

Day 3 Strings

Self-Study Guide · About 70 minutes · With answer key

By the end you will be able to

- Write strings with quotes, escape characters, and template literals, and predict what `+` does when one side is a string.
- Pull any part out of a string with `length`, indexing, `at`, `slice`, `indexOf`, and `split`.
- Clean up and change text with string methods, knowing that each one gives back a new string.
- Build formatted output with template literals, `padStart`, and `toFixed`.

What to remember from earlier

- A string is text in quotes, and `console.log` shows it without the quotes. ([Day 1: Setup and your first program, 1.3 Your First Program](#))
- A runtime error, such as the one from `"=".repeat(-12)`, 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](#))
- Use `const` unless the value must change; a `let` variable can be assigned a new value. ([Day 2: Values, variables and types, 2.2 Constants: Names That Never Change](#))
- A boolean is `true` or `false`, written in small letters without quotes. ([Day 2: Values, variables and types, 2.4 Numbers, Strings, and Booleans](#))
- Operators on the same row of the precedence table are applied left to right, and parentheses go first. ([Day 2: Values, variables and types, 2.9 Operator Precedence](#))
- Arithmetic such as `*` or `-` on a string is a type error; `Number()` makes a number from text and `String()` makes text from a value. ([Day 2: Values, variables and types, 2.10 Converting with Number and String](#))
- Numbers are stored as binary floats, so `0.1 + 0.2` shows a tiny error, and a result should be rounded only when it is shown. ([Day 2: Values, variables and types, 2.11 Floats Are Close, Not Exact](#))

3.1 Quotes and Escape Characters

Before you start, set up a new project folder named `day03` 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.

Almost every program shows text to a person: a greeting, a total, a label on a receipt. On Day 1 you learned that a piece of text in quotes is a **string**, with the type `string`, and that `console.log` shows it without the quotes. Today is all about strings: how to write them, how to build them from other values, and how to pull pieces out of them.

Start with a puzzle. The quotes mark where a string starts and ends. So how do you write a string that **contains** a quote, such as the sign `She said "hi"`?

The first answer is that TypeScript accepts two kinds of quotes. A string can go inside double quotes ("Lima") or single quotes ('Lima'); both make exactly the same string. A string must end with the same kind of quote it started with, so the other kind can sit inside it as an ordinary character.

quotes.ts

```
1 const city = "Lima";
2 const country = 'Peru';
3 console.log(city, country);
4 console.log("It's open");
5 console.log('She said "hi"');
```

```
Lima Peru
It's open
She said "hi"
```

Line 2 uses single quotes and makes the same kind of value as line 1: both are strings, and neither shows its quotes. Line 4 is in double quotes, so the apostrophe in `It's` is just a character. Line 5 is in single quotes, so the double quotes inside are just characters. This course writes strings in double quotes, and uses single quotes only for text that contains double quotes.

WATCH OUT

An apostrophe inside single quotes. The string ends at the **first** matching quote, wherever it is:

apostrophe-broken.ts

```
1 const message = 'It's raining';
2 console.log(message);
```

```
apostrophe-broken.ts:1:21 - error TS1005: ',' expected.
apostrophe-broken.ts:1:23 - error TS1005: ',' expected.
apostrophe-broken.ts:1:30 - error TS1005: ',' expected.
apostrophe-broken.ts:1:32 - error TS1002: Unterminated string literal.
```

The checker reads `'It'` as the whole string, because the apostrophe (column 20) matches the opening quote. Then it finds `s` (column 21) and `raining` (column 23) where nothing should be, and the last quote (column 30) starts a new string that never ends: Day 1's TS1002. As Day 1 taught, fix the **first** message and the rest follow. Put the text in double quotes: `"It's raining"`.

This is a **syntax error**, and syntax errors have a side effect worth knowing: while any file in your folder has one, `npm tsc` reports only syntax errors, so type errors in your other files stay hidden until you fix it. If you typed this file, fix or delete it now.

What if the text contains **both** kinds of quotes, or a character you cannot type into a string at all, such as a line break? For those, a string can hold **escape characters**: a backslash `\` followed by a letter or symbol, which together stand for one special character. The backslash tells the checker, "the next character does not have its usual meaning."

Write	You get	Why it is needed
<code>\"</code>	<code>"</code>	A double quote that does not end the string
<code>\'</code>	<code>'</code>	A single quote that does not end the string
<code>\\</code>	<code>\</code>	A backslash itself, since <code>\</code> starts an escape
<code>\n</code>	A newline: the text continues on a new line	Pressing Enter inside quotes would break the line
<code>\t</code>	A tab: a jump to the next column	Lining text up in columns

WORKED EXAMPLE Escapes in action

Each line uses one escape. Predict what each shows before you read the output.

escapes.ts

```
1 console.log("She said \"hi\"");
2 console.log('It\'s open');
3 console.log("Line one\nLine two");
4 console.log("Name:\tAna");
5 console.log("C:\\day03\\notes.txt");
```

```
She said "hi"
It's open
Line one
Line two
Name: Ana
C:\day03\notes.txt
```

Line	Escape	What happens
1	<code>\"</code>	Both inner quotes are shown; the string ends only at the last, unescaped <code>"</code>
2	<code>\'</code>	The apostrophe is shown, even inside single quotes
3	<code>\n</code>	One <code>console.log</code> shows two lines: the newline starts <code>Line two</code> on a new line
4	<code>\t</code>	A tab puts a wide gap after <code>Name: </code> ; a terminal jumps to its next column, which is why tabs are used to line text up
5	<code>\\</code>	Each pair shows one backslash; a single <code>\</code> would have started an escape such as <code>\d</code>

An escape is two characters in your code but **one** character in the string: `"\n"` holds a single newline character, not a backslash and an `n`.

KEY IDEA

A string goes in double or single quotes, and ends at the next matching quote. Inside it, a backslash starts an escape: `\"` and `\'` for quotes, `\\` for a backslash, `\n` for a new line, and `\t` for a tab.

PAUSE AND TRY

Question 1 Tracing

Predict exactly what this program shows. How many lines of output are there?

predict-escapes.ts

```

1 console.log("Tea\nCoffee");
2 console.log("A\\nB");
3 console.log('Say "yes"');
4 console.log("Don\'t\tstop");

```

Question 2 Multiple choice

Which line shows exactly Don't say "no"?

- A. `console.log('Don't say "no"');`
- B. `console.log("Don't say "no");`
- C. `console.log("Don't say \"no\");`
- D. `console.log("Don't say \\\"no\\");`

3.2 Joining Strings with +

A program often has the pieces of a message in separate variables: a first name in one, a last name in another. On Day 1, `console.log` put several values on one line with spaces between them. But that only **shows** the values; it does not make one string you can store, and it always puts in exactly one space. To build a new string out of pieces, you join them.

You already know `+` adds numbers. Between two strings, `+` does something different: it glues them together, end to end, into one new string. Joining strings this way is called **concatenation**. "Hot" + "dog" is "Hotdog", with no space, because `+` adds exactly the characters it is given and nothing else.

WORKED EXAMPLE A name badge

name-badge.ts

```

1 const firstName = "Ana";
2 const lastName = "Lee";
3 const fullName = firstName + " " + lastName;
4 console.log(fullName);
5 console.log("Hello, " + fullName + "!");
6 console.log(firstName + lastName);

```

```

Ana Lee
Hello, Ana Lee!
AnaLee

```

Line	Expression	Worked out, left to right	Value
3	<code>firstName + " " + lastName</code>	"Ana" + " " is "Ana ", then + "Lee"	"Ana Lee"
5	<code>"Hello, " + fullName + "!"</code>	"Hello, " + "Ana Lee" is "Hello, Ana Lee", then + "!"	"Hello, Ana Lee!"
6	<code>firstName + lastName</code>	Nothing between them	"AnaLee"

Line 3 joins three strings; the middle one, " ", is a string holding a single space. Line 6 shows what happens without it. Every space you want in the result must be inside some string's quotes.

Now vary the types: put a number on one side of `+`. The rule is: if **either** side of `+` is a string, `+` joins, and the other side is first turned into text the way `String()` from Day 2 would turn it. Only when both sides are numbers does `+` add. And as Day 2 taught, operators on the same row of the precedence table are applied left to right, one pair at a time, so the rule is applied to each `+` in turn.

mixed-plus.ts

```
1 console.log("5" + 1);
2 console.log(5 + 1);
3 console.log(1 + 2 + "3");
4 console.log("1" + 2 + 3);
5 console.log("Total: " + 2 + 3);
6 console.log("Total: " + (2 + 3));
7 console.log(typeof ("5" + 1));
```

```
51
6
33
123
Total: 23
Total: 5
string
```

Line	Expression	Step by step	Result
1	"5" + 1	A string and a number: join "5" and "1"	"51"
2	5 + 1	Two numbers: add	6
3	1 + 2 + "3"	1 + 2 is 3 (two numbers), then 3 + "3" joins	"33"
4	"1" + 2 + 3	"1" + 2 is "12" (a string), then "12" + 3 joins	"123"
5	"Total: " + 2 + 3	"Total: " + 2 is "Total: 2", then + 3 joins again	"Total: 23"
6	"Total: " + (2 + 3)	Parentheses first: 5, then join	"Total: 5"
7	typeof ("5" + 1)	The result of joining is a string	"string"

So `"5" + 1` is `"51"`, not 6: the quotes make `"5"` text, and text plus anything is more text. Lines 3 and 4 have the same three values in a different order and give different answers, because the first `+` decides whether the running result is a number or a string. Line 5 is the classic bug: a label followed by a calculation. Put the calculation in parentheses, as on line 6.

