

Day 1 Setup and your first program

Self-Study Guide · About 65 minutes · With answer key

By the end you will be able to

- Install Node, check its version from the terminal, and set up a TypeScript project folder.
- Create a .ts file, type-check it with `npx tsc`, and run it with `node`.
- Use `console.log`, comments, and top-to-bottom order to write a short program.
- Tell a type error from a runtime error and explain what each message is pointing at.

What to remember from earlier

This is the first day, so you need no programming experience. You only need everyday computer skills: saving a file into a folder, finding that folder again, and typing exactly what you see. If you have ever followed instructions like "type this into the box and press Enter," you are ready.

1.1 Installing Node and Opening a Terminal

A computer does exactly what it is told, one instruction at a time, and nothing more. A **program** is a list of instructions written for a computer, and a **programming language** is a language with strict rules for writing those instructions.

This course teaches **TypeScript**. It is built on top of **JavaScript**, the programming language that every web browser understands and that runs a huge share of the world's websites and apps. TypeScript is JavaScript plus **types**: labels that say what kind of value something is, such as a number or a piece of text. A tool called the type checker reads those labels and warns you about mistakes before your program ever runs. You will meet it in 1.7. Everything you learn about how JavaScript runs also holds for TypeScript, because a TypeScript program runs as JavaScript.

Your computer does not understand JavaScript on its own. Outside a web browser, it needs a program called **Node** (also written Node.js), which reads JavaScript or TypeScript instructions and carries them out. Installing Node also installs **npm**, a tool for adding ready-made code to your projects, which you will use in 1.2. Today you will also install a **code editor**, an app for writing program files. Any plain text editor works, but a code editor colors the parts of your code, and VS Code has TypeScript support built in, so it underlines mistakes as you type.

Install the tools

1. Go to nodejs.org and download the version labeled **LTS** (long-term support) for your computer. This course needs Node 24 or newer: it runs TypeScript files directly, and it is the version this course is tested with.
2. Run the installer and accept the default choices. `npm` comes with it.
3. Install VS Code from code.visualstudio.com and open it once so it finishes setting up.

Check that it worked

To talk to Node you use a **terminal**: a window where you type a command, press Enter, and read the reply as text. Every computer has one (Terminal on macOS and Linux, PowerShell or Terminal on Windows), and VS Code has one built in: choose **Terminal → New Terminal**.

Type this command into the terminal and press Enter:

```
node --version
```

The command has two parts: `node` names the program to run, and `--version` asks it only to report which version it is. If Node is installed, the terminal answers with one line: the letter `v` followed by the version number, with the main version number first. Version numbers change often, so yours will not match anyone else's exactly; what matters is that the first number is 24 or higher.

KEY IDEA

Node is the program that runs JavaScript and TypeScript outside a browser. Typing `node --version` in a terminal checks that it is installed and shows which version you have.

Now vary the program you ask: `npm` reports its version the same way.

```
npm --version
```

`npm` answers with a version number too (without the `v`). Its numbers are different from Node's, because `npm` is a separate program that happens to be installed alongside Node. If both commands answer, your tools are ready.

WATCH OUT

"Command not found," or a version below 24. If the terminal says it does not recognize `node`, either Node is not installed or the terminal cannot find it yet. **Close the terminal and open a new one:** a terminal that was already open when you installed Node does not notice the new program. If that does not help, run the installer again. If the version starts with a number below 24 (for example `v20`), an older Node is installed; download the current LTS from nodejs.org and install it over the old one, then open a new terminal and check again.

PAUSE AND TRY

Question 1 Short answer

Sam types `node --version` and the terminal answers `v20` followed by more numbers. Then Sam's friend Priya types the same command and the terminal says the command was not found. What should each of them do, and why?

1.2 Creating a Project

Node can run a single file, but the type checker needs a little setup first: it has to be installed, and it has to be told how strict to be. Programmers keep each program, with its settings and tools, in its own folder called a **project**. Every day of this course starts the same way, so it is worth doing slowly once.

Make the folder and open it

1. Make a folder named `day01` somewhere easy to find, such as your Documents folder.

2. In VS Code, choose **File → Open Folder** and open `day01`.
3. Open the terminal with **Terminal → New Terminal**. It starts inside `day01`, the folder you opened. Every command below must run in that folder.

Tell npm about the project

Type this command and press Enter:

```
npm init -y
```

`npm init` creates a file named `package.json`, a short description of the project: its name, its version, and later the tools it uses. Normally `npm init` asks you a list of questions; `-y` means "yes, use the default answer for every question." The terminal prints the new file's contents, and `package.json` appears in VS Code's file list. You do not need to edit it today.

Install the type checker

Next, install TypeScript into the project:

```
npm install --save-dev typescript
```

A **package** is ready-made code that someone has published for others to use, and npm (short for "Node package manager") downloads packages for you. This command has three parts:

Part	Meaning
<code>npm install</code>	Download a package into this project
<code>--save-dev</code>	Record it in <code>package.json</code> as a tool for developing the program (you need it while writing code, not to run it)
<code>typescript</code>	The package's name; it contains the type checker

npm downloads the package into a new folder named `node_modules`, and writes a file named `package-lock.json` that records exactly what it installed. Both are managed by npm: never edit them by hand. If you delete `node_modules`, running `npm install` brings it back.

Turn on strict checking

Finally, the type checker reads its settings from a file named `tsconfig.json`. In VS Code choose **File → New File**, name it `tsconfig.json`, and type this into it exactly:

```
{
  "compilerOptions": {
    "strict": true,
    "noEmit": true,
    "target": "ES2024",
    "module": "nodenext",
    "erasableSyntaxOnly": true,
    "allowImportingTsExtensions": true
  }
}
```

Every setting is a name in quotes, a colon, and a value, and the settings are separated by commas. Here is what each one asks for. You do not need to remember them today; Day 13 comes back to this file.

Setting	What it asks the type checker to do
"strict": true	Use strict mode : the full set of checks, so it catches as many mistakes as possible
"noEmit": true	Only check; do not write any new files (Node runs your <code>.ts</code> files directly)
"target": "ES2024"	Expect a modern version of JavaScript, the one Node 24 understands
"module": "nodenext"	Understand files the way Node does
"erasableSyntaxOnly": true	Allow only TypeScript features that Node can run by simply removing the types
"allowImportingTsExtensions": true	Allow one file to load another by its <code>.ts</code> name (used from Day 13)

Your folder now looks like this:

```
day01/  
  node_modules/  
  package-lock.json  
  package.json  
  tsconfig.json
```

KEY IDEA

A TypeScript project is a folder with a `package.json` (from `npm init -y`), the TypeScript package (from `npm install --save-dev typescript`), and a `tsconfig.json` that turns on strict mode.

WATCH OUT

Running the commands in the wrong folder. `npm` creates `package.json` and `node_modules` in whatever folder the terminal is in. If you run `npm init -y` in your home folder by accident, the project files land there instead of in `day01`. Always open the project folder in VS Code first and use its terminal; the folder name appears in VS Code's file list, so you can check before you type. If it happens, delete the stray files and run the commands again in the right folder.

PAUSE AND TRY

Question 2 Multiple choice

What does `npm install --save-dev typescript` do?

- A. It creates `package.json` and `tsconfig.json`.
- B. It downloads the TypeScript package into `node_modules` and records it in `package.json` as a development tool.
- C. It installs Node, which comes with npm.
- D. It checks every `.ts` file in the folder for mistakes.

Question 3 Short answer

Lee sets up `day01`, writes a program, and runs the type checker, but it prints a long help screen that includes the line `tsc: The TypeScript Compiler` instead of checking anything. Lee's folder contains `node_modules`, `package.json`, `package-lock.json`, and the program file. What is missing, and why does it matter?

