.floor(x)` rounds `x` **down** to a whole number. `Math.floor(2.25)` is 2. ("Floor" means the whole number at or below, like the floor below you.)
- `Math.round(x)` rounds `x` to the **nearest** whole number. `Math.round(2.25)` is 2, and `Math.round(2.75)` is 3.

`Math` is built in, like `console`, and holds many math tools. Read `Math.floor` as Day 1 taught you to read `console.log`: "Math's `floor` function." You call it with the number in parentheses, and it gives back the rounded number, which you can store, show, or use in another expression.

**WORKED EXAMPLE** Minutes to hours and minutes

minutes-to-hours.ts

```

1 const totalMinutes = 135;
2 const hours = Math.floor(totalMinutes / 60);
3 const minutes = totalMinutes % 60;
4 console.log(totalMinutes, "minutes is", hours, "hours and", minutes, "minutes");

```

135 minutes is 2 hours and 15 minutes

| Line | Expression                                 | Worked out                     | Value |
|------|--------------------------------------------|--------------------------------|-------|
| 2    | <code>Math.floor(totalMinutes / 60)</code> | 135 / 60 is 2.25, rounded down | 2     |
| 3    | <code>totalMinutes % 60</code>             | 135 - 2 × 60 = 135 - 120       | 15    |

Line 4 shows six arguments, separated by spaces. Check the answer: 2 whole hours of 60 minutes plus 15 left over is  $2 \times 60 + 15 = 135$ , the number you started with. That check always works for positive numbers, because `Math.floor(a / b)` counts the whole groups and `a % b` counts what is left. Change `totalMinutes` to 200 and run it again: you get 3 hours and 20 minutes.

Now vary the numbers, to see what the rules really depend on:

operator-edges.ts

```

1 console.log(8 % 2, 7 % 2);
2 console.log(-7 % 2);
3 console.log(2 ** 10, 9 ** 0.5);
4 console.log(Math.floor(3.99), Math.floor(-3.5));
5 console.log(Math.round(2.4), Math.round(2.5), Math.round(-2.5));

```

```

0 1
-1
1024 3
3 -4
2 3 -2

```

- `% 2` gives 0 for every even number and 1 for every odd one, so the remainder tells you whether a number is even.
- `-7 % 2` is `-1`: the remainder takes the sign of the number on the **left**.
- `9 ** 0.5` raises 9 to the power one half, which is the square root of 9: 3.
- `Math.floor(3.99)` is 3, not 4: it always rounds down, however close the next number is. And `Math.floor(-3.5)` is `-4`, because "down" means toward the lower number, and for negatives that is `-4`, not `-3`.
- `Math.round(2.4)` is 2, and a number exactly halfway rounds **up** to the larger number: `Math.round(2.5)` is 3, and `Math.round(-2.5)` is `-2` (because `-2` is larger than `-3`).

**KEY IDEA**

`%` gives the remainder and `**` raises to a power. `Math.floor` rounds down and `Math.round` rounds to the nearest whole number, halves going up. `Math.floor(a / b)` whole groups and `a % b` left over rebuild `a`.

**WATCH OUT**

**Expecting an error when dividing by zero.** Sharing 12 pizza slices among 0 people has no sensible answer, but neither the type checker nor Node reports an error:

share-pizza.ts

```
1 const pizzaSlices = 12;
2 const people = 0;
3 console.log("Each person gets", pizzaSlices / people);
4 console.log("Left over:", pizzaSlices % people);
```

```
Each person gets Infinity
Left over: NaN
```

The types fit (numbers divided by numbers), so `npx tsc` reports no errors for this file. And JavaScript does not stop the program: `/` by zero gives the special number `Infinity`, and `%` by zero gives `NaN`, a value you will meet in 2.10. Both are numbers, so the program runs on with a nonsense result. When a result shows `Infinity` or `NaN`, look for a division by zero.

**PAUSE AND TRY****Question 9** Tracing

Predict exactly what this program shows.

predict-operators.ts

```
1 console.log(17 % 5);
2 console.log(Math.floor(17 / 5));
3 console.log(17 / 5);
4 console.log(2 ** 5);
5 console.log(Math.round(17 / 5));
```

## 2.9 Operator Precedence

With six arithmetic operators, an expression like `10 + 3 ** 2 * 2 - 8 % 3` needs a clear rule for what happens first. Day 1 gave part of it: multiplication and division before addition and subtraction, parentheses first. Here is the full rule for today's operators.