Why does the type checker allow a string plus a number at all, when Day 2 showed it rejects `ageText * 2`? Because joining a number into a message is something programs do all the time, and the result is always a sensible string. The checker knows the rule too: it gives `"5" + 1` the type `string`. But the other arithmetic operators have no meaning for text, so the checker still rejects them:

minus-text.ts

```
1 const stock = "5";
2 console.log(stock - 1);
```

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

This is Day 2's TS2362, pointing at `stock` (column 13). Fix it with `Number(stock) - 1`, or delete the file, before your next check.

Expression	npx tsc	Result	Why
"5" + 1	Accepts	"51"	+ with a string joins
5 + 1	Accepts	6	+ with two numbers adds
"5" - 1	TS2362	(none)	-, *, /, %, ** need numbers

KEY IDEA

`+` between two strings joins them (concatenation). If either side is a string, `+` joins and turns the other side into text, so `"5" + 1` is `"51"`. Each `+` is applied left to right, so put a calculation inside a message in parentheses.

WATCH OUT

A number that is really text. Numbers that arrive as text, as on Day 2, cause a quiet bug with `+`:

tickets-text.ts

```
1 const adultTickets = "3";
2 const childTickets = 2;
3 console.log("Tickets:", adultTickets + childTickets);
4 console.log("Tickets:", Number(adultTickets) + childTickets);
```

```
Tickets: 32
Tickets: 5
```

Three adult tickets and two child tickets should make 5, but line 3 shows 32. `adultTickets` holds the string `"3"`, so `+` joins `"3"` and `"2"`. Neither `npx tsc` nor `node` reports anything, because joining a string and a number is allowed. Line 4 converts the text with `Number()` first, so `+` sees two numbers and adds. When a sum comes out looking like two numbers stuck together, look for a string.

PAUSE AND TRY**Question 3** Tracing

Predict exactly what this program shows. Work each line out one `+` at a time.

predict-plus.ts

```
1 const apples = 4;
2 const pears = 6;
3 console.log("Fruit: " + apples + pears);
4 console.log(apples + pears + " pieces of fruit");
5 console.log("Fruit: " + (apples + pears));
6 console.log("2" + "2");
```

Question 4 Multiple choice

points is a constant holding the number 10. Which line shows Score: 15?

- A. console.log("Score: " + points + 5);
- B. console.log("Score: " + (points + 5));
- C. console.log("Score: points + 5");
- D. console.log("Score: " - points - 5);

3.3 Template Literals

Concatenation works, but a long message built with + is hard to read and easy to get wrong. Look at line 4 of this program: every space has to be inside a string, and each + has to be in the right place.

order-summary.ts

```
1 const item = "muffin";
2 const price = 2.5;
3 const quantity = 4;
4 console.log("Order: " + quantity + " x " + item + " = " + quantity * price);
5 console.log(`Order: ${quantity} x ${item} = ${quantity * price}`);
6 console.log(`Each ${item} costs ${price}.`);
7 console.log(`Next order: ${quantity + 1} items`);
```

```
Order: 4 x muffin = 10
Order: 4 x muffin = 10
Each muffin costs 2.5.
Next order: 5 items
```

Line 5 makes exactly the same text in a much more readable way. It is a **template literal**: a string written between **backticks** ` instead of quotes. (On many keyboards the backtick key is at the top left, next to the 1 key.) Inside a template literal, you can write `${expression}`, called a **placeholder**: when the line runs, the program works out the expression, turns the result into text, and puts it in the string at that spot. Everything outside the placeholders is copied exactly, spaces included.

```
`text ${expression} more text`
```

Piece	What it is
Backticks	Start and end the template literal
text	Ordinary characters, copied as written

<code>\${expression}</code>	A placeholder: a dollar sign, then any expression (a variable, a calculation) in curly braces
-----------------------------	---

Read line 5 the way the program works through it. `${quantity}` becomes 4, `${item}` becomes muffin, and `${quantity * price}` is worked out to 10 before it is put in. Line 6 uses a placeholder inside a sentence, and line 7 shows that a placeholder can hold a calculation: `${quantity + 1}` is 5. Inside a placeholder, the expression is worked out on its own before it is put in, so `${quantity + 1}` adds two numbers without the extra parentheses 3.2 needed. A template literal is still a string: its type is `string`.

Now vary one more thing: press Enter inside a template literal. Unlike quotes, backticks can span several lines, and every line break and every space between the backticks becomes part of the string. A string that spans several lines like this is a **multi-line string**.

WORKED EXAMPLE A poster on three lines

poster.ts

```
1 const show = "Robot Dance";
2 const time = "7:30";
3 const poster = `${show}
4 Tonight at ${time}
5 Tickets at the door`;
6 console.log(poster);
```

```
Robot Dance
Tonight at 7:30
  Tickets at the door
```

`poster` holds one string with two newline characters in it, exactly where lines 3 and 4 of the code end. The two spaces at the start of line 5 are inside the backticks, so they are part of the string, and the third output line is indented. With double quotes, you would write the same string as `show + "\nTonight at " + time + "\n Tickets at the door"`. A string in ordinary quotes cannot continue onto a new line at all: pressing Enter before the closing quote gives Day 1's TS1002: Unterminated string literal as the first message.

A template literal has one more advantage: the type checker reads the expressions in its placeholders, just like any other code. A typo there is caught before anything runs:

template-typo.ts

```
1 const total = 12;
2 console.log(`Total: ${totl}`);
```

```
template-typo.ts:2:23 - error TS2552: Cannot find name 'totl'. Did you mean 'total'?
```

Column 23 is where `totl` starts, inside the placeholder. Fix the spelling, or delete the file, before your next check.

KEY IDEA

A template literal goes between backticks, and each `${expression}` placeholder is replaced by the expression's value. Template literals can span lines, and every line break and space inside them is part of the string.

WATCH OUT

A placeholder in ordinary quotes. `${...}` works only between backticks. In double or single quotes, it is just ordinary characters:

wrong-quotes.ts

```
1 const total = 12;
2 console.log("Total: ${total}");
3 console.log(`Total: ${total}`);
```

```
Total: ${total}
Total: 12
```

Line 2 shows the dollar sign, the braces, and the word, exactly as typed. Nobody reports a problem: to the checker, "Total: \${total}" is an ordinary string of 15 characters, and Node shows it faithfully. When you see `${` in your output, change the quotes to backticks.

PAUSE AND TRY**Question 5** Tracing

Predict exactly what this program shows.

predict-template.ts

```
1 const width = 3;
2 const height = 4;
3 console.log(`${width} x ${height}`);
4 console.log(`Area: ${width * height}`);
5 console.log(`${width} + ${height} = ${width + height}`);
6 console.log("Perimeter: ${2 * (width + height)}");
```

3.4 Length and Indexing

To print someone's initials, a program needs the **first letter** of their name. To check a password is long enough, it needs to know **how many** characters it has. Both questions are about the characters inside a string.

A string is a sequence of characters, one after another, in order. Each character has a numbered position called its **index**. The first character's index is **0**, the next is 1, and so on. Think of the index as "how many characters come before this one": nothing comes before the first character, so its index is 0. Here are the indexes in "Hopper":

Character	H	o	p	p	e	r
Index	0	1	2	3	4	5
Counting from end	-6	-5	-4	-3	-2	-1

Three tools read this sequence:

- `text.length` is the number of characters in `text`. `"Hopper".length` is 6. Every character counts: spaces, punctuation, and each escape such as `\n` (one character, as 3.1 showed). `length` is a **property**: a value that belongs to another value, read with a dot and **no** parentheses, just as `console.log` reads `log` from `console`.
- `text[i]` gives the character at index `i`, as a string of length 1. The square brackets `[ ]` do the **indexing**. Since counting starts at 0, the last index is always `text.length - 1`.
- `text.at(i)` also gives the character at index `i`, but it accepts negative indexes, which count from the end: `at(-1)` is the last character, `at(-2)` the one before it. `at` is a **method**: a function that belongs to a value, called with a dot and parentheses, such as `word.at(-1)`. You have used one already: Day 1's `"=".repeat(12)` called the string method `repeat`. The value in the parentheses is the method's argument, as with `console.log`.

WORKED EXAMPLE Reading the characters of Hopper

indexing.ts

```

1 const word = "Hopper";
2 console.log(word.length);
3 console.log(word[0]);
4 console.log(word[3]);
5 console.log(word[word.length - 1]);
6 console.log(word.at(-1));
7 console.log(word.at(-2));
8 console.log(word[6]);
9 console.log(word[-1]);

```

```

6
H
p
r
r
e
undefined
undefined

```

Line	Expression	Worked out	Value
2	<code>word.length</code>	Six characters	6
3	<code>word[0]</code>	The first character	"H"
4	<code>word[3]</code>	Three characters come before it: H, o, p	"p"
5	<code>word[word.length - 1]</code>	<code>word[6 - 1]</code> , which is <code>word[5]</code>	"r"
6	<code>word.at(-1)</code>	The last character	"r"
7	<code>word.at(-2)</code>	The second to last	"e"
8	<code>word[6]</code>	No character has index 6	undefined

9	<code>word[-1]</code>	Square brackets do not count from the end	undefined
---	-----------------------	---	-----------

Lines 5 and 6 are two ways to reach the last character: `word[word.length - 1]` works out the last index from the length, and `at(-1)` counts back from the end. Lines 8 and 9 ask for a position that does not exist. Neither tool reports an error; the result is Day 2's `undefined`, "no value here."