1.3 Your First Program

The project is ready, but it does nothing yet. Time to give the computer its first instruction.

A TypeScript program lives in a file whose name ends in `.ts`. You write the file in your editor, save it, and then ask Node to run it from the terminal. Node starts at the first line and carries out the instructions until it reaches the end of the file.

WORKED EXAMPLE Your first program

1. In VS Code, choose **File → New File**, name it `hello.ts`, and type these two lines:

`hello.ts`

```
1 console.log("Hello, world!");
2 console.log("I am learning TypeScript.");
```

1. Save the file.
2. In the terminal (still in `day01`), run it by typing `node` followed by the file name:

```
node hello.ts
```

Node opens `hello.ts`, carries out line 1, then line 2, and the terminal shows:

```
Hello, world!
I am learning TypeScript.
```

You have written and run your first program. Every later day works the same way: edit the file, save, and run it again with the same command.

Take one line apart. `console.log("Hello, world!");` has four pieces:

Piece	What it is
<code>console</code>	The console : the text output of your program, which appears in the terminal
<code>.log</code>	Read the dot as "the console's": <code>console.log</code> is the console's <code>log</code> instruction
<code>("Hello, world!")</code>	The value handed to <code>log</code> , inside parentheses
<code>;</code>	A semicolon marks the end of the instruction, like a period at the end of a sentence

`console.log` is a **function**: a named instruction that does one job when you use it. Using a function is called **calling** it, and you call a function by writing its name followed by parentheses. The value inside the parentheses is the **argument**, the thing you hand to the function to work on.

`console.log` shows its argument in the terminal and then moves to a new line.

Text in a program goes inside double quotes, like `"Hello, world!"`. A piece of text in quotes is called a **string**, because it is a string of characters one after another. The quotes are not part of the text; they only show where it starts and ends, which is why the terminal shows `Hello, world!` with no quotes. (Single quotes work too, and Day 3 covers strings in depth; this course uses double quotes.)

KEY IDEA

A TypeScript program is a saved `.ts` file. `node hello.ts` runs it from the first line to the last, and each `console.log("...")` shows one line of text in the terminal.

WATCH OUT

Running an old version of the file, or from the wrong place. Node runs the file as it is saved on disk, not the text in your editor. If you change a line and run again without saving, you see the old output; VS Code shows a dot on the file's tab while it has unsaved changes. And if the terminal is not in the folder that holds `hello.ts`, Node reports that it cannot find the module (the file) you named, because it looks for `hello.ts` in the terminal's current folder. Use VS Code's terminal with the project folder open, so the two always match.

PAUSE AND TRY

Question 4 Short answer

Ana changes line 2 of `hello.ts` to `console.log("I am learning to code.");`, but she does not save the file. Then she runs `node hello.ts`. What does the terminal show, and what should she do?

1.4 Printing Text and Numbers

A program only shows what you log, so `console.log` is the most important tool of the day. It is worth knowing exactly how it treats text and numbers.

Numbers are written without quotes, like `9` or `3.5`. A program can also do arithmetic: `+` adds, `-` subtracts, `*` (an asterisk) multiplies, and `/` divides. A piece of code that works out to a value, such as `4 + 5`, is called an **expression**. When an expression is the argument to `console.log`, the program works out its value first and then shows the result.

The quotes change everything. Whatever is inside quotes is plain text, copied exactly; anything outside quotes is an expression to work out.

WORKED EXAMPLE Quotes or no quotes

print-basics.ts

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

```
Total cost:
9
4 + 5
```

Line	Argument	What the program does	Shown
1	"Total cost:"	It is a string, so copy the text inside the quotes	Total cost:
2	4 + 5	No quotes, so work out the expression: 9	9
3	"4 + 5"	It is a string, so copy the characters exactly	4 + 5

Lines 2 and 3 differ only by the quotes, and that is the whole difference between doing the math and showing the math.

Now vary the number of arguments. `console.log` can take several, separated by commas; it shows them on one line with one space between each. And `console.log()` with nothing inside shows an empty line.

print-commas.ts

```
1 console.log("Tickets:", 3 * 12);
2 console.log();
3 console.log("Thanks for visiting!");
```

```
Tickets: 36
```

```
Thanks for visiting!
```

Line 1 has two arguments: the string "Tickets:" and the expression `3 * 12`. The program works out 36, then shows the text, a space, and the number. Line 2 shows a blank line, and line 3 shows the last string.

Arithmetic follows the same order of operations as math class: multiplication and division happen before addition and subtraction, and parentheses go first.

order-of-operations.ts

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

```
14
20
3.5
4
```

In `2 + 3 * 4`, the program multiplies `3 * 4` first to get 12, then adds 2 to get 14. In `(2 + 3) * 4`, the parentheses make it add first to get 5, then multiply by 4 to get 20. The numbers are the same; only the order changed, and so did the answer. The last two lines show that division gives a decimal answer when it needs one (3.5) and a whole number when it comes out even (4).

Finally, an expression that is not inside `console.log` is worked out and then thrown away. Nothing shows it:

silent-sum.ts

```
1 console.log("Adding 2 and 3:");
2 2 + 3;
3 console.log(2 + 3);
```

```
Adding 2 and 3:
5
```

Line 2 works out $2 + 3$, but nothing asks to show the result, so no line appears for it. Only line 3, which hands the expression to `console.log`, shows 5.

KEY IDEA

`console.log` shows its arguments on one line, separated by spaces, then moves to the next line. Text in quotes is shown exactly as written; an expression without quotes is worked out first. A value that is never logged is never shown.

WATCH OUT

Putting quotes around math you want worked out. `console.log("6 * 7")` shows `6 * 7`, not 42, because the quotes turn it into text. Remove the quotes when you want the answer. The opposite mistake, leaving the quotes off words, is an error you will meet in 1.9.

PAUSE AND TRY**Question 5** Multiple choice

Which line shows 7?

- A. `console.log("3 + 4");`
- B. `console.log(3 + 4);`
- C. `console.log("3", "+", "4");`
- D. `console.log(34);`

Question 6 Tracing

What does this program show?

price-line.ts

```
1 console.log("Price:", 2 * 5, "dollars");
```

1.5 Programs Run Top to Bottom

Your programs so far have had a few lines each, and their output appeared in the same order as the lines. That is not a coincidence; it is the rule every program follows.

Each complete instruction in a program, such as one call to `console.log` ending in a semicolon, is a **statement**. This course writes one statement per line. When you run a program, Node starts at

line 1 and carries out the statements **one at a time, from top to bottom**. It finishes each statement before starting the next, and it never skips ahead or rearranges them to make more sense.

WORKED EXAMPLE Following the order

morning-steps.ts

```
1 console.log("Wake up");
2 console.log("Eat breakfast");
3 console.log("Walk to school");
```

Step	Line run	Terminal so far
1	1	Wake up
2	2	Wake up, Eat breakfast
3	3	Wake up, Eat breakfast, Walk to school

```
Wake up
Eat breakfast
Walk to school
```

Now change one thing: move the last line to the top. The statements are the same, but the order is not:

morning-steps-swapped.ts

```
1 console.log("Walk to school");
2 console.log("Wake up");
3 console.log("Eat breakfast");
```

```
Walk to school
Wake up
Eat breakfast
```

Node ran the lines in exactly the order you wrote them, even though walking to school before waking up makes no sense. The computer cannot know what you meant; it follows the order on the page.

KEY IDEA

A program's statements run one at a time, from the top line to the bottom line, in exactly the order written.

WATCH OUT

"The computer will know what I meant." A program can run without any error message and still do the wrong thing, as `morning-steps-swapped.ts` did. No error does not mean correct. Before you run a program, write down what it should show; afterward, compare.

