Module 2 of 16

Values, types and coercion

The seven primitives and typeof, numbers as doubles with NaN and BigInt, strings and their toolkit, truthiness and the rules behind == and +, the nullish operators, and let/const/var with the temporal dead zone.

Most JavaScript bugs that make it into production are type bugs: a count that was zero and therefore "missing", a string compared to a number, an emoji counted as two characters, a var shared by every callback in a loop. None of them are random — each follows from a rule this module states.

It begins with the eight kinds of value and the two typeof traps, then takes the number type seriously: doubles, NaN, the safe-integer limit and BigInt. Strings get their toolkit and their Unicode footnote. The coercion lesson lays out the eight falsy values, the algorithm behind ==, what + does when a string is involved, and the modern operators — ??, ?., ??= — that make the common cases safe. The module closes with let, const and var: block scope, hoisting, the temporal dead zone, and the loop-closure classic that justifies let.

The exercises make the rules visible: programs that classify values the way the engine does, tables of what == and + produce, and code that demonstrates the temporal dead zone by catching the error it throws.

CONTENTS

1	The seven primitives, objects and typeof	12 min
2	Numbers — doubles, NaN, safe integers and BigInt	14 min
3	Strings — immutable text, template literals and the method toolkit	14 min
4	Truthiness, coercion and the two equalities	14 min
5	let, const and var — scope, hoisting and the temporal dead zone	13 min
✓	Checkpoint — Values, types and coercion	24 min

Exercises in this module compile on Node 16. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

The seven primitives, objects and typeof

JavaScript has exactly eight kinds of value: seven **primitives** — `undefined`, `null`, `boolean`, `number`, `bigint`, `string`, `symbol` — and **object**, which covers everything else: plain objects, arrays, functions, dates, regexes, maps. Every surprise about "types" in this language comes from three facts: variables have no type (only values do), primitives are copied by value while objects are shared by reference, and the `typeof` operator has two historical quirks. This lesson lays the type system out plainly so the coercion rules in the following lessons have something to stand on.

Primitives are values; objects are references

```
let a = "hi";  let b = a;  b += "!";       // a is still "hi": strings are co
let o = { n: 1 }; let p = o; p.n = 2;      // o.n is 2: both names point at o
```

A primitive is **immutable** — you never change a string or a number, you make a new one — and comparing two primitives with `===` compares their contents. An object is a **mutable thing with an identity**; two object variables are `===` only if they refer to the same object, and `{} === {}` is `false`. This single distinction explains function-argument behaviour (module 3), copying (module 4) and why `const o = {}` still lets you change `o.n`.

`typeof`, and its two lies

```
typeof undefined   // "undefined"
typeof true        // "boolean"
typeof 42          // "number"
typeof 42n         // "bigint"
typeof "s"         // "string"
typeof Symbol()    // "symbol"
typeof {}          // "object"
typeof []          // "object"     ← arrays are objects; use Array.isArray
typeof null        // "object"     ← a bug from 1995, kept for compatibility
typeof (() => 1)   // "function"   ← functions are objects, but typeof singles
typeof notDeclared // "undefined"  ← the one operator that does not throw on u
```

Remember the two exceptions: **`typeof null` is `"object"`** (test null with `=== null`), and **arrays report `"object"`** (test with `Array.isArray(x)`). `typeof` on an undeclared identifier returns `"undefined"` instead of throwing, which is why feature-detection code writes `typeof window !== "undefined"`.

`undefined` versus `null`

Both mean "no value", and the language uses them differently. **`undefined`** is what the engine produces: an uninitialised variable, a missing property, a missing argument, a function with no `return`, an out-of-range array index. **`null`** is what *you* write to say "deliberately empty": a search that found nothing, an optional field that is absent. `undefined == null` is `true` (the one useful `==`), `undefined === null` is `false`. The **nullish** operators treat them alike: `x ?? d` supplies `d` for either; `a?.b` short-circuits on either. Style rule: return `null` for "no result" from your own functions; check for both with `== null` or `??`.

`number` and `bigint`

There is one `number` type — a 64-bit IEEE-754 double — for integers and fractions alike: `1` and `1.0` are the same value, `0.1 + 0.2 !== 0.3`, and integers are exact only up to `Number.MAX_SAFE_INTEGER` ($2^{53} - 1 \approx 9 \times 10^{15}$). `bigint` (2020) is arbitrary precision for whole numbers: `123n`, `BigInt("9007199254740993")`; it never mixes implicitly with `number` (`1n + 1` is a `TypeError`), has no fractions, and cannot be used with `Math`. Both get a full lesson next.

`string`

Immutable sequences of UTF-16 code units, in single, double or backtick quotes. Backticks make **template literals**: `` `total: ${a + b}` `` interpolates any expression and may span lines. There is no separate character type — `"a"` is a string of length 1 — and `s[0]` or `s.at(-1)` read units, not necessarily whole characters (module on strings).

`symbol`

A guaranteed-unique value created by `Symbol("description")`, used as a property key that cannot collide with any string key. You will mostly meet the **well-known symbols** the language

defines — `Symbol.iterator` (what `for…of` looks for, module 6), `Symbol.asyncIterator`, `Symbol.toPrimitive` — rather than create your own.

Wrapper objects and autoboxing

`"abc".toUpperCase()` works although `"abc"` is a primitive because the engine temporarily wraps it in a `String` object to look up the method, then discards the wrapper. You never need `new String("x")` or `new Number(1)` — they create *objects*, so `new Number(1) === 1` is `false` and `typeof new String("x")` is `"object"`. Use `String(x)`, `Number(x)`, `Boolean(x)` **without** `new` as conversion functions; never the constructors.