**Operator precedence** is the order in which the program applies operators. From first to last:

| Order | Operators | Notes                               |
|-------|-----------|-------------------------------------|
| 1     | ( )       | Parentheses, innermost first        |
| 2     | **        | Power; groups from the <b>right</b> |
| 3     | * / %     | Left to right                       |
| 4     | + -       | Left to right                       |

Operators on the same row are applied left to right, in the order they appear, with one exception: a chain of `**` is worked out from the right. A function call like `Math.floor(...)` works out everything inside its parentheses first, just like a pair of parentheses.

### WORKED EXAMPLE Evaluating step by step

Work out `10 + 3 ** 2 * 2 - 8 % 3` the way the program does, one operator at a time:

| Step | Rule used                                                            | Expression now                       |
|------|----------------------------------------------------------------------|--------------------------------------|
| 0    | Start                                                                | <code>10 + 3 ** 2 * 2 - 8 % 3</code> |
| 1    | <code>**</code> first: <code>3 ** 2</code> is 9                      | <code>10 + 9 * 2 - 8 % 3</code>      |
| 2    | <code>*</code> next, leftmost: <code>9 * 2</code> is 18              | <code>10 + 18 - 8 % 3</code>         |
| 3    | Then <code>%</code> : <code>8 % 3</code> is 2                        | <code>10 + 18 - 2</code>             |
| 4    | <code>+</code> and <code>-</code> left to right: <code>28 - 2</code> | <code>26</code>                      |

precedence.ts

```
1 console.log(10 + 3 ** 2 * 2 - 8 % 3);
2 console.log(100 / 10 / 2);
3 console.log(2 ** 3 ** 2);
4 console.log((2 ** 3) ** 2);
```

```
26
5
512
64
```

Line 1 agrees: 26. The next lines show the same-row rules. `100 / 10 / 2` goes left to right: `100 / 10` is 10, then `10 / 2` is 5. `2 ** 3 ** 2` goes right to left: `3 ** 2` is 9 first, then `2 ** 9` is 512.

Parentheses change it: `(2 ** 3) ** 2` is `8 ** 2`, which is 64.

Now vary one thing in a real calculation: the parentheses. To average two test scores, you add them and divide by 2.

## average-scores.ts

```
1 const scoreA = 88;
2 const scoreB = 94;
3 console.log(scoreA + scoreB / 2);
4 console.log((scoreA + scoreB) / 2);
```

```
135
91
```

Without parentheses, `/` happens before `+`, so line 3 divides only 94 by 2 (47) and adds 88, giving 135, which is not an average. With parentheses, the addition happens first, and the average is  $182 / 2 = 91$ . Neither the checker nor Node can spot the mistake on line 3: both lines are correct code that works out a number. Only knowing what the answer should be reveals it.

## KEY IDEA

The program applies parentheses, then `**` (right to left), then `*` / `%`, then `+` - , left to right within a row. When you are not sure, add parentheses: they make the order certain and easier to read.

## WATCH OUT

**Squaring a negative number.** Programming languages and calculators disagree about whether  $-3^2$  means  $(-3)^2$  or  $-(3^2)$ , so JavaScript refuses to guess, and the checker rejects the expression:

## negative-power.ts

```
1 console.log(-3 ** 2);
```

```
negative-power.ts:1:13 - error TS17006: An unary expression with the '-' operator is not allowed in the left-hand side of an exponentiation expression. Consider enclosing the expression in parentheses.
```

A **unary** operator is one that works on a single value, like the minus sign in `-3`. The message asks for parentheses, so write the one you mean:

## negative-power-fixed.ts

```
1 console.log((-3) ** 2);
2 console.log(-(3 ** 2));
```

```
9
-9
```

`(-3) ** 2` squares -3, giving 9; `-(3 ** 2)` squares 3 and then makes it negative, giving -9. Delete `negative-power.ts` once you have read its message.