PAUSE AND TRY**Question 7** Short answer

This program should count down 3, 2, 1, then show `Liftoff!`. Run it, compare with what it should do, and say which change fixes it.

countdown-wrong.ts

```
1 console.log(3);
2 console.log(1);
3 console.log(2);
4 console.log("Liftoff!");
```

1.6 Comments: Notes the Computer Ignores

Next week you will open today's files and wonder what they were for. Programmers solve this by leaving notes in the code itself, written for people, not for the computer.

A **comment** is text in a program that Node and the type checker skip completely. TypeScript has two kinds:

- A **line comment** starts with `//` (two slashes) and runs to the end of that line. It can fill a whole line, or follow code on the same line, where the code before the `//` still runs.
- A **block comment** starts with `/*` and ends with `*/`, and can cover several lines.

WORKED EXAMPLE Comments in a receipt program

receipt-comments.ts

```
1 // Shopping receipt for Saturday
2 console.log("Apples:", 6);
3 console.log("Bread:", 4); // one loaf
4 // console.log("Milk:", 3);
5 console.log("Done");
```

```
Apples: 6
Bread: 4
Done
```

Line	What happens
1	Whole line is a comment: skipped
2	Shows Apples: 6
3	Shows Bread: 4; the comment <code>// one loaf</code> after the code is ignored
4	Starts with <code>//</code> , so the whole line is a comment, even though it looks like code
5	Shows Done

Line 4 shows a common trick called **commenting out**: putting `//` in front of a line of code turns it off without deleting it. Remove the `//` and the line runs again.

Now vary the kind of comment, and where the slashes sit. A block comment covers every line between `/*` and `*/`. And inside quotes, `//` is just two characters in the text, not the start of a comment:

block-comment.ts

```
1 /*
2   Ticket prices for the school play.
3   All prices are in dollars.
4 */
5 console.log("Adult:", 8);
6 console.log("Student:", 5); // half price, rounded down
7 console.log("Visit https://play.example // tickets");
```

```
Adult: 8
Student: 5
Visit https://play.example // tickets
```

Lines 1 to 4 are one block comment, so nothing runs until line 5. Line 6 ends with a line comment. On line 7, both pairs of slashes are inside the string, so the whole web address and the `// tickets` are shown as text.

KEY IDEA

`//` outside quotes starts a comment that runs to the end of the line; `/* ... */` comments out everything between them. Comments explain code to people; commenting out a line turns it off.

WATCH OUT

Commenting the wrong part of a line. `//` at the start of a line turns off everything on it. Students sometimes put `//` in front of a `console.log` to add a note and are surprised when that line stops showing. Put notes on their own line, or after the code. And a block comment that is never closed with `*/` turns the rest of the file into a comment.

PAUSE AND TRY

Question 8 Tracing

What does this program show?

comment-trace.ts

```
1 // Team scores
2 console.log("Red team:", 10 + 5);
3 // console.log("Blue team:", 12);
4 console.log("Green team:", 7); // console.log("Gold team:", 9);
5 /* console.log("Silver team:", 4); */
```

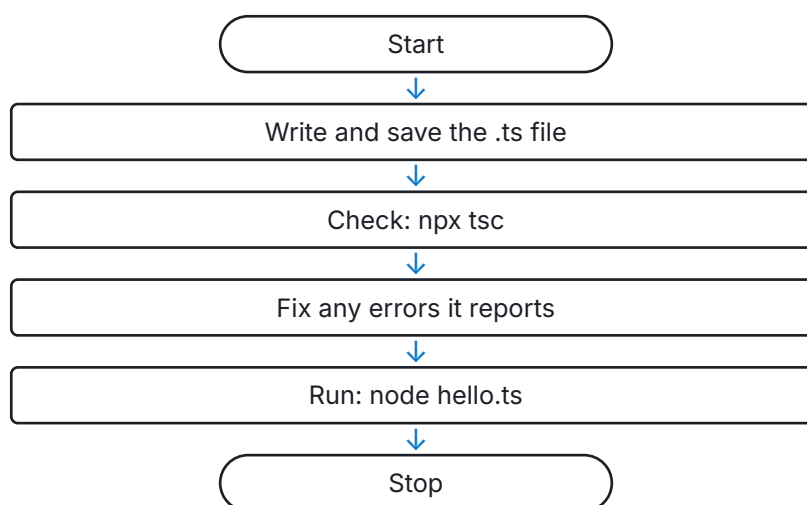
1.7 Two Steps: Check, Then Run

So far you have run every program straight away with `node`. That works, but it skips the reason TypeScript exists. Some mistakes can be found just by reading the code, before it ever runs, and TypeScript has a tool that does exactly that.

Every value in a program has a **type**, the kind of value it is. `12` and `3.5` are of type `number`; `"Hello"` is of type `string`. (Day 2 introduces more types.) The **type checker** is a program that reads your code without running it and checks that every value is used in a way its type allows. A mistake it finds is a **type error**. The type checker is called `tsc`, short for "TypeScript compiler," and you installed it in 1.2.

So working on a TypeScript program has two steps, with two different commands:

Step	Command	What it does
1. Check	<code>npx tsc</code>	Reads every <code>.ts</code> file in the project and reports type errors. Runs nothing.
2. Run	<code>node hello.ts</code>	Runs one file from top to bottom and shows its output. Checks no types.



The two-step routine for every program: check first, then run.

`npx` runs a tool installed inside the current project; `npx tsc` means "run the `tsc` that lives in this project's `node_modules`." Try it on your project now:

```
npx tsc
```

If every file is fine, `npx tsc` prints **nothing at all** and the terminal is ready for the next command. Silence means success. When it does find a mistake, it prints a message for each one; 1.8 shows how to read them. It checks every `.ts` file in the folder at once, so each message starts with the name of the file it is about.

Why does `node` not check types? Node runs TypeScript by **type stripping**: it removes the type labels and runs what is left as plain JavaScript, without reading the labels first. That is fast, but it means `node` **runs a file even if `npx tsc` would report type errors in it**. Only `npx tsc` checks. That is why the routine is always check first, then run.

A mistake that `npx tsc` cannot see shows up only while the program runs, when Node reaches the broken line and has to stop. That is a **runtime error**, and you will meet one in 1.10. So TypeScript programmers talk about two kinds of error:

Question	Type error	Runtime error
Who reports it?	The type checker, <code>npx tsc</code>	Node, while running the program
When?	Before anything runs	When Node reaches the broken line
Does any output appear?	No, nothing has run	Yes, lines before the mistake already ran

VS Code runs the same type checker in the background, so many type errors show up as a red wavy underline (a "red squiggle") while you type. Treat each squiggle as a mistake found for free, but still run `npx tsc` before `node`: the terminal command checks every file, including ones you do not have open.

KEY IDEA

`npx tsc` checks types without running anything, and prints nothing when all is well.
`node file.ts` runs the program without checking types. Check first, then run.

WATCH OUT

Leaving a broken file in the folder. `npx tsc` checks every `.ts` file in the folder, not just the one you are working on. Today you will type several programs that are broken on purpose, to see their messages. If you leave one there, every later `npx tsc` reports it again, and your new program never gets a clean check. So whenever you type a broken example, read its message, then **fix it or delete it** before you move on.

WATCH OUT

Skipping the check because the program ran. "It ran, so it has no type errors" is false: Node strips the types without reading them, so a program can run while `npx tsc` would have reported a mistake. The reverse also catches people: `npx tsc` printing nothing does not mean the program shows the right thing. It means the types fit; the order of your lines, your arithmetic, and runtime errors are still yours to check by running it.