Now vary it: what does the type checker think of `word[6]`? It treats `word[i]` as a string for **every** `i`, because it does not work out which positions exist. So a mistake with the index slips past `npx tsc` and only shows up when the program runs:

index-too-far.ts

```
1 const word = "Hopper";
2 const last = word[word.length];
3 console.log("Last letter:", last);
4 console.log(last.length);
```

`npx tsc` reports no errors for this file, but `node index-too-far.ts` shows:

```
Last letter: undefined
TypeError: Cannot read properties of undefined (reading 'length')
```

`word.length` is 6, so line 2 asks for `word[6]`, one past the end, and `last` holds `undefined`. Line 3 shows it. Line 4 asks for `last.length`, and `undefined` has no properties at all, so Node stops with a runtime error, as on Day 1 (the full report also shows the file, line 4, and a caret). This is an **off-by-one error**: being one away from the right number, here because the last index is `length - 1`, not `length`.

`at` is different. The checker knows `at` can run off the end, so it gives its result the type `string | undefined`, read "a string or `undefined`." (Hover over `last` in VS Code to see it. This is a preview of Day 9, which teaches types like this.) The checker then refuses to let you use the result as if it were certainly a string:

at-possibly-undefined.ts

```
1 const word = "Hopper";
2 const last = word.at(-1);
3 console.log(last.length);
```

```
at-possibly-undefined.ts:3:13 - error TS18048: 'last' is possibly 'undefined'.
```

The message is exact: `last` might be `undefined`, and `undefined` has no `length`. Day 9 shows how to deal with it. Until then, use `at`'s result where any value is fine, such as in `console.log` or a template literal, and use square brackets when you need to call something on the character. Fix or delete this file before your next check.

Tool	Negative index?	Past the end gives	The checker's type
<code>word[i]</code>	No	<code>undefined</code>	<code>string</code>
<code>word.at(i)</code>	Yes, from end	<code>undefined</code>	<code>string undefined</code>

KEY IDEA

Indexes start at 0, so the last one is `length - 1`. `text[i]` and `text.at(i)` give one character; `at` also counts back from the end with `at(-1)`. A position that does not exist gives `undefined`. `length` is a property (no parentheses); `at` is a method (with parentheses).

WATCH OUT

Starting to count at 1. In everyday life the first letter of a word is "letter 1," so it is natural to write `word[1]` for the first letter. That gives `"o"` for `"Hopper"`, and no error ever appears. Whenever you write an index, ask "how many characters come before the one I want?"

PAUSE AND TRY**Question 6** Tracing

Predict exactly what this program shows. Write out the indexes of `"Lima, Peru"` first.

`predict-index.ts`

```
1 const city = "Lima, Peru";
2 console.log(city.length);
3 console.log(city[4]);
4 console.log(city[city.length - 2]);
5 console.log(city.at(-4));
6 console.log(city[10]);
```

Question 7 Multiple choice

`city` is a string of unknown length. Which expression always gives its last character?

- A. `city[city.length]`
- B. `city[-1]`
- C. `city[city.length - 1]`
- D. `city.at(1)`

3.5 Slicing

Indexing gives one character. Often you need a longer piece of a string: the year from a date like `"2026-10-16"`, or a file's name without its `.txt`. A part of a string is called a **substring**.

The method `slice(start, end)` gives a new string holding the characters from index `start` **up to, but not including**, index `end`. Leave out `end`, and the slice runs to the end of the string. Like `at`, `slice` accepts negative indexes, which count from the end.

Why is `end` left out of the slice? Because it makes the numbers easy to work with. `end - start` is exactly the number of characters you get, so `slice(0, 4)` gives 4 characters. And a slice that ends at `i` fits perfectly against one that starts at `i`: `text.slice(0, i) + text.slice(i)` is always the whole `text` again, with no character lost or doubled.

WORKED EXAMPLE Taking a date apart

date-parts.ts

```

1 const date = "2026-10-16";
2 const year = date.slice(0, 4);
3 const month = date.slice(5, 7);
4 const day = date.slice(8);
5 console.log(year, month, day);
6 console.log(date.slice(-5));

```

```

2026 10 16
10-16

```

Character	2	0	2	6	-	1	0	-	1	6
Index	0	1	2	3	4	5	6	7	8	9

Line	Call	Characters taken	Value
2	slice(0, 4)	Indexes 0, 1, 2, 3 (stops before 4)	"2026"
3	slice(5, 7)	Indexes 5, 6 (stops before 7)	"10"
4	slice(8)	Index 8 to the end	"16"
6	slice(-5)	The last five characters	"10-16"

Each slice skips the dashes at indexes 4 and 7 because no range includes them. `date` itself is unchanged: `slice` copies characters into a new string.

Now vary the numbers, to see what `slice` does at the edges. To make an empty string visible, the program puts square brackets around the result in a template literal; an **empty string** is a string with no characters at all, written `""`, with length 0.

slice-edges.ts

```

1 const word = "Hopper";
2 console.log(word.slice(0, 3));
3 console.log(word.slice(3));
4 console.log(word.slice(0, 3) + word.slice(3));
5 console.log(word.slice(-3));
6 console.log(`[${word.slice(2, 2)}]`);
7 console.log(`[${word.slice(4, 1)}]`);
8 console.log(word.slice(0, 100));

```

```

Hop
per
Hopper
per
[]
[]
Hopper

```

- Lines 2–4: `slice(0, 3)` and `slice(3)` meet at index 3, and joined they rebuild "Hopper".
- Line 5: `slice(-3)` is the last three characters.
- Line 6: start and end are the same, so no characters are taken: an empty string.

- Line 7: when `start` comes after `end`, `slice` gives an empty string too; it never runs backward.
- Line 8: an `end` past the last index just stops at the end of the string. `slice` never gives an error and never gives `undefined`.

KEY IDEA

`text.slice(start, end)` gives a new string from index `start` up to, but not including, `end`; without `end`, it runs to the end of the string, and negative indexes count from the end.
`end - start` is the number of characters taken.

WATCH OUT

Treating `end` as included, or as a length. Both mistakes give a wrong substring with no message at all:

month-wrong.ts

```
1 const date = "2026-10-16";
2 console.log(`${date.slice(5, 2)} `);
3 console.log(`${date.slice(5, 6)} `);
4 console.log(`${date.slice(5, 7)} `);
```

```
[]
[1]
[10]
```

The month is the two characters at indexes 5 and 6. Line 2 writes `slice(5, 2)` as if the second number were "how many characters," but it is a position, and 2 comes before 5, so the result is empty. Line 3 writes `slice(5, 6)` as if `end` were included, and gets only 1. Line 4 is right: `end` is the index just **after** the last character you want.

PAUSE AND TRY

Question 8 Tracing

Predict exactly what this program shows. Write out the indexes of `"notes.txt"` first.

predict-slice.ts

```
1 const fileName = "notes.txt";
2 console.log(fileName.slice(0, 5));
3 console.log(fileName.slice(6));
4 console.log(fileName.slice(-3));
5 console.log(fileName.slice(0, -4));
6 console.log(fileName.slice(2, 4));
```

3.6 Changing Case and Trimming

Text that people type is messy. One person types their email as `Ana.Lee@Example.COM`, another adds a stray space at the end. To a program, `"Ana"` and `"ana"` are different strings, and so are

"ana" and "ana ". Before using typed text, programs usually clean it up. Three string methods do the most common cleaning:

- `toUpperCase()` gives a copy of the string with every letter changed to a capital letter.
- `toLowerCase()` gives a copy with every letter changed to a small letter.
- `trim()` gives a copy with the **whitespace** removed from both ends. Whitespace means characters you cannot see: spaces, tabs, and newlines. Whitespace in the middle is kept.

None of them needs an argument, but the parentheses are still required: they are what makes the method run.

WORKED EXAMPLE Cleaning an email address

clean-email.ts

```
1 const typed = " Ana.Lee@Example.COM ";
2 const email = typed.trim().toLowerCase();
3 console.log(`[${typed}]`);
4 console.log(`[${email}]`);
5 console.log(typed.length, email.length);
```

```
[ Ana.Lee@Example.COM ]
[ana.lee@example.com]
23 19
```

Line 2 calls two methods in a row, which is called **chaining**. It works left to right: `typed.trim()` gives a new string without the four outside spaces, "Ana.Lee@Example.COM", and `.toLowerCase()` is then called on **that** string, giving "ana.lee@example.com". Each method gives back a string, so the next method has a string to work on. The brackets on lines 3 and 4 make the spaces visible. Line 5 confirms it: `typed` still has 23 characters, and the clean copy has 19.

Now vary the text, to see exactly what each method touches:

case-and-trim.ts

```
1 const title = " the Hobbit ";
2 console.log(`[${title.trim()}]`);
3 console.log(`[${title.toUpperCase()}]`);
4 console.log("Room 12b".toUpperCase());
5 console.log(`[${" two words ".trim()}]`);
6 console.log("Line\n".trim().length);
```

```
[the Hobbit]
[ THE HOBBIT ]
ROOM 12B
[two words]
4
```

`toUpperCase` changes only letters: the spaces stay on line 3, and the digits on line 4 are unchanged. Line 5 shows that `trim` keeps the two spaces between the words. On line 6, the newline at the end is whitespace too, so `"Line\n".trim()` is "Line", 4 characters.

KEY IDEA

`toUpperCase()` and `toLowerCase()` change the case of every letter, and `trim()` removes whitespace from both ends. Each gives back a new string, so methods can be chained: `typed.trim().toLowerCase()`.