**PAUSE AND TRY****Question 10** Tracing

Predict exactly what this program shows. Show the order of the steps.

predict-precedence.ts

```
1 console.log(7 + 8 % 3 * 2);
2 console.log((7 + 8) % 4 * 2);
3 console.log(2 * 3 ** 2);
```

**Question 11** Multiple choice

Which expression has the value 25?

- A. `-5 ** 2`
- B. `(-5) ** 2`
- C. `10 + 5 ** 2 % 2`
- D. `5 + 5 * 2`

## 2.10 Converting with Number and String

Numbers do not always arrive as numbers. On Day 20 your programs will read what people type into a web form, and whatever they type arrives as text: the string `"15"`, not the number `15`. Files hold text too. The digits look the same, but the type is different, and the type decides what you can do:

text-times-number.ts

```
1 const ageText = "15";
2 console.log(ageText * 2);
```

```
text-times-number.ts:2:13 - error TS2362: The left-hand side of an arithmetic operation
must be of type 'any', 'number', 'bigint' or an enum type.
```

An **operand** is a value an operator works on, and the **left-hand side** is the operand to the left of the operator, here `ageText`. The message lists the types that `*` accepts there; of those, the only one you know is `number` (the others are for later or for other situations). So it says: `*` needs a number on its left, and `ageText` is a string. The type checker is protecting you: arithmetic on text is almost always a mistake. If you typed this file, fix it with `Number()` (shown next) or delete it before your next check.

The fix is **type conversion**: making a new value of a different type from an existing one. Two built-in functions do it:

- `Number(value)` makes a number. Given a string of digits, such as `"42"` or `"1.75"`, it gives that number.
- `String(value)` makes a string: the text you would see if you showed the value.

conversions.ts

```
1 const fromText = Number("42");
2 const fromDecimal = Number("1.75");
3 const asText = String(42);
4 console.log(fromText, typeof fromText);
5 console.log(fromDecimal, typeof fromDecimal);
6 console.log(asText, typeof asText);
7 console.log(String(true), typeof String(true));
```

```
42 number
1.75 number
42 string
true string
```

Line 3's `asText` shows as `42`, the same as line 1's number, because `console.log` shows strings without quotes. Only `typeof` reveals that it is text. Converting makes a **new** value; it does not change the original. After `Number(priceText)`, `priceText` is still a string.

### WORKED EXAMPLE Prices that arrive as text

ticket-from-text.ts

```
1 const priceText = "12";
2 const quantityText = "3";
3 const price = Number(priceText);
4 const quantity = Number(quantityText);
5 console.log("Total:", price * quantity);
6 console.log(typeof priceText, typeof price);
```

```
Total: 36
string number
```

| Line | Name         | Value | Type   |
|------|--------------|-------|--------|
| 1    | priceText    | "12"  | string |
| 2    | quantityText | "3"   | string |
| 3    | price        | 12    | number |
| 4    | quantity     | 3     | number |

Line 5 multiplies the two numbers,  $12 \times 3 = 36$ . Line 6 confirms that converting did not change `priceText`: it is still a string, while `price` is a number. The checker knows this too: it infers `number` for `price`, because `Number(...)` always gives back a number.

Now vary the text: give `Number` something that is not a number.

## number-from-word.ts

```
1 const seats = Number("forty");
2 console.log(seats, typeof seats);
3 console.log("Seats for two buses:", seats * 2);
```

```
NaN number
Seats for two buses: NaN
```

`Number("forty")` cannot read a word as a number, so it gives back `NaN`, short for "Not a Number." `NaN` is a special value that means "the result of a calculation that failed." Its name is confusing, because its type is `number`, as line 2 shows. And any arithmetic involving `NaN` gives `NaN` again, so one bad conversion spreads through every result that uses it.

Notice who reported a problem: **nobody**. The type checker accepted the program, because `Number(...)` always gives a number, and `seats * 2` is a number times a number. Node ran it without a runtime error, because producing `NaN` is not an error in JavaScript. The only sign is `NaN` in the output. (Day 4 shows how a program can check for it.)