PAUSE AND TRY

Question 9 Multiple choice

Kai types `npx tsc` and the terminal prints nothing and is ready for the next command. What does that mean?

- A. The program ran, but `console.log` did not work.
- B. The type checker found no type errors in any `.ts` file in the project.
- C. The type checker is not installed.
- D. The program is correct and will show the right output.

Question 10 Short answer

Ravi says, "I always just run `node`. If there were a type error, Node would tell me." Is Ravi right? Explain using what `node` does with the types.

1.8 Reading a Type Error

Everyone who writes code makes mistakes, every day. The difference between a beginner who gets stuck and one who keeps going is reading the error message instead of panicking. The type checker's messages tell you exactly where to look and what is wrong.

To see a type error you need code that says what type something should be. Here is a small preview of Day 2:

```
1 const muffinsLeft: number = 12;
```

Read it as "muffinsLeft is a name for a value; its type is `number`; its value is 12." The word `const` creates the name, and the part `: number` is a **type annotation**, a label that tells the type checker what type the value must be. Day 2 explains names and annotations fully; today you need only that one sentence. After this line, `console.log("Muffins left:", muffinsLeft)` shows the value the name holds.

WORKED EXAMPLE A number in quotes

A café program records how many muffins are left. The value was typed in quotes by mistake:

muffins-wrong.ts

```
1 const muffinsLeft: number = "12";
2 console.log("Muffins left:", muffinsLeft);
```

Running `npm tsc` prints this line (in color in most terminals):

```
muffins-wrong.ts:1:7 - error TS2322: Type 'string' is not assignable to type 'number'.
```

Read it in four parts:

Part	Meaning
muffins-wrong.ts	The file the mistake is in
1:7	Line 1, column 7: the seventh character on the line, where <code>muffinsLeft</code> starts
error TS2322	It is an error, and TS2322 is its error code , a number for this kind of problem
Type 'string' is not assignable to type 'number'.	What is wrong, in words

The words need one translation: **assignable** means "allowed to be stored there." So the message says: "you are trying to store a string where only a number is allowed." "12" is in quotes, so it is a string, but the annotation says `number`. Under the message, the terminal repeats line 1 of the code and draws a row of `~` characters under `muffinsLeft`, then ends with a summary such as `Found 1 error in muffins-wrong.ts:1`.

The fix is to remove the quotes, so the value is the number 12:

muffins.ts

```
1 const muffinsLeft: number = 12;  
2 console.log("Muffins left:", muffinsLeft);
```

Delete `muffins-wrong.ts` (or fix it the same way) and type the fixed program as `muffins.ts`. Now `npx tsc` prints nothing, and `node muffins.ts` shows:

```
Muffins left: 12
```

Now vary the mistake the other way round: a number where a string is expected.

shop-name-wrong.ts

```
1 const shopName: string = 42;  
2 console.log("Welcome to", shopName);
```

```
shop-name-wrong.ts:1:7 - error TS2322: Type 'number' is not assignable to type 'string'.
```

The same code, TS2322, with the two types swapped: 42 is a number, but the annotation asks for a string. Read the message as "Type *what you gave* is not assignable to type *what was asked for*." The fix depends on what you meant: either put the shop's name in quotes, or change the annotation. Fix the file one of those ways, or delete it, before you go on, so it does not show up in your next check.

KEY IDEA

A type error message gives the file, `line:column`, an error code, and a sentence. TS2322 means a value of one type was put where another type was required: "Type 'given' is not assignable to type 'required'."

WATCH OUT

Reading only the error code. TS2322 alone does not tell you which line is wrong or which types clash. Read the whole line: the file and line number tell you where to look, and the sentence tells you which type you gave and which was required. The column points at the start of the name or value the checker is unhappy about.

PAUSE AND TRY

Question 11 Debugging

`npx tsc` reports the message below for this program. Explain what the message is pointing at, and write the fix.

debug-type.ts

```
1 console.log("Class trip");  
2 const busSeats: number = "forty";  
3 console.log("Seats:", busSeats);
```

```
debug-type.ts:2:7 - error TS2322: Type 'string' is not assignable to type 'number'.
```

1.9 Names the Checker Does Not Know

The most common mistakes on the first day are tiny: a letter left out, or quotes left off. The type checker catches these too, because it knows every name a program is allowed to use.

A **name** is a word in code that stands for something, such as `console` or `muffinsLeft`. Some names are built in (`console` is always available); others, like `muffinsLeft`, exist because your code created them. Any word outside quotes is treated as a name, and the type checker looks it up. If it finds nothing by that name, it reports an error. Names are **case-sensitive**: capital and small letters count as different, so `Console` is not the same name as `console`.

WORKED EXAMPLE Forgetting the quotes

missing-quotes.ts

```
1 console.log("Ready");
2 console.log(Hello);
3 console.log("Done");
```

```
missing-quotes.ts:2:13 - error TS2304: Cannot find name 'Hello'.
```

Without quotes, `Hello` is not text; the checker treats it as a name and looks it up. There is no name `Hello`, so it reports **TS2304: Cannot find name**. Line 2, column 13 is where `Hello` starts, just after `console.log(`. The fix is `console.log("Hello");`.

Because `npm tsc` runs nothing, `Ready` does not appear: you get the message about line 2 before line 1 ever runs.

Now vary the mistake: keep the quotes, but misspell `console`.

misspelled-console.ts

```
1 console.log("Ready");
2 consol.log("Go");
```

```
misspelled-console.ts:2:1 - error TS2552: Cannot find name 'consol'. Did you mean 'console'?
```

The same problem, a name the checker cannot find, with a bonus: when a known name looks very close, the checker uses code **TS2552** and adds a suggestion, "Did you mean 'console'?". With `Hello` there was no similar name to suggest, so the message was plain TS2304. Either way, the words "Cannot find name" mean "check the spelling, the capitals, and the quotes."

There is one more kind of mistake the checker reports before anything runs: broken punctuation. Every language has rules for how code must be written, called its **syntax**: every `(` needs a matching `)`, and every string needs both quotes. A broken rule is a **syntax error**:

missing-paren.ts

```
1 console.log("Hello");
2 console.log("World";
3 console.log("Goodbye");
```

```
missing-paren.ts:2:20 - error TS1005: ')' expected.
```

At line 2, column 20, the checker reached the semicolon while the (was still open, so it says it expected a) there. Syntax errors have codes starting with TS1. Add the) before the semicolon to fix it.

Mistake	Example	Message
Text without quotes	<code>console.log(Hello);</code>	TS2304: Cannot find name 'Hello'.
Misspelled name	<code>consol.log("Go");</code>	TS2552: Cannot find name 'consol'. Did you mean 'console'?
Missing)	<code>console.log("World";</code>	TS1005: ')' expected.

If you typed `missing-quotes.ts`, `misspelled-console.ts`, and `missing-paren.ts` to see these messages, fix or delete each one now, so your next `npm tsc` checks only the programs you are working on.

KEY IDEA

Any word outside quotes is a name, and the type checker reports a name it does not know: TS2304 "Cannot find name," or TS2552 when it can suggest a close match. Check spelling, capitals, and quotes.

WATCH OUT

Fixing the wrong line. Usually the line number points straight at the mistake. With broken punctuation, the checker sometimes notices only a little later: a missing closing quote can produce several messages, some about the next line. Fix the **first** message, run `npm tsc` again, and see what is left; often the others disappear.

Errors that appear only after a fix. When any file in the project has a syntax error, `npm tsc` reports only the syntax errors and stops there. Once they are fixed, the next run can report type errors such as TS2304 that were there all along. That is normal: keep fixing and rerunning until `npm tsc` prints nothing.