WATCH OUT

Parentheses on a property, or none on a method. `length` is a property, so it never takes parentheses; `toUpperCase` is a method, so it always does. Mixing them up gives two different results:

missing-parens.ts

```
1 const title = "the hobbit";
2 console.log(title.toUpperCase);
3 console.log(title.toUpperCase());
```

```
[Function: toUpperCase]
THE HOBBIT
```

Without parentheses, line 2 does not run the method; it shows the method itself, which Node describes as `[Function: toUpperCase]`. No one reports an error, because a method is a value that may be shown. Line 3 adds the parentheses and gets the result. The opposite mistake is caught by the checker:

length-call.ts

```
1 const title = "the hobbit";
2 console.log(title.length());
```

```
length-call.ts:2:19 - error TS2349: This expression is not callable.
  Type 'Number' has no call signatures.
```

To **call** something means to run it with parentheses. The message points at `length` (column 19) and says it cannot be called: `title.length` is a number, and a number cannot be run. (The second line says the same in the checker's own terms.) Remove the parentheses, or delete the file, before your next check.

Name	Kind	Write it
<code>length</code>	Property	<code>title.length</code>
<code>toUpperCase</code>	Method	<code>title.toUpperCase()</code>
<code>at</code>	Method	<code>title.at(-1)</code>

PAUSE AND TRY**Question 9** Tracing

Predict exactly what this program shows. Notice the spaces inside `code`.

predict-case.ts

```
1 const code = " ab-12 ";
2 console.log(`[${code.toUpperCase()}]`);
3 console.log(code.trim().toUpperCase().length);
4 console.log(code.trim()[0].toUpperCase());
```

3.7 Strings Are Immutable

Suppose a program stores "hello" and you want "Jello". The natural idea is to replace the first character, the way you would correct a letter on paper:

change-letter.ts

```
1 let greeting = "hello";
2 greeting[0] = "J";
3 console.log(greeting);
```

```
change-letter.ts:2:1 - error TS2542: Index signature in type 'String' only permits reading.
```

The checker rejects line 2. An **index signature** is the checker's name for reaching parts of a value with square brackets, as in `greeting[0]`, and "only permits reading" means you may read `greeting[0]` but never assign to it. Notice that `greeting` is a `let` variable, so the variable may change. What may not change is the string. Fix or delete this file before your next check.

This is a rule of JavaScript itself: strings are **immutable**, which means a string can never be changed after it is made. No method and no assignment can change the characters of an existing string. Every tool you have met today (`slice`, `toUpperCase`, `trim`, and the rest) leaves the original alone and gives back a **new** string.

Why design strings this way? Because a string you have stored is then guaranteed to stay the same. If one part of a program could change the characters of a string, every other line using that string would see the change, which makes bugs very hard to trace. With immutable strings, a `const` name always means the same text.

So how do you get "Jello"? Build a new string from pieces of the old one, and store the new string in the variable:

change-letter-fixed.ts

```
1 let greeting = "hello";
2 greeting = "J" + greeting.slice(1);
3 console.log(greeting);
```

Jello

Line 2 works out the right side first, as Day 2 taught: `greeting.slice(1)` is "ello", and `"J" + "ello"` is a new string, "Jello". Then the assignment stores the new string in `greeting`. The variable changed, from one string to another; the string "hello" never did.

WORKED EXAMPLE A method gives back a new string

Because a method gives back a new string, the result must be stored or used, or it is lost.

shout-fixed.ts

```
1 const nickname = "ana";
2 const loud = nickname.toUpperCase();
3 console.log(nickname, loud);
```

ana ANA

Name	Holds	Where it came from
nickname	"ana"	Line 1, never changed
loud	"ANA"	The new string toUpperCase made

Both strings exist side by side: toUpperCase made a new one and left nickname as it was.

KEY IDEA

Strings are immutable: no assignment or method changes an existing string. Methods give back a new string, so store the result (`const loud = nickname.toUpperCase();`) or use it right away.

WATCH OUT

Calling a method and throwing the result away. This program means to make the nickname capital:

shout-forgot.ts

```
1 const nickname = "ana";
2 nickname.toUpperCase();
3 console.log(nickname);
```

ana

Line 2 runs `toUpperCase`, which makes the new string "ANA", but nothing stores or uses it, so it is thrown away. Line 3 shows the unchanged `nickname`. Neither `npx tsc` nor `node` reports anything. The fix is `shout-fixed.ts` above: store the result in a new constant. (Or, if `nickname` were a `let`, assign it back: `nickname = nickname.toUpperCase();`.)

PAUSE AND TRY**Question 10** Debugging

This program should show the title with a capital first letter, Matilda. `npx tsc` reports the message below. Explain the mistake and write the fix.

capitalize-title.ts

```
1 const title = "matilda";
2 title[0] = title[0].toUpperCase();
3 console.log(title);
```

capitalize-title.ts:2:1 - error TS2542: Index signature in type 'String' only permits reading.

3.8 Searching a String

Is there an @ in this email address? Where is the space in this full name? Does this file name start with IMG? Each is a question about what a string contains, and three methods answer them:

- `indexOf(text)` gives the index where `text` **first** appears in the string, or `-1` if it does not appear at all. `-1` is used because no character can be at index `-1` when counting from the start, so it cannot be mistaken for a real position.
- `includes(text)` gives `true` if `text` appears anywhere in the string, and `false` if not.
- `startsWith(text)` gives `true` if the string begins with `text`. (A matching method, `endsWith`, checks the end.)

`true` and `false` are Day 2's booleans. All three methods are **case-sensitive**, as names are in code: "A" and "a" are different characters, so searching for "Ana" does not find "ana".

WORKED EXAMPLE Splitting a full name at the space

`indexOf` and `slice` together pull a string apart at a character whose position you do not know in advance.

find-space.ts

```
1 const fullName = "Ada Lovelace";
2 const space = fullName.indexOf(" ");
3 const first = fullName.slice(0, space);
4 const last = fullName.slice(space + 1);
5 console.log(space);
6 console.log(first);
7 console.log(last);
```

```
3
Ada
Lovelace
```

Line	Expression	Worked out	Value
2	<code>fullName.indexOf(" ")</code>	A 0, d 1, a 2, then the space	3
3	<code>fullName.slice(0, space)</code>	<code>slice(0, 3)</code> : up to, not including, the space	"Ada"
4	<code>fullName.slice(space + 1)</code>	<code>slice(4)</code> : from just after the space	"Lovelace"

This works for any first and last name: with "Grace Hopper", space is 5, and the slices give "Grace" and "Hopper". The + 1 on line 4 skips the space itself.

Now vary what you search for:

search-checks.ts

```
1 const email = "ana.lee@example.com";
2 console.log(email.includes("@"));
3 console.log(email.indexOf("@"));
4 console.log(email.indexOf("."));
5 console.log(email.indexOf("#"));
6 console.log(email.startsWith("ana"));
7 console.log(email.startsWith("Ana"));
8 console.log(email.includes("Example"));
```

```
true
7
3
-1
true
false
false
```

`indexOf("@")` is 7: a n a . l e e fill indexes 0 to 6. `indexOf(".")` is 3, the **first** dot, even though there is another one later. `"#"` does not appear, so line 5 shows -1. Lines 7 and 8 show case-sensitivity: the address starts with `ana`, not `Ana`, and contains `example`, not `Example`.

Sometimes you want to ask whether two strings are exactly the same. The operator `===` does that: `a === b` gives `true` when `a` and `b` have exactly the same characters in the same order, and `false` otherwise. (This is a preview of Day 4, which teaches comparisons fully.) Case counts here too:

compare-strings.ts

```
1 const word = "Level";
2 const lower = word.toLowerCase();
3 console.log(lower === "level");
4 console.log(lower === "Level");
5 console.log(word.toUpperCase() === "LEVEL");
```

```
true
false
true
```

`lower` is `"level"`, so line 3 is `true` and line 4 is `false`: a capital `L` is a different character. Line 5 compares the capital copy, `"LEVEL"`, with `"LEVEL"`. Changing both strings to the same case before comparing is the usual way to ignore case.

KEY IDEA

`indexOf(text)` gives the index of the first match, or -1 if there is none. `includes(text)` and `startsWith(text)` give `true` or `false`. All of them are case-sensitive, and so is `===` (a preview of Day 4), which checks whether two strings are exactly the same.

WATCH OUT

Forgetting that `indexOf` can give -1. The full-name program assumes there is a space. Give it a one-word name:

no-space.ts

```
1 const fullName = "Cher";
2 const space = fullName.indexOf(" ");
3 console.log(space);
4 console.log(fullName.slice(0, space));
5 console.log(fullName.slice(space + 1));
```

```
-1
Che
Cher
```

There is no space, so `space` is -1. Line 4 becomes `slice(0, -1)`, which drops the last character, giving `Che`. Line 5 becomes `slice(0)`, the whole name. No error appears; the program quietly gives nonsense. Day 4 shows how to check for -1 and do something sensible instead. Until then, know that any program using `indexOf` should ask "what if it is not there?"

PAUSE AND TRY**Question 11** Tracing

Predict exactly what this program shows.

predict-search.ts

```
1 const song = "banana boat";
2 console.log(song.indexOf("a"));
3 console.log(song.indexOf("boat"));
4 console.log(song.indexOf("Boat"));
5 console.log(song.includes("nan"));
6 console.log(song.startsWith("boat"));
7 console.log(song.slice(song.indexOf(" ") + 1));
```

3.9 Replacing, Repeating, and Padding

Some changes are about the text inside a string: a phone number written with dashes that should use spaces, a word to swap for another. Others are about layout: a row of dashes under a heading, or numbers lined up in a column. Four methods handle these, and like every string method, each gives back a new string.