`Number` has a few more rules worth knowing:

## conversion-edges.ts

```
1 console.log(Number(" 12 "));
2 console.log(Number(""));
3 console.log(Number("12px"));
4 console.log(Number("3,5"));
```

```
12
0
NaN
NaN
```

Spaces at either end are ignored (" 12 " is 12), and empty text gives 0, not `NaN`. But anything extra inside makes the whole conversion fail: "12px" is `NaN`, and so is "3,5", because a number uses a dot, not a comma, for its decimal point.

## KEY IDEA

`Number(text)` makes a number from text and `String(value)` makes text from a value; both make a new value. Text that is not a number becomes `NaN`, a `number` value meaning "failed," which neither the checker nor Node reports.

## WATCH OUT

**NaN far from its cause.** Because `NaN` spreads silently, it often shows up many lines after the conversion that made it. When you see `NaN` in the output, work backward: find the first variable that holds `NaN` (show it with `console.log`), then look at the text that was converted to make it.

**PAUSE AND TRY****Question 12** Tracing

Predict exactly what this program shows.

predict-conversions.ts

```
1 console.log(Number("8") * 2);
2 console.log(String(5 + 5), typeof String(5 + 5));
3 console.log(Number("8 apples"));
4 console.log(Number(" 2.5 "));
```

**Question 13** Debugging

This program should show how many years until age 18. `npx tsc` reports an error. Predict the message, explain it, and write the fix.

years-left.ts

```
1 const ageText = "15";
2 const yearsLeft = 18 - ageText;
3 console.log("Years until 18:", yearsLeft);
```

## 2.11 Floats Are Close, Not Exact

Run this program, and expect a surprise:

float-surprise.ts

```
1 console.log(0.1 + 0.2);
2 console.log(1.1 * 3);
3 console.log(10 / 3);
4 console.log(1 + 2);
```

```
0.30000000000000004
3.3000000000000003
3.333333333333335
3
```

$0.1 + 0.2$  is not exactly  $0.3$ . Nothing is broken; every programming language that uses standard floating-point numbers (explained below) gets the same result. The reason is how a computer stores numbers.

A computer stores numbers in **binary**, using only the digits 0 and 1. Whole numbers of everyday size convert to binary exactly, so  $1 + 2$  is exactly 3. Many fractions do not. Think of  $1/3$  in ordinary decimal:  $0.3333\dots$ , with threes forever. You can write 0.33 or 0.3333333, but never exactly one third. In binary, 0.1 is like that: it repeats forever, so the computer stores the closest value it can fit. That stored value is a tiny bit off, the tiny errors add up, and sometimes they show in the last digit. A number stored this way is called a **floating-point number** (or **float**), and the tiny error is called **floating-point imprecision**. Every JavaScript `number` is stored as a float.

The error is tiny (about 0.00000000000000004 here), so it almost never matters for a calculation. It matters when you **show** the result: nobody wants to read a price of 3.3000000000000003. The fix is to round when you show a value. `Math.round` rounds to a whole number, so to keep two decimal places, multiply by 100, round, and divide by 100 again: `Math.round(x * 100) / 100`. For one decimal place, use 10 instead of 100.

### WORKED EXAMPLE Splitting a bill

Three friends split a 10-dollar bill.

split-bill.ts

```
1 const bill = 10;
2 const friends = 3;
3 const share = bill / friends;
4 console.log("Exact:", share);
5 console.log("Rounded:", Math.round(share * 100) / 100);
```

```
Exact: 3.333333333333335
Rounded: 3.33
```

$10 / 3$  is  $3.333\dots$ , with threes forever, and the stored float runs out of room after about 16 digits, so the last digit is off. Rounding to cents takes three steps:

| Step | Expression                   | Value              |
|------|------------------------------|--------------------|
| 1    | <code>share * 100</code>     | 333.33333333333337 |
| 2    | <code>Math.round(...)</code> | 333                |
| 3    | <code>... / 100</code>       | 3.33               |

`share` itself is unchanged: `Math.round` gives back a new value, just like the conversions in 2.10. (Day 3 shows another way to show exactly two decimal places.)

**KEY IDEA**

Numbers are stored in binary as floats, so many decimals, like 0.1, are stored as a very close approximation, and tiny errors can show in the last digit. Round when you show a result:

`Math.round(x * 100) / 100` keeps two decimal places.