PAUSE AND TRY

Question 12 Debugging

`npm tsc` reports an error for this program. Predict the message, then fix the program.

debug-name.ts

```
1 console.log("Quiz time!");
2 console.log("Question 1");
3 console.log(Goodbye);
```

Question 13 Multiple choice

Which line makes `npx tsc` report Did you mean 'console'?

- A. `console.log(Ready);`
- B. `consle.log("Ready");`
- C. `console.log("Ready";`
- D. `console.log("Ready");`

1.10 Reading a Runtime Error

The type checker reads code without running it, so it can only catch mistakes visible in the code itself. Some mistakes depend on what happens when the code actually runs. Those slip past `npx tsc`, and Node reports them when it reaches them.

To see one, you need a tool that can fail while running. Strings have a handy one: `"=".repeat(12)` makes a new string of 12 copies of `"="`. (This is a preview of Day 3, which covers many such string tools.) The number in the parentheses is how many copies to make.

WORKED EXAMPLE A line of equals signs

banner.ts

```
1 console.log("=".repeat(12));
2 console.log("BAKE SALE");
3 console.log("=".repeat(12));
```

```
=====
BAKE SALE
=====
```

Lines 1 and 3 each show twelve `=` characters, making a border around the title.

Now vary one character: a stray minus sign on line 3 asks for `-12` copies, and add a fourth line.

banner-crash.ts

```
1 console.log("=".repeat(12));
2 console.log("BAKE SALE");
3 console.log("=".repeat(-12));
4 console.log("Everything is $2");
```

`npx tsc` prints nothing: `repeat` needs a number, and `-12` is a number, so the types fit. Then `node banner-crash.ts` shows this (in the line ending in `:3`, your terminal shows the full path to your own folder where `...` appears here):

```

=====
BAKE SALE
.../day01/banner-crash.ts:3
console.log("=".repeat(-12));
                        ^

RangeError: Invalid count value: -12
    at ...

```

After that come several more lines starting with `at`, which list the places inside Node involved, and a last line naming your Node version. Read the report in order:

Part	Meaning
===== and BAKE SALE	Normal output: lines 1 and 2 ran before the error
.../day01/banner-crash.ts:3	The file, and the line where the program stopped: 3
The code line and the <code>^</code>	The line itself, with a caret under the part that failed (<code>repeat</code>)
RangeError: Invalid count value: -12	The kind of error and its cause: -12 is outside the range of allowed counts
Lines starting with <code>at</code>	Where inside Node it happened; you can skip them for now

You cannot make a negative number of copies, so `repeat` gave up and the program stopped with a **RangeError**, the kind of runtime error for a number outside the allowed range. Line 4 never ran: once a program stops with an error, the statements after it are skipped. Fix the minus sign and all four lines show.

The type checker checks the **kind** of each value, not whether the value makes sense. `-12` is the right kind of value (a number); only running the program discovers that it is the wrong number.

Question	Type error (npx tsc)	Runtime error (node)
When is it found?	Before anything runs	When Node reaches that line
Does earlier output appear?	No, nothing has run	Yes, earlier lines have already run
What it looks like	file.ts:2:7 - error TS2322: ...	The line, a caret, then RangeError: ...
Today's examples	Wrong type, unknown name, missing)	<code>repeat</code> with a negative count

KEY IDEA

A runtime error happens while the program runs, so lines before it have already shown their output and lines after it never run. Read the line number, the caret, and the line naming the error, such as `RangeError`.

WATCH OUT

Reading only the first line of the message. The most useful parts of a runtime error are the line naming the error (`RangeError: ...`), which says what went wrong, and the code line with

its caret, which shows where. They are buried between your program's normal output and the `at` lines. Look for the word ending in `Error`, then look just above it.

PAUSE AND TRY

Question 14 Tracing

`npx tsc` reports no errors for this program. Write everything `node runtime-trace.ts` shows, up to and including the line naming the error.

`runtime-trace.ts`

```
1 console.log("Step 1");
2 console.log("Step 2:", 5 * 2);
3 console.log("-".repeat(2 - 5));
4 console.log("Step 3");
```

Question 15 Multiple choice

A program's first two lines are correct `console.log` statements, and line 3 has a mistake. The program is checked with `npx tsc` and then run with `node`, whatever the check says. In which case do the first two lines show in the terminal **and** `npx tsc` report no error for the program?

- A. Line 3 is `console.log(Done);`
- B. Line 3 is `console.log("Done");`
- C. Line 3 is `console.log("-".repeat(-4));`
- D. Line 3 is `const total: number = "9";`

Putting it together

Maya is making a sign for her school's bake sale. Follow her through three rounds of check and run; each round uses a different part of today's lesson.

Round 1: check (1.6, 1.7, 1.9). Maya starts with a comment saying what the program is for, then the sign's lines. She runs `npx tsc` first:

`bake-sale-attempt1.ts`

```
1 // Sign for the bake sale table
2 console.log("=".repeat(20));
3 console.log("BAKE SALE TODAY");
4 console.log("Muffins left:", 30 - 12);
5 consol.log("Cookies left:", 48 - 20);
6 console.log("=".repeat(20 - 25));
```

```
bake-sale-attempt1.ts:5:1 - error TS2552: Cannot find name 'consol'. Did you mean 'console'?
```

Nothing has run yet. The checker points at line 5, column 1, where she typed `consol`, and suggests `console`. She fixes the spelling and checks again; this time `npx tsc` prints nothing.

Round 2: run (1.4, 1.5, 1.10). Now she runs `node bake-sale-attempt2.ts`:

bake-sale-attempt2.ts

```

1 // Sign for the bake sale table
2 console.log("=".repeat(20));
3 console.log("BAKE SALE TODAY");
4 console.log("Muffins left:", 30 - 12);
5 console.log("Cookies left:", 48 - 20);
6 console.log("=".repeat(20 - 25));

```

```

=====
BAKE SALE TODAY
Muffins left: 18
Cookies left: 28
.../day01/bake-sale-attempt2.ts:6
console.log("=".repeat(20 - 25));
                        ^

```

RangeError: Invalid count value: -5

Lines 2 to 5 ran (line 1 is a comment), working out $30 - 12$ and $48 - 20$ along the way. Then line 6 stopped the program. Maya wanted a short border of 5, but she wrote $20 - 25$, which is -5 . The type checker could not catch it: $20 - 25$ is a number, just the wrong one.

Round 3: fixed. She swaps the numbers to $25 - 20$, checks, and runs:

bake-sale.ts

```

1 // Sign for the bake sale table
2 console.log("=".repeat(20));
3 console.log("BAKE SALE TODAY");
4 console.log("Muffins left:", 30 - 12);
5 console.log("Cookies left:", 48 - 20);
6 console.log("=".repeat(25 - 20));

```

```

=====
BAKE SALE TODAY
Muffins left: 18
Cookies left: 28
=====

```

Round	Kind of problem	Found by	What showed
1	Type error, TS2552	npx tsc, before running	Nothing ran
2	Runtime error	node, while running	The first four lines
3	None	Runs to the end	All five lines

Common mistakes

Every message from today in one place, to look up when you meet one again.