- `replace(old, new)` gives a copy with the **first** match of `old` replaced by `new`.
- `replaceAll(old, new)` gives a copy with **every** match replaced.
- `repeat(count)` gives the string repeated `count` times, as on Day 1.
- `padStart(width, fill)` adds copies of `fill` to the **start** until the string is `width` characters long. Leave out `fill`, and it pads with spaces. A string already `width` long or longer is unchanged.

WORKED EXAMPLE Rewriting a phone number

phone-format.ts

```

1 const phone = "555-123-4567";
2 console.log(phone.replace("-", " "));
3 console.log(phone.replaceAll("-", " "));
4 console.log(phone.replaceAll("-", ""));
5 console.log(phone.replace("123", "999"));
6 console.log(phone.replace("888", "999"));
7 console.log(phone);

```

```

555 123-4567
555 123 4567
5551234567
555-999-4567
555-123-4567
555-123-4567

```

Line	Call	What changes
2	<code>replace("-", " ")</code>	Only the first dash
3	<code>replaceAll("-", " ")</code>	Both dashes
4	<code>replaceAll("-", "")</code>	Both dashes, replaced by nothing: removed
5	<code>replace("123", "999")</code>	The match can be longer than one character
6	<code>replace("888", "999")</code>	No match, so nothing changes

Line 7 shows `phone` itself, unchanged: every result above was a new string. Like the search methods, `replace` and `replaceAll` are case-sensitive.

Now the layout tools. To line numbers up on their right edge, as on a receipt or a scoreboard, pad each one to the same width:

scoreboard.ts

```

1 const anaScore = 92;
2 const benScore = 7;
3 const kimScore = 100;
4 console.log("-".repeat(9));
5 console.log("Ana" + String(anaScore).padStart(6));
6 console.log("Ben" + String(benScore).padStart(6));
7 console.log("Kim" + String(kimScore).padStart(6));
8 console.log("-".repeat(9));
9 console.log(String(benScore).padStart(3, "0"));
10 console.log("Hopper".padStart(3));

```

```

-----
Ana    92
Ben     7
Kim   100
-----
007
Hopper

```

`padStart` is a string method, so each score is first turned into text with Day 2's `String()`. `String(92)` is "92", two characters, so `padStart(6)` adds four spaces in front; "7" gets five and "100" gets three. Every padded score is six characters wide, so the right edges line up. Line 10 pads with "0" instead of spaces, the usual way to make a code like 007. Line 11 shows that a string already longer than the width is left as it is; `padStart` never cuts a string short.

KEY IDEA

`replace` changes the first match and `replaceAll` changes every match. `repeat(count)` repeats a string, and `padStart(width, fill)` adds `fill` (spaces by default) at the start until the string is `width` characters long.

WATCH OUT

Padding a number. `padStart` belongs to strings, and a number does not have it:

pad-number.ts

```
1 const total = 4 * 2.5;
2 console.log(total.padStart(8));
```

pad-number.ts:2:19 - error TS2339: Property 'padStart' does not exist on type 'number'.

The checker knows `total` is a number and that numbers have no `padStart`, so it points at the name `padStart` (column 19). Here, "property" covers methods too: a method is a property whose value is a function. Convert first: `String(total).padStart(8)`. Fix or delete the file before your next check.

PAUSE AND TRY**Question 12** Tracing

Predict exactly what this program shows.

predict-replace.ts

```
1 const sign = "Cats and cats and CATS";
2 console.log(sign.replace("cats", "dogs"));
3 console.log(sign.replaceAll("cats", "dogs"));
4 console.log(sign.toLowerCase().replaceAll("cats", "dogs"));
5 console.log("ab".repeat(3));
6 console.log("42".padStart(5, "*"));
```

3.10 Splitting a String

A sentence is made of words separated by spaces. A line from a spreadsheet file is made of values separated by commas, like pens,4,1.25. To work with each piece, you need to cut the string apart at its separators.

The method `split(separator)` does that: it cuts the string at every place `separator` appears, drops the separators, and gives back the pieces. Splitting with the empty string `""` cuts between every character.

The pieces come back in a new kind of value: an **array**, one value that holds several values in order. Node shows an array in square brackets, with each string in single quotes, like `[ 'Ada', 'Lovelace' ]`. The checker's type for it is `string[]`, read "an array of strings." Arrays get a full day of their own, Day 6; today you only need to read one and use two things with it.

WORKED EXAMPLE Cutting strings into pieces

split-parts.ts

```
1 const fullName = "Ada King Lovelace";
2 const csvLine = "pens,4,1.25";
3 console.log(fullName.split(" "));
4 console.log(csvLine.split(","));
5 console.log("abc".split(""));
6 console.log(fullName.split(" ").length);
```

```
[ 'Ada', 'King', 'Lovelace' ]
[ 'pens', '4', '1.25' ]
[ 'a', 'b', 'c' ]
3
```

Line 3 cuts at each of the two spaces, giving three names. Line 4 cuts at each comma; notice that `'4'` and `'1.25'` are in quotes, so they are still strings, and you would need `Number()` to do arithmetic with them. Line 5 splits into single characters. Line 6 shows that an array has a `length` too, the number of pieces: here, three words.

Now vary it by putting the pieces back together. This is a preview of Day 6: an array has a `reverse()` method, which turns the order of its pieces around, and a `join(separator)` method, which glues the pieces into one string with `separator` between each pair. Chained after `split("")`, they reverse a string:

reverse-word.ts

```
1 const word = "stressed";
2 const reversed = word.split("").reverse().join("");
3 console.log(reversed);
4 console.log("2026-10-16".split("-").join("/"));
5 console.log("one two three".split(" ").join(""));
```

```
desserts
2026/10/16
onetwothree
```

Step on line 2	Value
<code>word</code>	<code>"stressed"</code>
<code>.split("")</code>	<code>['s', 't', 'r', 'e', 's', 's', 'e', 'd']</code>
<code>.reverse()</code>	<code>['d', 'e', 's', 's', 'e', 'r', 't', 's']</code>
<code>.join("")</code>	<code>"desserts"</code>

Joining with `""` puts nothing between the pieces. Line 4 splits a date at its dashes and joins the pieces with slashes. Line 5 splits at the spaces and joins with nothing, which removes the spaces.

KEY IDEA

`text.split(separator)` cuts a string at every `separator` and gives back an array of the pieces; `split("")` gives every character. As a preview of Day 6, `pieces.join(separator)` glues an array back into a string, and `text.split("").reverse().join("")` reverses a string.

WATCH OUT

Two separators in a row. If a sentence has a double space, splitting at `" "` finds an empty piece between the two spaces:

double-space.ts

```
1 const sentence = "Strings are fun";
2 console.log(sentence.split(" "));
3 console.log(sentence.split(" ").length);
```

```
[ 'Strings', '', 'are', 'fun' ]
4
```

The sentence has three words, but the array has four pieces, one of them the empty string `''`. A word count based on `split(" ")` is only right when the words are separated by exactly one space. `trim()` does not help here, because it only removes whitespace at the ends.

PAUSE AND TRY

Question 13 Tracing

Predict exactly what this program shows.

predict-split.ts

```
1 const tags = "red,green,,blue";
2 console.log(tags.split(","));
3 console.log(tags.split(",").length);
4 console.log("level".split("").reverse().join(""));
5 console.log("a-b-c".split("-").join(" + "));
```

3.11 Formatting Numbers with `toFixed`

Day 2 rounded money with `Math.round(x * 100) / 100`. That gives the right number, but not always the right look: a price of 4.5 shows as 4.5, and a receipt should show 4.50. Rounding gives a number; what you need for display is **text** with exactly two decimal places.

Numbers have a method for exactly this. `toFixed(digits)` gives a **string** showing the number rounded to `digits` places after the decimal point, adding zeros if needed. `toFixed(0)` rounds to a

whole number. You call it on a number the same way you call a string method on a string:
`price.toFixed(2)`.

WORKED EXAMPLE Prices with two decimal places

price-format.ts

```
1 const price = 4.5;
2 const share = 10 / 3;
3 const total = 0.1 + 0.2;
4 console.log(price.toFixed(2));
5 console.log(share.toFixed(2));
6 console.log(total.toFixed(2));
7 console.log(share.toFixed(0));
8 console.log(typeof price.toFixed(2));
9 console.log(`Price: ${price.toFixed(2)}`);
```

```
4.50
3.33
0.30
3
string
Price: $4.50
```

Line	Number	Call	Result
4	4.5	toFixed(2)	"4.50"
5	3.3333333333333335	toFixed(2)	"3.33"
6	0.30000000000000004	toFixed(2)	"0.30"
7	3.3333333333333335	toFixed(0)	"3"

Line 4 adds a zero. Lines 5 and 6 round away Day 2's floating-point leftovers. Line 8 confirms the result is a string; here `typeof` applies to the whole call `price.toFixed(2)`, because a value followed by a dot and a method call counts as one value. Line 9 puts a formatted price in a template literal. The `$` before the placeholder is shown as an ordinary dollar sign; only `$` followed by `{` starts a placeholder.