**WATCH OUT**

**Rounding too early.** Round only when you show a result, not in the middle of a calculation:

round-too-early.ts

```
1 const share = Math.round((10 / 3) * 100) / 100;
2 console.log("Share:", share);
3 console.log("Three shares:", share * 3);
```

```
Share: 3.33
Three shares: 9.99
```

Rounding the share to 3.33 first threw away a third of a cent, and multiplying by 3 made the loss visible: three shares total 9.99, not 10. Keep the full value for calculations and round the final answer.

**PAUSE AND TRY****Question 14** Tracing

Predict exactly what this program shows.

predict-rounding.ts

```
1 console.log(Math.round((7 / 3) * 100) / 100);
2 console.log(1.1 * 3);
3 console.log(Math.round(1.1 * 3 * 10) / 10);
```

## Putting it together

Jordan is planning a 523-kilometre road trip at an average 90 km/h. The car uses 6.8 litres of fuel per 100 km, and fuel costs 1.89 per litre. The distance arrives as text, as if typed into a form, so it has to be converted. The comment on line 1 is from Day 1; the blank lines just split the program into steps.

**First check.** Jordan's first attempt annotates the distance as a number but forgets to convert the text:

road-trip-attempt.ts

```
1 // Trip details: the distance arrives as text, as if typed into a form
2 const distanceText = "523";
3 const speedKmh = 90;
4 const litresPer100Km = 6.8;
5 const pricePerLitre = 1.89;
6
7 const distanceKm: number = distanceText;
8 console.log("Distance:", distanceKm, "km");
```

```
road-trip-attempt.ts:7:7 - error TS2322: Type 'string' is not assignable to type 'number'.
```

The annotation `: number` on line 7 states Jordan's intention, so the checker can hold the line to it: `distanceText` is a string. Nothing has run yet. The fix is to convert with `Number` (2.10).

**Check and run.** Jordan fixes line 7 with the conversion. If you typed along, fix `road-trip-attempt.ts` the same way, or delete it and type the fixed program as `road-trip.ts`, so that no broken file is left. Now `npx tsc` reports no errors for the trip program, and `node road-trip.ts` shows:

road-trip.ts

```
1 // Trip details: the distance arrives as text, as if typed into a form
2 const distanceText = "523";
3 const speedKmh = 90;
4 const litresPer100Km = 6.8;
5 const pricePerLitre = 1.89;
6
7 const distanceKm: number = Number(distanceText);
8
9 const totalMinutes = Math.floor((distanceKm * 60) / speedKmh);
10 const hours = Math.floor(totalMinutes / 60);
11 const minutes = totalMinutes % 60;
12
13 const fuelLitres = (distanceKm / 100) * litresPer100Km;
14 const fuelCost = fuelLitres * pricePerLitre;
15
16 console.log("Distance:", distanceKm, "km");
17 console.log("Driving time:", hours, "hours", minutes, "minutes");
18 console.log("Fuel needed:", Math.round(fuelLitres * 10) / 10, "litres");
19 console.log("Fuel cost:", Math.round(fuelCost * 100) / 100);
```

```
Distance: 523 km
Driving time: 5 hours 48 minutes
Fuel needed: 35.6 litres
Fuel cost: 67.22
```

Follow the values and types, in the order the lines run:

| Name (line) | Worked out | Value | Type |
|-------------|------------|-------|------|
|-------------|------------|-------|------|

|                  |                                                       |          |        |
|------------------|-------------------------------------------------------|----------|--------|
| distanceText (2) | A string in quotes                                    | "523"    | string |
| distanceKm (7)   | Number("523") (2.10)                                  | 523      | number |
| totalMinutes (9) | 523 * 60 is 31380, / 90 is 348.67, rounded down (2.8) | 348      | number |
| hours (10)       | Math.floor(348 / 60), 5.8 rounded down (2.8)          | 5        | number |
| minutes (11)     | 348 % 60, since 5 × 60 = 300 (2.8)                    | 48       | number |
| fuelLitres (13)  | 523 / 100 is 5.23, times 6.8                          | 35.564   | number |
| fuelCost (14)    | fuelLitres * 1.89                                     | 67.21596 | number |

Every name is a `const` (2.2), because nothing changes after it is worked out. The last two lines round only when showing (2.11): 35.564 shows as 35.6 and 67.21596 as 67.22. Notice line 9: the parentheses make the program multiply before dividing, so the minutes are worked out from the full distance and then rounded down once. Writing `Math.floor(distanceKm / speedKmh) * 60` instead would round down to 5 whole hours **before** converting to minutes and lose the 48 extra minutes.