Message	Reported by	Usual cause	Fix
---------	-------------	-------------	-----

TS2322: Type 'string' is not assignable to type 'number'.	<code>npx tsc</code>	A value in quotes where a number is required	Remove the quotes, or fix the annotation
TS2304: Cannot find name 'Hello'.	<code>npx tsc</code>	Text without quotes, or a misspelled name	Add quotes, or fix the spelling
TS2552: Cannot find name 'console'. Did you mean 'console'?	<code>npx tsc</code>	A misspelled name close to a real one	Use the suggested name
TS1005: ')' expected.	<code>npx tsc</code>	A (with no matching)	Add the)
TS1002: Unterminated string literal.	<code>npx tsc</code>	A string with no closing quote	Add the closing "
RangeError: Invalid count value: -12	<code>node</code>	repeat given a negative number	Give it zero or a positive number

Vocabulary

Term	Definition	Example
program	A list of instructions for a computer	<code>hello.ts</code>
programming language	A language with strict rules for writing programs	TypeScript
JavaScript / TypeScript	The language browsers and Node run / JavaScript plus types	A <code>.ts</code> file is TypeScript
Node / npm	The program that runs JavaScript and TypeScript outside a browser / the tool that installs packages	<code>node hello.ts</code> , <code>npm init -y</code>
code editor	An app for writing program files	VS Code
terminal	A window where you type commands and read text replies	Where you type <code>node --version</code>
project	A folder holding one program with its settings and tools	<code>day01/</code>
package / <code>node_modules</code>	Ready-made code published for others to use / the folder npm installs packages into	<code>typescript</code>
<code>package.json</code>	The file describing a project and the packages it uses	Made by <code>npm init -y</code>
<code>tsconfig.json</code> / strict mode	The type checker's settings file / its setting for the full set of checks	<code>"strict": true</code>
console	The text output of a program, shown in the terminal	<code>console.log("Hi");</code>
function / calling	A named instruction that does one job / using it, with parentheses	<code>console.log(...)</code>

argument	A value handed to a function inside its parentheses	<code>"Hi" in console.log("Hi")</code>
string	Text in quotes	<code>"Hello"</code>
expression	Code that works out to a value	<code>4 + 5</code>
statement	One complete instruction, ending in a semicolon	<code>console.log("Done");</code>
comment / commenting out	Text the computer skips, after <code>//</code> or between <code>/*</code> and <code>*/</code> / turning code off with <code>//</code>	<code>// one loaf</code>
line comment / block comment	A comment from <code>//</code> to the end of the line / a comment from <code>/*</code> to <code>*/</code>	<code>/* prices in dollars */</code>
type	The kind of value something is	<code>number, string</code>
type checker / npx	The program (<code>tsc</code>) that checks types without running code / runs a tool installed in the project	<code>npx tsc</code>
type stripping	How Node runs TypeScript: it removes the types without checking them	<code>node hello.ts</code>
type annotation	A label saying what type a value must be	<code>: number</code>
type error / error code	A mistake the type checker reports / its number for that kind of mistake	<code>TS2322</code>
assignable	Allowed to be stored in a place that requires a certain type	A number is assignable to <code>number</code>
name / case-sensitive	A word outside quotes that stands for something / capitals and small letters differ	<code>console</code> is not <code>Console</code>
syntax / syntax error	The rules for how code must be written / a broken syntax rule	<code>')' expected.</code>
runtime error	A mistake found while the program runs, which stops it at that line	<code>RangeError: Invalid count value: -12</code>

Summary

- **Node** runs TypeScript files (`node --version` must show 24 or newer). A **project** folder gets a `package.json` from `npm init -y`, the type checker from `npm install --save-dev typescript`, and a `tsconfig.json` with **strict mode** on.
- `console.log` shows its arguments on one line, separated by spaces; text in quotes is shown exactly, and **expressions** without quotes are worked out first. **Comments** (`//` and `/* */`) are skipped.

- **Statements** run one at a time, top to bottom, in the order written. A program with no error can still do the wrong thing.
- Two steps, two commands: `npx tsc` checks **types** without running anything (silence means no type errors); `node file.ts` runs the program without checking types.
- **Type errors** (TS2322, TS2304, TS2552, TS1005) are reported before anything runs and give `file:line:column`, a code, and a sentence. A **runtime error** such as `RangeError` stops a running program at that line, after earlier lines have shown their output.

Practice

Do these in your `day01` folder. Type each program yourself, check it with `npx tsc`, run it with `node`, and only then open the answer.

Question 16 Short answer

Name and address. Write a program that shows your name, then a three-line address, using one `console.log()` per line. (Use a made-up address if you prefer.)

Question 17 Short answer

Five calculations. Write a program that shows the results of five arithmetic expressions, one per line. Use all four operators (`+`, `-`, `*`, `/`) and at least one pair of parentheses. Predict each answer before you run it.

Question 18 Short answer

Break it on purpose. Break your name-and-address program three different ways, one at a time: remove a closing quote, misspell `console`, and remove the quotes around one line's text. After each break, run `npx tsc` and write down the error code, the line and column, and what the message is pointing at. Then undo that break before making the next one, so each run shows only one mistake.

Review questions

Question 19 Multiple choice

A file `answer.ts` contains these two lines. What appears when you run `node answer.ts`?

`answer.ts`

```
1 6 * 7;  
2 console.log("Done");
```

- A. 42, then Done
- B. 6 * 7, then Done
- C. Only Done
- D. A type error, because line 1 does nothing

Question 20 Tracing

Write exactly what this program shows, including any blank line.

review-trace.ts

```
1 // Movie night plan
2 console.log("Movie night");
3 console.log("Snacks:", 3 + 4); // popcorn and juice
4 // console.log("Cancelled");
5 console.log();
6 /* Tickets cost 9 dollars each. */
7 console.log("Tickets:", 2 * 9);
8 console.log("2 * 9");
```

Question 21 Multiple choice

Program P and program Q each have four lines and differ only in line 3. Ana keeps each one in its own project folder and follows the routine there: she runs `npm tsc`, and she runs `node` only if the check prints nothing.

program-p.ts

```
1 console.log("A");
2 console.log("B");
3 console.log(Finished);
4 console.log("C");
```

program-q.ts

```
1 console.log("A");
2 console.log("B");
3 console.log("*".repeat(-1));
4 console.log("C");
```

Which statement is true?

- A. Both fail the check, so neither one runs.
- B. P fails the check, so it never runs; Q passes, runs, shows A and B, then stops with a runtime error.
- C. P passes, runs, and shows A and B; Q fails the check.
- D. Both pass the check, and both show A, B, and C.

Question 22 Tracing

Predict the exact line `npm tsc` prints for this program: file, line, column, code, and message. Then write the fix and what the fixed program shows.

review-menu.ts

```
1 // Lunch menu
2 console.log("Soup:", 4);
3 console.log(Salad, 6);
```

Question 23 Short answer

Jo's program should show a countdown, but `node` shows the numbers in the wrong order. Jo says, "That's impossible: `npm tsc` printed nothing, so my program is correct." Explain what `npm tsc` printing nothing does and does not tell Jo, and how Jo should find the problem.

Question 24 Short answer

Sort these four problems into "reported by `npx tsc`" and "reported only by `node`", and say how you can tell from the terminal which one you are looking at: (a) `console.log("Total:" , 4 + 5;` (b) `"-".repeat(3 - 10)` inside a `console.log` (c) `const pages: number = "300";` (d) `console.log(Hi);`

Go further

The TypeScript Handbook's opening pages explain where TypeScript comes from and what the type checker is for, with more examples of errors it catches: [TypeScript for the New Programmer](#) and [The Basics](#).