Now vary the numbers: try values that sit exactly halfway on paper.

fixed-rounding.ts

```
1 const fee = 1.005;
2 const tax = 2.345;
3 console.log(fee.toFixed(2));
4 console.log(tax.toFixed(2));
5 console.log(fee.toFixed(20));
```

```
1.00
2.35
1.00499999999999989342
```

On paper, 1.005 rounds up to 1.01, but line 3 shows `1.00`. Day 2 explains why: 1.005 cannot be stored exactly in binary, and line 5 asks for 20 decimal places to reveal the value actually stored,

which is a tiny bit **less** than 1.005. `toFixed` rounds the stored value correctly, so it rounds down. 2.345 happens to be stored a tiny bit above, so it rounds up. For everyday prices this almost never matters, but it explains a result that looks like a rounding mistake.

KEY IDEA

`number.toFixed(digits)` gives a string showing the number with exactly `digits` decimal places, rounded and padded with zeros. Use it last, only when showing a number.

WATCH OUT

Doing math with the text from `toFixed`. The result is a string, so `+` joins it and the other operators are rejected:

`fixed-too-early.ts`

```
1 const price = 4.5;
2 const priceText = price.toFixed(2);
3 console.log(priceText + 1);
```

4.501

`priceText` is "4.50", so `priceText + 1` joins the text and the 1, as in 3.2. With `*` instead, the checker steps in:

`fixed-times.ts`

```
1 const price = 4.5;
2 const priceText = price.toFixed(2);
3 console.log(priceText * 2);
```

`fixed-times.ts:3:13 - error TS2362: The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.`

The rule from Day 2 applies: calculate with full numbers, and format only the final value you show. Fix or delete `fixed-times.ts`, then do the arithmetic first and format the result:

`fixed-too-early-fixed.ts`

```
1 const price = 4.5;
2 console.log((price + 1).toFixed(2));
3 console.log((price * 2).toFixed(2));
```

5.50
9.00

The parentheses make the arithmetic happen first, on numbers; `toFixed` then formats each result.

PAUSE AND TRY**Question 14** Tracing

Predict exactly what this program shows.

predict-fixed.ts

```
1 const cost = 7;
2 const rate = 0.0825;
3 const tax = cost * rate;
4 console.log(cost.toFixed(2));
5 console.log(tax.toFixed(2));
6 console.log((cost + tax).toFixed(2));
7 console.log(`${cost.toFixed(1)}`);
```

Putting it together

A library makes a member card from the name a person typed on a sign-up sheet. The typed name is messy, with stray spaces and odd capitals, so the program cleans it, takes it apart, and builds three new strings from the pieces. Before you type it, make sure `npx tsc` is clean in your `day03` folder: fix or delete any broken file from today.

member-card.ts

```
1 const typedName = " grace HOPPER ";
2 const memberNumber = 42;
3 const yearlyFee = 12.5;
4
5 const cleanName = typedName.trim().toLowerCase();
6 const space = cleanName.indexOf(" ");
7 const first = cleanName.slice(0, space);
8 const last = cleanName.slice(space + 1);
9
10 const firstCapital = first[0].toUpperCase() + first.slice(1);
11 const lastCapital = last[0].toUpperCase() + last.slice(1);
12 const username = first[0] + last + String(memberNumber);
13 const cardNumber = String(memberNumber).padStart(6, "0");
14
15 console.log(`Member: ${firstCapital} ${lastCapital}`);
16 console.log(`Username: ${username}`);
17 console.log(`Card: ${cardNumber}`);
18 console.log(`Fee: ${yearlyFee.toFixed(2)}`);
```

`npx tsc` reports no errors, and `node member-card.ts` shows:

```
Member: Grace Hopper
Username: ghopper42
Card: 000042
Fee: $12.50
```

Follow the values in the order the lines run:

Name (line)	Worked out	Value
cleanName (5)	trim(), then toLowerCase() (3.6)	"grace hopper"
space (6)	indexOf(" "): grace fills indexes 0–4 (3.8)	5
first (7)	slice(0, 5) (3.5)	"grace"
last (8)	slice(6)	"hopper"
firstCapital (10)	"G" + "race": index 0 capitalized, plus the rest (3.4, 3.7)	"Grace"
lastCapital (11)	"H" + "opper"	"Hopper"
username (12)	"g" + "hopper" + "42" (3.2)	"ghopper42"
cardNumber (13)	"42".padStart(6, "0") (3.9)	"000042"

Lines 15–18 are one multi-line template literal (3.3), so one `console.log` shows four lines. Its placeholders put in the names, the username, and the card number, and `yearlyFee.toFixed(2)` turns 12.5 into "12.50" (3.11), after an ordinary \$. Every name is a `const`: the cleaning never changes `typedName`; each step makes a new string (3.7). Try another typed name, such as "ADA Lovelace", and only the values change.

One thing the program assumes: the name contains a space. With a one-word name, `space` is -1, and the slices go wrong as in 3.8. Day 4 gives you the tools to check.

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
TS1005: ',' expected. then TS1002: Unterminated string literal.	An apostrophe inside single quotes	Use double quotes, or <code>\'</code>
TS2362: The left-hand side of an arithmetic operation must be ...	<code>-</code> , <code>*</code> , or <code>/</code> on a string, such as <code>toFixed</code> 's result (TS2363 when the string is on the right)	Convert with <code>Number()</code> , or format last
TS2552: Cannot find name 'totl'. Did you mean 'total'?	A typo inside a <code>\${...}</code> placeholder	Fix the spelling
TS18048: 'last' is possibly 'undefined'.	Using the result of <code>at()</code> as a string	Use <code>text[i]</code> (Day 9 shows another way)
TS2349: This expression is not callable.	<code>length()</code> with parentheses	<code>length</code> is a property: no parentheses
TS2542: Index signature in type 'String' only permits reading.	Assigning to <code>text[i]</code>	Build a new string with <code>slice</code> and <code>+</code>
TS2339: Property 'padStart' does not exist on type 'number'.	A string method called on a number	<code>String(value).padStart(...)</code>

TypeError: Cannot read properties of undefined (reading 'length')	An index past the end, such as <code>text[text.length]</code> (from node)	The last index is <code>length - 1</code>
51 instead of 6, or Total: 23	+ with a string joins	Number(), or parentheses around the sum
<code>\${total}</code> shown in the output	A placeholder in ordinary quotes	Use backticks
[Function: toUpperCase] shown	A method without parentheses	Add ()
The string did not change	A method's result was not stored	<code>const result = text.method();</code>
A wrong piece, or an empty string, from slice	end treated as included or as a length; <code>indexOf</code> gave -1	end is the index after the last character

Vocabulary

Term	Definition	Example
escape character	A backslash and a character that together stand for one special character	<code>\n, \"</code>
newline / tab	The character that starts a new line / the one that jumps to the next column	<code>"a\nb", "a\tb"</code>
concatenation	Joining strings end to end with +	<code>"Hot" + "dog" is "Hotdog"</code>
template literal	A string in backticks that can hold placeholders and span lines	<code>`Total: \${total}`</code>
placeholder	<code>\${expression}</code> in a template literal, replaced by the expression's value	<code>\${quantity * price}</code>
multi-line string	A string that spans several lines, keeping its line breaks	A template literal over three lines
index	A character's position in a string, counting from 0	<code>"Hopper"[0] is "H"</code>
indexing	Reading one character with square brackets	<code>word[3]</code>
property	A value that belongs to another value, read with a dot and no parentheses	<code>word.length</code>
method / call	A function that belongs to a value / running it with parentheses	<code>word.at(-1), name.trim()</code>
off-by-one error	Being one away from the right number, such as using <code>length</code> as the last index	<code>word[word.length]</code>
substring	A part of a string	<code>"Hop" in "Hopper"</code>
empty string	A string with no characters, length 0	<code>""</code>
whitespace	Characters you cannot see: spaces, tabs, newlines	What <code>trim()</code> removes from the ends
chaining	Calling one method on the result of another	<code>typed.trim().toLowerCase()</code>

immutable	Cannot be changed after it is made; strings are immutable	<code>greeting[0] = "J"</code> is rejected
index signature	The checker's name for reaching parts of a value with square brackets	<code>text[i]</code>
case-sensitive	Treating capital and small letters as different	"Ana" is not "ana"
array	One value that holds several values in order (Day 6)	<code>['Ada', 'LoveLace']</code>

Summary

- A string goes in double or single quotes, and **escape characters** (`\`, `'`, `\\`, `\n`, `\t`) put in characters that quotes cannot. A **template literal** in backticks can span lines and holds `${expression}` **placeholders**.
- `+` with a string **concatenates**: if either side is a string, the other is turned into text, so `"5" + 1` is `"51"`. Each `+` is applied left to right, so put a calculation inside a message in parentheses. Other arithmetic on a string is a type error.
- Indexes start at 0. `length` counts the characters, `text[i]` and `text.at(i)` read one (with `at(-1)` the last), and `slice(start, end)` takes a **substring** up to, but not including, `end`. A position past the end gives `undefined`.
- Strings are **immutable**: every method gives back a new string, so store or use the result. `toUpperCase`, `toLowerCase`, `trim`, `replace`, `replaceAll`, `repeat`, and `padStart` change text and layout; `indexOf` (`-1` when not found), `includes`, and `startsWith` search it; `split` cuts it into an **array**.
- `number.toFixed(digits)` turns a number into a string with exactly that many decimal places. Calculate with numbers first, and format only what you show.

Practice

Do these in your `day03` folder. First make sure `npm tsc` prints nothing there: if any broken file from today is still in the folder (such as `apostrophe-broken.ts`, `change-letter.ts`, or the question's `capitalize-title.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 to test your program.