## Common mistakes

Every message from today in one place, to look up when you meet one again. All of them come from `npx tsc`, before anything runs, except the last row.

| Message                                                            | Usual cause                                                       | Fix                                        |
|--------------------------------------------------------------------|-------------------------------------------------------------------|--------------------------------------------|
| TS2448: Block-scoped variable 'total' used before its declaration. | Using a variable above the line that declares it                  | Move the declaration up                    |
| TS2451: Cannot redeclare block-scoped variable 'price'.            | <code>let</code> or <code>const</code> again for an existing name | Assign without <code>let</code>            |
| TS2588: Cannot assign to 'busSeats' because it is a constant.      | Assigning to a <code>const</code>                                 | Use <code>let</code> , or do not change it |
| TS2552: Cannot find name 'Score'. Did you mean 'score'?            | Wrong capitals in a variable name                                 | Use the suggested name                     |
| TS1389: 'class' is not allowed as a variable declaration name.     | A reserved word as a name                                         | Choose another name                        |
| TS2322: Type 'string' is not assignable to type 'number'.          | A value of a different type than the variable's                   | Fix the value, or the declaration          |
| TS2322: Type 'null' is not assignable to type 'number'.            | <code>null</code> stored in a <code>number</code> in strict mode  | Store a number                             |
| TS2454: Variable 'finalScore' is used before being assigned.       | Reading a <code>let</code> with no value yet                      | Assign a value first                       |

|                                                                                                         |                                                           |                                                  |
|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|--------------------------------------------------|
| TS2362 /<br>TS2363: The left-hand side / right-hand side of an arithmetic operation must be of type ... | Arithmetic with a string                                  | Convert with <code>Number()</code> first         |
| TS17006: An unary expression with the '-' operator is not allowed ...                                   | <code>-3 ** 2</code>                                      | <code>(-3) ** 2</code> or <code>-(3 ** 2)</code> |
| No message, but <code>NaN</code> or <code>Infinity</code> in the output                                 | Converting text that is not a number, or dividing by zero | Fix the text, or the divisor                     |

## Vocabulary

| Term                                   | Definition                                                                         | Example                                                    |
|----------------------------------------|------------------------------------------------------------------------------------|------------------------------------------------------------|
| <b>variable</b>                        | A name that holds a value                                                          | <code>score</code>                                         |
| <b>variable declaration</b>            | A statement that makes a new variable, with <code>let</code> or <code>const</code> | <code>let score = 10;</code>                               |
| <b>keyword / reserved word</b>         | A word with a fixed meaning in the language / one that cannot be a name            | <code>let</code> , <code>const</code> , <code>class</code> |
| <b>assignment operator / assigning</b> | The <code>=</code> sign / storing a value in a variable                            | <code>score = 0;</code>                                    |
| <b>constant</b>                        | A variable declared with <code>const</code> , which can never be assigned again    | <code>const ticketPrice = 12;</code>                       |
| <b>identifier</b>                      | A name you choose, following the naming rules                                      | <code>ticketsSold</code>                                   |
| <b>camelCase</b>                       | Words joined with no spaces, each after the first starting with a capital          | <code>maxSpeedKmh</code>                                   |
| <b>number</b>                          | The type of every number, whole or decimal                                         | <code>42</code> , <code>0.45</code>                        |
| <b>string</b>                          | The type of text in quotes                                                         | <code>"Hatchet"</code>                                     |
| <b>boolean</b>                         | The type whose only values are <code>true</code> and <code>false</code>            | <code>const isOpen = true;</code>                          |
| <b>typeof</b>                          | Gives a value's type name as a string while the program runs                       | <code>typeof 42</code> is <code>"number"</code>            |
| <b>type annotation</b>                 | <code>: type</code> after a variable's name, telling the checker its type          | <code>let visitors: number = 0;</code>                     |
| <b>type inference</b>                  | The checker working out a type from the starting value                             | <code>let visitors = 0;</code> is <code>number</code>      |