Reflection

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

- ☐ I can check which version of Node is installed from the terminal. Reread: [1.1 Installing Node and Opening a Terminal](#)
- ☐ I can set up a project with `npm init`, the `typescript` package, and a strict `tsconfig.json`. Reread: [1.2 Creating a Project](#)
- ☐ I can create a `.ts` file and run it with `node`. Reread: [1.3 Your First Program](#)
- ☐ I can predict what `console.log` shows with and without quotes. Reread: [1.4 Printing Text and Numbers](#)
- ☐ I can trace a program from top to bottom. Reread: [1.5 Programs Run Top to Bottom](#)
- ☐ I can write both kinds of comment and comment out a line. Reread: [1.6 Comments: Notes the Computer Ignores](#)
- ☐ I can explain what `npx tsc` does and what `node` does. Reread: [1.7 Two Steps: Check, Then Run](#)
- ☐ I can read a TS2322 message and say which types clash. Reread: [1.8 Reading a Type Error](#)
- ☐ I can fix a name the checker cannot find, and a missing parenthesis. Reread: [1.9 Names the Checker Does Not Know](#)
- ☐ I can tell a runtime error from a type error and say which lines ran before it. Reread: [1.10 Reading a Runtime Error](#)

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

Model answer: Sam has Node installed, but version 20 is older than this course needs; the course requires Node 24 or newer, which runs `.ts` files directly and is the version the course is tested with. Sam should install the current LTS version from nodejs.org, open a new terminal, and check again. Priya's terminal cannot find Node at all. She should close the terminal and open a new one (a terminal that was open during the install does not see the new program), and if that fails, run the installer again.

A complete answer must:

- say that Sam needs a newer Node (24 or higher), because the course requires it to run `.ts` files, and
- give Priya a fix (open a new terminal, or reinstall) and say why it might help.

Reread *1.1 Installing Node and Opening a Terminal* if your answer differed.

Question 2

B. `npm install` downloads a package into the project's `node_modules` folder, and `--save-dev` records it in `package.json` as a tool you need while developing.

- **A** is wrong: `npm init -y` creates `package.json`, and you type `tsconfig.json` yourself.
- **C** is wrong: it is the other way round. You install Node from nodejs.org, and `npm` comes with it; you need `npm` already to run this command.
- **D** is wrong: installing the checker does not run it. Running it is a separate step, `npx tsc`, coming in 1.7.

Reread *1.2 Creating a Project* if you chose differently.

Question 3

Model answer: `tsconfig.json` is missing. The type checker reads its settings, including which files to check, from `tsconfig.json`; with no such file and no file named in the command, it does not know what to check, so it prints its help instead. Creating `tsconfig.json` with the starter settings (including `"strict": true`) fixes it.

A complete answer must:

- name `tsconfig.json` as the missing file, and
- say that it tells the type checker what to check and how strictly.

Reread *1.2 Creating a Project* if your answer differed.

Question 4

Model answer: It shows the old output:

```
Hello, world!  
I am learning TypeScript.
```

Node runs the file as it is saved on disk, and the saved file still has the old line 2. Ana should save the file (the dot on the tab disappears) and run `node hello.ts` again; then line 2 shows `I am learning to code.`

A complete answer must:

- give the old two lines as the output, and
- explain that Node reads the saved file, so she must save before running.

Reread *1.3 Your First Program* if your answer differed.

Question 5

B. With no quotes, `3 + 4` is an expression, so the program works it out and shows `7`.

- **A** is wrong: the quotes make `"3 + 4"` a string, so it shows `3 + 4` exactly.
- **C** is wrong: it shows three strings separated by spaces, which also looks like `3 + 4`.
- **D** is wrong: `34` is the number thirty-four; nothing adds the digits, so it shows `34`.

Reread *1.4 Printing Text and Numbers* if you chose differently.

Question 6

```
Price: 10 dollars
```

There are three arguments. The first and last are strings, shown as written. The middle one, `2 * 5`, has no quotes, so the program works out 10. `console.log` puts one space between each argument.

Reread *1.4 Printing Text and Numbers* if your output differed.

Question 7

It shows `3, 1, 2, Liftoff!`, because the statements run in the order written and lines 2 and 3 are in the wrong order. Swap them:

countdown-fixed.ts

```
1 console.log(3);
2 console.log(2);
3 console.log(1);
4 console.log("Liftoff!");
```

```
3
2
1
Liftoff!
```

There was no error message, because every line is a correct statement. Only comparing the output with what should happen revealed the problem.

Reread *1.5 Programs Run Top to Bottom* if your fix differed.

Question 8

```
Red team: 15
Green team: 7
```

Line 1 is a comment. Line 2 shows `Red team: 15`. Line 3 is commented out, so Blue team never shows. Line 4 shows `Green team: 7`; everything after its `//`, including the code for Gold team, is a comment. Line 5 is a block comment, so Silver team never shows either.

Reread *1.6 Comments: Notes the Computer Ignores* if your output differed.

Question 9

B. `npx tsc` prints nothing when every file passes the check. Silence means success.

- **A** is wrong: `npx tsc` never runs the program, so no output was expected. Running is `node`'s job.
- **C** is wrong: if `tsc` were missing, `npx` would report a problem (or ask to download something) rather than print nothing.
- **D** is wrong: passing the check means the types fit. The program can still show the wrong thing (lines in the wrong order) or stop with a runtime error.

Reread *1.7 Two Steps: Check, Then Run* if you chose differently.

Question 10

Model answer: No. Node runs TypeScript by type stripping: it removes the types and runs the rest as JavaScript without checking them. So Node does not report type errors, and a program with a type error can still run (and may then show something wrong or stop later). Only `npx tsc` checks types, so Ravi should run `npx tsc` first, then `node`.

A complete answer must:

- say that `node` strips types without checking them, and
- say that `npx tsc` is the step that reports type errors.

Reread *1.7 Two Steps: Check, Then Run* if your answer differed.

Question 11

The message points at line 2, column 7, which is where the name `busSeats` starts. The annotation says `busSeats` must be a number, but `"forty"` is in quotes, so it is a string: a string is not allowed where a number is required. Write the value as a number:

debug-type-fixed.ts

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

```
Class trip
Seats: 40
```

Notice that `Class trip` did not appear before the fix: `npx tsc` only checks, so nothing runs while you read its messages.

Reread *1.8 Reading a Type Error* if you missed it.

Question 12

```
debug-name.ts:3:13 - error TS2304: Cannot find name 'Goodbye'.
```

Line 3 leaves the quotes off `Goodbye`, so the checker treats it as a name, finds no such name, and points at column 13, where `Goodbye` starts. Put it in quotes:

debug-name-fixed.ts

```
1 console.log("Quiz time!");
2 console.log("Question 1");
3 console.log("Goodbye");
```

```
Quiz time!
Question 1
Goodbye
```

Reread *1.9 Names the Checker Does Not Know* if you missed it.

Question 13

B. `console` is a name the checker cannot find, and it looks very close to the known name `console`, so the message (code TS2552) adds the suggestion.

- **A** is wrong: `Ready` without quotes is an unknown name too, but nothing similar exists, so the message is TS2304 "Cannot find name 'Ready'." with no suggestion.
- **C** is wrong: that is a syntax error, TS1005 `' ) '` expected.
- **D** is wrong: that line is correct, and the checker prints nothing for it.

Reread *1.9 Names the Checker Does Not Know* if you chose differently.

Question 14

```
Step 1
Step 2: 10
.../day01/runtime-trace.ts:3
console.log("-".repeat(2 - 5));
                        ^
```

```
RangeError: Invalid count value: -3
```

The types fit (`2 - 5` is a number), so the program runs. Line 1 shows `Step 1`, and line 2 shows `Step 2: 10`. Line 3 works out `2 - 5`, which is `-3`, and `repeat` cannot make `-3` copies, so the program stops with a `RangeError`. Line 4 never runs, so `Step 3` never appears.

Reread *1.10 Reading a Runtime Error* if your output differed.

Question 15

C. `-4` is a number, so the types fit and `npx tsc` reports no error for it. The problem appears only when Node runs line 3: it shows lines 1 and 2, then stops with `RangeError: Invalid count value: -4`.