Question 15 Short answer

Initials. Store a full name, a first and a last name separated by one space, in a constant, such as `"ada loveLace"`. Show the initials as capital letters followed by dots: `Initials: A.L.`

Question 16 Short answer

Reverse and palindrome. A **palindrome** is a word that reads the same forwards and backwards, like `"level"`. Store a word in a constant, show it reversed using `split`, `reverse`, and `join`, then show whether it is a palindrome as `true` or `false`. Make `"Racecar"` count as a palindrome even with its capital R.

Question 17 Short answer

Receipt line. Store an item name and a price in constants, such as "Notebook" and 3.5. Show one receipt line exactly 20 characters wide: the item name on the left, and the price on the right with a dollar sign and two decimal places. Underline it with 20 dashes.

Review questions

Question 18 Tracing

Predict exactly what this program shows. Track `space` and each new string.

review-trace.ts

```
1 const city = "san francisco";
2 const space = city.indexOf(" ");
3 const short = city[0].toUpperCase() + city[space + 1].toUpperCase();
4 console.log(short);
5 console.log(`${city.slice(space + 1)} has ${city.length - space - 1} letters`);
6 console.log(city.replace("san", "SAN").at(-1));
```

Question 19 Multiple choice

What does `console.log("3" + 4 * 2);` show?

- A. 11
- B. 14
- C. 38
- D. 342

Question 20 Debugging

This program should show `Qty:` followed by the quantity right-aligned in 4 characters. `npx tsc` reports an error. Predict the message, explain it, and write the fix.

quantity-pad.ts

```
1 const quantity = 3 + 4;
2 console.log("Qty:" + quantity.padStart(4));
```

Question 21 Tracing

Predict exactly what this program shows.

email-domain.ts

```
1 const email = "Kim.Park@School.example";
2 const at = email.indexOf("@");
3 const user = email.slice(0, at).toLowerCase();
4 const domain = email.slice(at + 1).toLowerCase();
5 console.log(user.replaceAll(".", " "));
6 console.log(domain);
7 console.log(email.length - domain.length);
```

Question 22 Short answer

Ravi's program shows `B14 11`. He expected `B14 2`: the room code, then the floor plus one. `npx tsc` and `node` reported nothing. Explain each value, and write a line that shows the floor plus one correctly.

code-label.ts

```
1 const roomCode = "B" + 2 * 7;
2 const floor = roomCode.slice(1, 2);
3 console.log(roomCode, floor + 1);
```

Question 23 Multiple choice

What does this program show?

immutable-trace.ts

```
1 let pet = "cat";
2 pet.toUpperCase();
3 pet = pet.replace("c", "b");
4 pet.repeat(2);
5 console.log(pet);
```

- A. cat
- B. bat
- C. BAT
- D. batbat

Question 24 Tracing

Predict the four lengths this program shows.

escape-length.ts

```
1 console.log("a\tb".length);
2 console.log("a\\tb".length);
3 console.log(`a
4 b`.length);
5 console.log("".length);
```

Go further

MDN's guide explains strings and template literals in more depth, and its reference lists every string method, including some not covered today, such as `endsWith` and `padEnd`: MDN: Text formatting and MDN: String (see the instance methods list).

Reflection

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

- ☐ I can write a string that contains quotes, new lines, and tabs. Reread: 3.1 Quotes and Escape Characters
- ☐ I can predict what `+` does when one side is a string, and explain why `"5" + 1` is `"51"`.

- ☐ I can build a message with a template literal, including one that spans lines. Reread: 3.3 Template Literals
- ☐ I can read any character with an index or at(), and find a string's length. Reread: 3.4 Length and Indexing
- ☐ I can take a substring with slice and say exactly which characters it includes. Reread: 3.5 Slicing
- ☐ I can clean up text with trim, toUpperCase, and toLowerCase, and chain methods. Reread: 3.6 Changing Case and Trimming
- ☐ I can explain what immutable means and store a method's result. Reread: 3.7 Strings Are Immutable
- ☐ I can search a string with indexOf, includes, and startsWith. Reread: 3.8 Searching a String
- ☐ I can use replace, replaceAll, repeat, and padStart. Reread: 3.9 Replacing, Repeating, and Padding
- ☐ I can split a string into pieces and reverse a string. Reread: 3.10 Splitting a String
- ☐ I can format a number with toFixed and explain why the result is a string. Reread: 3.11 Formatting Numbers with toFixed

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Tea
Coffee
A\nB
Say "yes"
Don't stop
```

Five lines from four `console.log` calls:

- Line 1: `\n` is a newline, so `Tea` and `Coffee` go on separate lines.
- Line 2: `\\` is one backslash, and the `n` after it is an ordinary letter. The output is the three characters `A\n` and then `B`, all on one line.
- Line 3: the double quotes are inside single quotes, so they are ordinary characters.
- Line 4: `\'` is an apostrophe (not needed inside double quotes, but allowed), and `\t` jumps to the next column before `stop`.

Reread 3.1 *Quotes and Escape Characters* if your output differed.

Question 2

C. The text contains both kinds of quotes. In double quotes, the apostrophe is fine, and each inner double quote is escaped as `\`.

- **A** is wrong: the apostrophe ends the single-quoted string after `Don`, a syntax error.
- **B** is wrong: the second `"` ends the string after `say`, a syntax error.
- **D** is wrong: `\\` is an escaped backslash, so the next `"` is not escaped and ends the string, again a syntax error.

Reread 3.1 *Quotes and Escape Characters* if you chose differently.

Question 3

```
Fruit: 46
10 pieces of fruit
Fruit: 10
22
```

- Line 3: `"Fruit: " + 4` is `"Fruit: 4"` (a string), then `+ 6` joins again: **Fruit: 46**.
- Line 4: `4 + 6` is two numbers, so it adds to 10 first; then `10 + " 10 pieces of fruit"` joins: **10 pieces of fruit**.
- Line 5: the parentheses add first, then join: **Fruit: 10**.
- Line 6: both are strings, so `+` joins them: **22**, not 4.

Reread 3.2 *Joining Strings with +* if any line differed.

Question 4

B. The parentheses make `points + 5` a sum of two numbers, 15, which is then joined to the label.

- **A** is wrong: left to right, `"Score: " + 10` is already a string, so `+` joins and shows `Score: 105`.

- **C** is wrong: everything is inside the quotes, so it shows the text `Score: points + 5`.
- **D** is wrong: `-` needs numbers, so `npx tsc` rejects it with TS2362.

Reread 3.2 *Joining Strings with +* if you chose differently.

Question 5

```
3 x 4
Area: 12
3 + 4 = 7
Perimeter: ${2 * (width + height)}
```

- Line 3: each placeholder becomes its value, and the `x` between them is copied.
- Line 4: the placeholder is worked out first, `3 × 4 = 12`.
- Line 5: the `+` outside the placeholders is just a character; only `${width + height}` adds, giving 7.
- Line 6: double quotes, not backticks, so nothing is worked out and the text is shown exactly.

Reread 3.3 *Template Literals* if any line differed.

Question 6

```
10
,
r
P
undefined
```

Character	L	i	m	a	,	(space)	P	e	r	u
Index	0	1	2	3	4	5	6	7	8	9

- `city.length` counts the comma and the space too: **10**.
- `city[4]` is the comma.
- `city[city.length - 2]` is `city[8]`, the **r**.
- `city.at(-4)` counts back four from the end: **u, r, e, P**.
- `city[10]` is one past the last index, 9, so it is **undefined**.

Reread 3.4 *Length and Indexing* if any line differed.

Question 7

C. Indexes start at 0, so the last index is one less than the length.

- **A** is wrong: index `length` is one past the end, so it gives `undefined` (an off-by-one error).
- **B** is wrong: square brackets do not count from the end; `city[-1]` is `undefined`. Only `city.at(-1)` counts back.
- **D** is wrong: `at(1)` is the **second** character, counted from the start.

Reread 3.4 *Length and Indexing* if you chose differently.

Question 8

```
notes
txt
txt
notes
te
```

Character	n	o	t	e	s	.	t	x	t
Index	0	1	2	3	4	5	6	7	8

- `slice(0, 5)`: indexes 0 to 4, **notes**.
- `slice(6)`: index 6 to the end, **txt**.
- `slice(-3)`: the last three, **txt** again.
- `slice(0, -4)`: -4 is index 5, the dot, so it stops just before the dot: **notes**. This drops the last four characters, whatever the name's length.
- `slice(2, 4)`: indexes 2 and 3, **te**.

Reread 3.5 *Slicing* if any line differed.

Question 9

```
[ AB-12 ]
5
A
```

- Line 2: `toUpperCase` changes a and b and leaves the spaces, the dash, and the digits alone; there is no trimming, so all three spaces stay.
- Line 3: the chain runs left to right: `trim()` gives "ab-12", `toUpperCase()` gives "AB-12", and its length is 5.
- Line 4: `code.trim()[0]` is the first character of the trimmed string, "a", and `toUpperCase()` makes it **A**. Without the trim, `code[0]` would be a space.

Reread 3.6 *Changing Case and Trimming* if any line differed.

Question 10

Line 2 tries to change the first character of the string, but strings are immutable: `title[0]` can be read, never assigned. (The right side, `title[0].toUpperCase()`, is fine; it reads the "m" and makes a new "M".) Build a new string instead: the capital first letter joined to the rest of the title from index 1.

capitalize-title-fixed.ts

```
1 const title = "matilda";
2 const capitalized = title[0].toUpperCase() + title.slice(1);
3 console.log(capitalized);
```

```
Matilda
```

`title[0].toUpperCase()` is "M" and `title.slice(1)` is "atilda", so `capitalized` is the new string "Matilda". `title` still holds "matilda". If you typed the original program, fix or delete it before you go on.

Reread 3.7 *Strings Are Immutable* if you missed it.

Question 11