|                                            |                                                                                        |                                                             |
|--------------------------------------------|----------------------------------------------------------------------------------------|-------------------------------------------------------------|
| <b>static typing</b>                       | A variable's type is fixed when it is declared and checked before running              | <code>count = "five"</code> is rejected                     |
| <b>null / undefined</b>                    | Deliberately empty / not assigned yet; each the only value of its own type             | <code>const middleName = null;</code>                       |
| <b>remainder (%) / exponent (**)</b>       | What is left over after dividing / raising to a power                                  | <code>135 % 60</code> is 15, <code>2 ** 10</code> is 1024   |
| <b>Math.floor / Math.round</b>             | Round down / round to the nearest whole number                                         | <code>Math.floor(2.25)</code> is 2                          |
| <b>Infinity</b>                            | The number that dividing by zero gives                                                 | <code>12 / 0</code>                                         |
| <b>operator precedence</b>                 | The order in which operators are applied                                               | <code>**</code> before <code>*</code> before <code>+</code> |
| <b>unary operator</b>                      | An operator that works on one value                                                    | the <code>-</code> in <code>-3</code>                       |
| <b>operand</b>                             | A value an operator works on                                                           | <code>ageText</code> in <code>ageText * 2</code>            |
| <b>type conversion</b>                     | Making a new value of another type with <code>Number()</code> or <code>String()</code> | <code>Number("42")</code> is 42                             |
| <b>NaN</b>                                 | "Not a Number": a number value meaning a calculation failed                            | <code>Number("forty")</code>                                |
| <b>floating-point number / imprecision</b> | A number stored approximately in binary / the tiny errors that causes                  | <code>0.1 + 0.2</code>                                      |

## Summary

- A **variable** holds a value under a name. `const` makes one that can never be assigned again, and `let` makes one that can; use `const` unless the value must change. Names follow the **identifier** rules and, by convention, are descriptive **camelCase**.
- Every value has a **type**: **number** (whole or decimal), **string**, **boolean** (`true` or `false`), **null**, or **undefined**. `typeof` reports it while the program runs (with the quirk that `typeof null` is `"object"`).
- The checker knows each variable's type from a **type annotation** or by **inference**. With **static typing**, that type never changes: `count = "five"` for a **number** is rejected before anything runs, and so are `null` in a **number** and reading a variable before it is assigned.
- `%` gives the remainder, `**` raises to a power, `Math.floor` rounds down, and `Math.round` rounds to the nearest whole number. **Precedence**: parentheses, then `**`, then `*` / `%`, then `+` - , left to right within a row.
- `Number()` and `String()` make new values of another type. Text that is not a number becomes **NaN**, and dividing by zero gives **Infinity**; neither the checker nor Node reports them.
- Numbers are stored as **floats**, so `0.1 + 0.2` shows a tiny error. Calculate with full values and round only for display, with `Math.round(x * 100) / 100`.

## Practice

Do these in your `day02` folder. First make sure `npx tsc` prints nothing there: if any broken file from today is still in the folder (such as a question's `price-change.ts`, `lunch-sales.ts`, `thermostat.ts`, or `years-left.ts`), fix it or delete it. Then type each program yourself, check it with `npx tsc`, run it with `node`, and only then open the answer. Change the starting values and run them again to test your program.

### Question 15 Short answer

**Fahrenheit to Celsius.** Write a program that stores a temperature in Fahrenheit in a constant, converts it to Celsius with the formula  $C = (F - 32) \times 5 \div 9$ , and shows both, with the Celsius value rounded to one decimal place. Try 72 and 212.

### Question 16 Short answer

**Seconds to hours, minutes, and seconds.** Store a number of seconds in a constant, such as 7384. Use `Math.floor` and `%` to work out the hours, the minutes, and the seconds left over, and show them on one line.