- **A** is wrong: `Done` without quotes is an unknown name, and `npx tsc` reports TS2304.
- **B** is wrong: the missing `)` is a syntax error, and `npx tsc` reports TS1005.
- **D** is wrong: a string where a number is required is a type error, TS2322.

Reread the comparison table in *1.10 Reading a Runtime Error* if you chose differently.

Question 16

One possible solution:

address.ts

```
1 console.log("Jordan Lee");
2 console.log("27 Willow Street");
3 console.log("Apartment 3");
4 console.log("Brookfield");
```

```
Jordan Lee
27 Willow Street
Apartment 3
Brookfield
```

Each line is one statement calling `console.log` with one string, so the program shows four lines, top to bottom, in the order written. Your text will differ; what matters is four `console.log` lines, quotes around every piece of text, and a semicolon at the end of each.

Reread *1.4 Printing Text and Numbers* if your program reported an error or showed the wrong text.

Question 17

One possible solution (adding up items, minutes in a day, a discount, sharing equally, and the average of three scores):

five-expressions.ts

```
1 console.log(12 + 7 + 5);
2 console.log(60 * 24);
3 console.log(100 - 3 * 15);
4 console.log(45 / 9);
5 console.log((8 + 6 + 10) / 3);
```

```
24
1440
55
5
8
```

- `100 - 3 * 15`: multiplication first, `3 * 15` is 45, then `100 - 45` is 55.
- `45 / 9` divides evenly, so it shows the whole number 5.
- In `(8 + 6 + 10) / 3`, the parentheses make the program add first (24), then divide by 3. Without them, only the 10 would be divided by 3.

Reread *1.4 Printing Text and Numbers* if a result surprised you.

Question 18

Here is each break applied to the sample solution.

Missing closing quote on line 2:

address-broken-quote.ts

```
1 console.log("Jordan Lee");
2 console.log("27 Willow Street);
3 console.log("Apartment 3");
4 console.log("Brookfield");
```

```
address-broken-quote.ts:2:32 - error TS1002: Unterminated string literal.
address-broken-quote.ts:3:1 - error TS1005: ',' expected.
address-broken-quote.ts:3:27 - error TS1005: ')' expected.
```

The first message is the real one: a **string literal** is a string typed directly into the code, and **unterminated** means it never ends. The string on line 2 has no closing quote, so it runs to the end of the line (column 32), swallowing the `)` and `;`. The two messages about line 3 are knock-on effects: the checker is still looking for the end of line 2's `console.log(`. Add the closing quote and all three disappear.

Misspelled console on line 3:

address-broken-name.ts

```
1 console.log("Jordan Lee");
2 console.log("27 Willow Street");
3 consol.log("Apartment 3");
4 console.log("Brookfield");
```

```
address-broken-name.ts:3:1 - error TS2552: Cannot find name 'consol'. Did you mean 'console'?
```

A name the checker cannot find, at line 3, column 1, with a suggestion because `consol` is close to `console`.

Text without quotes on line 4:

address-broken-bare.ts

```
1 console.log("Jordan Lee");
2 console.log("27 Willow Street");
3 console.log("Apartment 3");
4 console.log(Brookfield);
```

```
address-broken-bare.ts:4:13 - error TS2304: Cannot find name 'Brookfield'.
```

Without quotes, `Brookfield` is treated as a name. No such name exists and none looks similar, so the message is TS2304 with no suggestion; column 13 is where the word starts.

In all three cases nothing ran: `npx tsc` only checks. (Had you made all three breaks at once, the first run would have shown only the syntax errors from the missing quote; see 1.9.) Your line numbers and columns depend on where you broke your program, but the codes should match.

Reread 1.9 *Names the Checker Does Not Know* if your results differed.

Question 19

C. Line 1 works out 42 and throws it away, because nothing hands it to `console.log`. Line 2 shows `Done`.

- **A** is wrong: a value is shown only when it is logged, and line 1 has no `console.log`.
- **B** is wrong: there are no quotes and no `console.log`, so the characters are never shown.
- **D** is wrong: the line follows every rule and all its values are numbers, so the type checker accepts it.

Reread 1.4 *Printing Text and Numbers* if you chose differently.

Question 20

```
Movie night
Snacks: 7

Tickets: 18
2 * 9
```

Line	What happens
1	Line comment, skipped
2	Shows Movie night
3	Works out <code>3 + 4</code> , shows Snacks: 7; the comment after it is ignored
4	Commented out, so Cancelled never shows
5	<code>console.log()</code> shows a blank line
6	Block comment, skipped
7	Works out <code>2 * 9</code> , shows Tickets: 18
8	<code>"2 * 9"</code> is a string, so it shows <code>2 * 9</code>

Reread [1.4 Printing Text and Numbers](#) and [1.6 Comments: Notes the Computer Ignores](#) if your output differed.

Question 21

B. In P, Finished has no quotes, so the checker reports

program-p.ts:3:13 - error TS2304: Cannot find name 'Finished'. and Ana does not run it. In Q, every value has the right type (`-1` is a number), so the check prints nothing; running it shows A and B, then stops at line 3 with `RangeError: Invalid count value: -1`, so C never shows.

- **A** is wrong: the type checker checks kinds of values, not whether a count makes sense, so Q passes.
- **C** is wrong: it has the two programs swapped.
- **D** is wrong: P's unknown name is a type error, and Q's negative count stops it before line 4.

Reread the comparison table in [1.10 Reading a Runtime Error](#) if you chose differently.

Question 22

```
review-menu.ts:3:13 - error TS2304: Cannot find name 'Salad'.
```

Line 3's `Salad` has no quotes, so it is a name the checker cannot find. Column 13 is the first character after `console.log(`, which is 12 characters long. No known name looks like `Salad`, so there is no suggestion. Add the quotes (and a colon, to match line 2):

review-menu-fixed.ts

```
1 // Lunch menu
2 console.log("Soup:", 4);
3 console.log("Salad:", 6);
```

```
Soup: 4
Salad: 6
```

Reread *1.8 Reading a Type Error* and *1.9 Names the Checker Does Not Know* if your prediction differed.

Question 23

Model answer: It tells Jo that every value is used in a way its type allows, and that there are no unknown names or syntax errors. It does not tell Jo that the program does the right thing: the checker does not know what the program is meant to show, and it cannot see runtime errors or statements in the wrong order. Statements run in exactly the order written, so Jo should compare the output with what it should be, line by line, and reorder the statements.

A complete answer must:

- say that passing the check means only that the types fit (no type errors), and
- say that wrong order (or wrong arithmetic, or a runtime error) is found only by running the program and comparing its output with what it should be.

Reread *1.7 Two Steps: Check, Then Run* and *1.5 Programs Run Top to Bottom* if your reasoning differed.

Question 24

Model answer: (a), (c), and (d) are reported by `npx tsc`: (a) is a syntax error (the `(` is never closed, TS1005), (c) is TS2322 (a string where a number is required), and (d) is TS2304, an unknown name, because `Hi` has no quotes. Only (b) gets past the checker, because `3 - 10` is a number; Node reports it while running, with a `RangeError` for the count `-7`.

You can tell them apart by the form of the message: the type checker prints `file.ts:line:column - error TS` followed by a number, and nothing from the program has been shown. A runtime error comes from `node`, after any earlier output, and shows the code line with a caret and a line naming the error, such as `RangeError: ...`.

A complete answer must:

- put (a), (c), and (d) under `npx tsc` and (b) under `node`, with a reason for (b), and
- describe at least one difference between the two kinds of message (the `TS` code and `line:column` versus the caret and `...Error: line`, or no output versus earlier output).

Reread *1.10 Reading a Runtime Error* if your sorting differed.