```
1
7
-1
true
false
boat
```

- `indexOf("a")` finds the **first** a: b is index 0, a is index 1.
- "banana" has six characters (0 to 5) and the space is 6, so `boat` starts at **7**.
- "Boat" with a capital B does not appear: **-1**.
- "nan" appears inside banana: **true**.
- The string starts with banana, not boat: **false**.
- `indexOf(" ")` is 6, so `slice(7)` gives everything after the space: **boat**.

Reread 3.8 *Searching a String* if any line differed.

Question 12

```
Cats and dogs and CATS
Cats and dogs and CATS
dogs and dogs and dogs
ababab
***42
```

- Line 2: `replace` is case-sensitive, so the only match for "cats" is the middle word.
- Line 3: `replaceAll` changes every "cats", and there is still only one in small letters.
- Line 4: `toLowerCase()` runs first, giving "cats and cats and cats"; now `replaceAll` finds all three.
- Line 5: "ab" three times.
- Line 6: "42" is two characters, so three * are added to reach five.

Reread 3.9 *Replacing, Repeating, and Padding* if any line differed.

Question 13

```
[ 'red', 'green', '', 'blue' ]
4
level
a + b + c
```

- Line 2: there is nothing between the two commas, so one piece is the empty string.
- Line 3: four pieces, the empty one included.
- Line 4: "level" reversed is "level" again; it reads the same both ways.
- Line 5: the pieces a, b, c are joined with " + " between each pair; the + here is just a character in the separator.

Reread 3.10 *Splitting a String* if any line differed.

Question 14

```
7.00  
0.58  
7.58  
$7.0
```

- Line 4: 7 with two zeros added: **7.00**.
- Line 5: `tax` is $7 \times 0.0825 = 0.5775$, which rounds to **0.58**.
- Line 6: the parentheses add first, 7.5775, and then `toFixed(2)` rounds to **7.58**.
- Line 7: one decimal place, with the dollar sign as ordinary text before the placeholder: **\$7.0**.

Reread 3.11 *Formatting Numbers with toFixed* if any line differed.

Question 15

One possible solution:

initials.ts

```
1 const fullName = "ada loveLace";  
2 const space = fullName.indexOf(" ");  
3 const firstInitial = fullName[0].toUpperCase();  
4 const lastInitial = fullName[space + 1].toUpperCase();  
5 console.log(`Initials: ${firstInitial}.${lastInitial}.`);
```

```
Initials: A.L.
```

The first initial is always at index 0. The last name starts just after the space, so `indexOf(" ")` finds the space (index 3 here) and `fullName[space + 1]` is the first letter of the last name, index 4.

`toUpperCase()` capitalizes each initial even if the name was typed in small letters, and the template literal adds the dots. Try "grace hopper": it shows `Initials: G.H..`

Reread 3.4 *Length and Indexing* and 3.8 *Searching a String* if your result differed.

Question 16

One possible solution:

palindrome.ts

```
1 const word = "Racecar";  
2 const lower = word.toLowerCase();  
3 const reversed = lower.split("").reverse().join("");  
4 console.log(`Reversed: ${reversed}`);  
5 console.log(`Palindrome: ${lower === reversed}`);
```

```
Reversed: racecar  
Palindrome: true
```

`toLowerCase()` runs first, so the capital R cannot spoil the match. `split("")` cuts "racecar" into its characters, `reverse()` turns their order around, and `join("")` glues them back with nothing between. `lower === reversed` gives `true` because the two strings have the same characters in the same order. Without the `toLowerCase()`, the reverse would be "racecaR", and "Racecar" === "racecaR" is `false`. Try "Hopper": it shows `reppoh` and `false`.

Reread 3.10 *Splitting a String* and 3.8 *Searching a String* if your result differed.

Question 17

One possible solution:

receipt-line.ts

```
1 const item = "Notebook";
2 const price = 3.5;
3 const lineWidth = 20;
4 const priceText = "$" + price.toFixed(2);
5 const line = item + priceText.padStart(lineWidth - item.length);
6 console.log(line);
7 console.log("-".repeat(lineWidth));
```

```
Notebook      $3.50
-----
```

`price.toFixed(2)` gives "3.50", and joining "\$" in front gives "\$3.50". The item takes `item.length` characters, 8, so the price must fill the other $20 - 8 = 12$: `padStart(12)` adds 7 spaces in front of the 5 characters of "\$3.50". The whole line is $8 + 12 = 20$ characters, the same as the dashes. Because the width is worked out from `item.length`, a different item still makes a 20-character line: with "Pens" and 12, the line is Pens followed by 10 spaces and \$12.00, and its price lines up under the one above.

Reread 3.9 *Replacing, Repeating, and Padding* and 3.11 *Formatting Numbers with toFixed* if your result differed.

Question 18

```
SF
francisco has 9 letters
o
```

Expression	Value
<code>city.indexOf(" ")</code>	3 (san fills 0–2)
<code>city[0].toUpperCase() + city[4].toUpperCase()</code>	"S" + "F", which is "SF"
<code>city.slice(4)</code>	"francisco"
<code>city.length - space - 1</code>	$13 - 3 - 1$, which is 9
<code>city.replace("san", "SAN")</code>	"SAN francisco"
<code>.at(-1)</code>	"o"

`city.length` is 13: 3 letters, a space, and 9 letters. Subtracting the 3 letters before the space and the space itself leaves the 9 letters after it. On the last line, the replacement changes only the start, so the last character is still `o`, and `city` itself is unchanged.

Reread 3.8 *Searching a String* and 3.4 *Length and Indexing* if your output differed.

Question 19

C. Precedence comes first: `*` is applied before `+`, so $4 * 2$ is 8, two numbers multiplied. Then `"3" + 8` has a string on one side, so it joins: "38".

- **A** is wrong: it treats `"3"` as the number 3. With quotes, it is text, so `+` joins instead of adding.
- **B** is wrong: it works left to right, ignoring precedence: `"3" + 4` would be `"34"`, and `"34" * 2` would not even type-check.
- **D** is wrong: it joins all three values, as if `*` were a join too. `*` always multiplies numbers.

Reread 3.2 *Joining Strings with +* if you chose differently.

Question 20

```
quantity-pad.ts:2:31 - error TS2339: Property 'padStart' does not exist on type 'number'.
```

`3 + 4` is two numbers added, so `quantity` is the number 7, and `padStart` is a string method; column 31 is where `padStart` starts. Turn the number into text first:

quantity-pad-fixed.ts

```
1 const quantity = 3 + 4;
2 console.log("Qty:" + String(quantity).padStart(4));
```

```
Qty:    7
```

`String(quantity)` is `"7"`, and `padStart(4)` adds three spaces. If you typed the original program, fix or delete it before you go on.

Reread 3.9 *Replacing, Repeating, and Padding* if you missed it.

Question 21

```
kim park
school.example
9
```

Name	Worked out	Value
at	<code>indexOf("@")</code> : Kim.Park fills indexes 0–7	8
user	<code>slice(0, 8)</code> , then <code>toLowerCase()</code>	"kim.park"
domain	<code>slice(9)</code> , then <code>toLowerCase()</code>	"school.example"

Line 5 replaces every dot in `user` with a space; there is only one. Line 7: `email` has 23 characters and `domain` has 14, so the difference is 9, the 8 characters before the `@` plus the `@` itself.

Reread 3.8 *Searching a String* and 3.9 *Replacing, Repeating, and Padding* if your output differed.

Question 22

Model answer: On line 1, `*` comes first, so `2 * 7` is 14, and then `"B" + 14` joins to `"B14"`. On line 2, `slice(1, 2)` takes index 1 only, giving `"1"`, a **string**. On line 3, `floor + 1` has a string on one side, so `+` joins: `"1" + 1` is `"11"`. Both lines are correct code to the checker, because `+` with a string is allowed, and Node reports nothing because joining is not an error. Convert first:

code-label-fixed.ts

```
1 const roomCode = "B" + 2 * 7;
2 const floor = roomCode.slice(1, 2);
3 console.log(roomCode, Number(floor) + 1);
```

B14 2

With `Number(floor)` on line 3, the `+` sees two numbers, 1 and 1, and adds them.

A complete answer must:

- explain that `"B" + 14` joins after `2 * 7` is worked out,
- say that `slice` gives a string, so `floor + 1` joins to `"11"`, and
- convert with `Number()` before adding.

Reread 3.2 *Joining Strings with +* and 3.5 *Slicing* if your reasoning differed.

Question 23

B. Strings are immutable, so a method's result matters only if it is stored or used. Line 2's `"CAT"` and line 4's `"batbat"` are made and thrown away. Only line 3 assigns its result back to `pet`, so `pet` becomes `"bat"`.

- **A** is wrong: line 3 does store a new string in `pet`.
- **C** is wrong: it assumes line 2 changed `pet`, but the capital copy was never stored.
- **D** is wrong: it assumes line 4 changed `pet`, but the repeated copy was never stored.

Reread 3.7 *Strings Are Immutable* if you chose differently.

Question 24

```
3
4
3
0
```

- `"a\tb"`: a, one tab character, b: **3**.
- `"a\\tb"`: a, one backslash, then the ordinary letters `t` and `b`: **4**.
- The template literal holds `a`, the newline where the line breaks, and `b`: **3**.
- The empty string has no characters: **0**.

An escape is two characters in the code but one in the string; a line break inside backticks is one newline character.

Reread 3.1 *Quotes and Escape Characters*, 3.3 *Template Literals*, and 3.4 *Length and Indexing* if any length differed.