### Question 17 Short answer

**Compound interest.** A savings account starts with a **principal** (the starting amount) and grows by a fixed **rate** each year, with each year's interest added to the balance. After `years` years the balance is  $\text{principal} \times (1 + \text{rate})^{\text{years}}$ . Store a principal of 1000, a rate of 0.05 (5%), and 10 years in constants; show the final amount and the interest earned, both rounded to two decimal places.

## Review questions

### Question 18 Multiple choice

A program starts with `let rainfall = 12.5;`. Which next line does `npx tsc` accept?

- A. `rainfall = "13";`
- B. `rainfall = 13;`
- C. `rainfall = null;`
- D. `let rainfall = 13;`

**Question 19** Tracing

This program makes change for 87 cents using quarters (25 cents) and dimes (10 cents). What does it show? Track `cents`, `quarters`, and `dimes`.

`coins.ts`

```
1 let cents = 87;
2 const quarters = Math.floor(cents / 25);
3 cents = cents % 25;
4 const dimes = Math.floor(cents / 10);
5 cents = cents % 10;
6 console.log(quarters, dimes, cents);
```

**Question 20** Debugging

Lena wants to show the average of the scores 80 and 90, rounded to a whole number. Her program has two mistakes: one makes `npx tsc` reject it, and one would give the wrong answer even after the first is fixed. Find both and write the fixed program.

`average-two-mistakes.ts`

```
1 const scoreA = 80;
2 const scoreB = 90;
3 const average = scoreA + scoreB / 2;
4 average = Math.round(average);
5 console.log("Average:", average);
```

**Question 21** Tracing

Predict exactly what this program shows, values and types.

## types-trace.ts

```
1 const half = 7 / 2;
2 const doubledText = String(half * 2);
3 const fromWord = Number("seven");
4 console.log(half, typeof half);
5 console.log(doubledText, typeof doubledText);
6 console.log(fromWord, typeof fromWord);
7 console.log(typeof null);
```

**Question 22** Short answer

Mia's program shows Price for 3: NaN, even though `npx tsc` reported no errors for it and `node` reported no error. Explain why neither the type checker nor Node reported a problem, and fix the program.

## mia-price.ts

```
1 const priceText = "4.50 dollars";
2 const price = Number(priceText);
3 console.log("Price for 3:", price * 3);
```

**Question 23** Multiple choice

This program shows four lines. Which line does **not** show 4?

## not-four.ts

```
1 console.log(Math.round(3.5));
2 console.log(Math.floor(4.99));
3 console.log(14 % 5);
4 console.log(-Math.floor(-4.5));
```

- A. Line 1
- B. Line 2
- C. Line 3
- D. Line 4

**Go further**

The TypeScript Handbook covers today's types and annotations in more detail, and MDN explains declarations and JavaScript's values: TypeScript Handbook: Everyday Types (read the primitives and type annotations sections) and MDN: Grammar and types.

## Reflection

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

- ☐ I can create a variable with `let` and trace how its value changes. Reread: 2.1 Variables: Giving a Value a Name
- ☐ I can choose between `const` and `let`, and explain why `const` is the default. Reread: 2.2 Constants: Names That Never Change
- ☐ I can tell whether a name is legal and write it in camelCase. Reread: 2.3 Naming Rules and camelCase
- ☐ I can name the type of a number, a string, or a boolean, and use `typeof`. Reread: 2.4 Numbers, Strings, and Booleans
- ☐ I can write a type annotation and say what type the checker infers without one. Reread: 2.5 Type Annotations and Inference
- ☐ I can say whether the type checker will accept an assignment. Reread: 2.6 Static Typing: A Variable Keeps Its Type
- ☐ I can explain `null` and `undefined` and what strict mode does with them. Reread: 2.7 Null and Undefined
- ☐ I can use `%`, `**`, `Math.floor`, and `Math.round`, including with negative numbers. Reread: 2.8 Remainders, Powers, and Rounding
- ☐ I can evaluate an expression in precedence order. Reread: 2.9 Operator Precedence
- ☐ I can convert values with `Number()` and `String()` and explain where `NaN` comes from. Reread: 2.10 Converting with Number and String
- ☐ I can explain why `0.1 + 0.2` is not exactly `0.3` and round a result for display. Reread: 2.11 Floats Are Close, Not Exact