Checking types in practice

Question	Write
is it a string / number / boolean?	`typeof x === "string"` …
is it an array?	`Array.isArray(x)`
is it `null` or `undefined`?	`x == null` (or `x === null`, `x === undefined` separately)
is it a function?	`typeof x === "function"`
is it a plain object?	`typeof x === "object" && x !== null && !Array.isArray(x)`
is it a specific class instance?	`x instanceof Date`
is it a usable number?	`typeof x === "number" && Number.isFinite(x)` — `NaN` is a number too

Interview angle

"What are JavaScript's data types?" Seven primitives — undefined, null, boolean, number, bigint, string, symbol — plus object.

"What is `typeof null`?" `"object"` — a historical bug; check with `=== null`.

"`undefined` versus `null`?" Engine-produced absence versus deliberate emptiness; `==` treats them equal, `===` does not; `??` and `?.` handle both.

"Primitive versus object?" Immutable, compared and copied by value versus mutable, compared and shared by identity.

"How do you tell an array from an object?" `Array.isArray` — `typeof` says `"object"` for both.

Key takeaways

Seven primitives + object; values have types, variables do not.

Primitives: immutable, by value. Objects: mutable, by reference; `{} !== {}`.

`typeof null === "object"` and `typeof [] === "object"` are the two traps; `Array.isArray`, `=== null`.

`undefined` is the engine's absence, `null` is yours; `??` and `?.` treat them alike.

Conversion functions `String(x)` / `Number(x)` / `Boolean(x)` — never the constructors with `new`.

EXERCISE 1

typeof census

Read `n` tokens and turn each into the value it names using the provided `parse` helper (`null`, `undefined`, `NaN`, booleans, numbers, `7n` bigints, `empty` for the empty string, `[…]` / `{…}` as JSON, anything else a string). Print `<token>: typeof=<typeof value> array=<Array.isArray> nullish=<value == null>`.

Example: `4` then `null [1,2] 7n hi` →

```
null: typeof=object array=false nullish=true
[1,2]: typeof=object array=true nullish=false
7n: typeof=bigint array=false nullish=false
hi: typeof=string array=false nullish=false
```

INPUT 1

```
4
null [1,2] 7n hi
```

OUTPUT 1

```
null: typeof=object array=false
nullish=true
[1,2]: typeof=object array=true
nullish=false
7n: typeof=bigint array=false
nullish=false
hi: typeof=string array=false
nullish=false
```

INPUT 2

```
3
undefined NaN true
```

OUTPUT 2

```
undefined: typeof=undefined array=false
nullish=true
NaN: typeof=number array=false
nullish=false
true: typeof=boolean array=false
nullish=false
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
// Turn a token into the value it names: null, undefined, NaN, booleans, numbers, bi
// empty string), [] and {} as JSON, anything else as a plain string.
function parse(t) {
  if (t === "null") return null;
  if (t === "undefined") return undefined;
  if (t === "NaN") return NaN;
  if (t === "true") return true;
  if (t === "false") return false;
  if (t === "empty") return "";
  if (/^-?\d+n$/.test(t)) return BigInt(t.slice(0, -1));
  if (t.startsWith("[") || t.startsWith("{")) { try { return JSON.parse(t); } catch
  if (t.trim() !== "" && Number.isFinite(Number(t))) return Number(t);
  return t;
}
const n = Number(tokens[0]);
for (let i = 1; i <= n; i++) {
  const v = parse(tokens[i]);
  console.log(`${tokens[i]}: typeof=${typeof v} array=${Array.isArray(v)} nullish=${
}
```

EXERCISE 2

Walk a path, nullishly

Read three lines: a JSON object, a dotted path (`a.b.c`), and a fallback word. Walk the path with optional chaining — `cur = cur?.[key]` — so that a `null` or `undefined` anywhere along the way yields `undefined` instead of throwing. Print `value=<JSON of the value, or undefined>`, `withDefault=<JSON of value ?? fallback>` and `typeof=<typeof value>`. Note that `??` replaces `null` as well as `undefined`.

Example: `{"a":{"b":null,"c":[1,2]}}` / `a.b.c` / `x` →

```
value=undefined
withDefault="x"
typeof=undefined
```

INPUT 1

```
{"a":{"b":null,"c":[1,2]}}
a.b.c
x
```

OUTPUT 1

```
value=undefined
withDefault="x"
typeof=undefined
```

INPUT 2

```
{"a":null}
a
d
```

OUTPUT 2

```
value=null
withDefault="d"
typeof=object
```

REFERENCE SOLUTION

```
"use strict";
const [objLine, pathLine, fallback] = require("fs").readFileSync(0, "utf8").split("\
const root = JSON.parse(objLine);
const keys = pathLine.trim().split(".");
let cur = root;
for (const k of keys) {
  cur = cur?.[k];               // optional chaining: undefined once anything along
}
console.log(`value=${cur === undefined ? "undefined" : JSON.stringify(cur)}`);
console.log(`withDefault=${JSON.stringify(cur ?? fallback.trim())}`);
console.log(`typeof=${typeof cur}`);
```

QUIZ

1. `typeof null` is…

A. `"null"`

B. `"object"`

C. `"undefined"`

D. A `TypeError`

2. `typeof []` is…

A. `"array"`

B. `"object"` — use `Array.isArray`

C. `"list"`

D. `"function"`

3. Predict: `let a = { n: 1 }; let b = a; b.n = 2; console.log(a.n)`

A. `1`

B. `2` — objects are shared by reference

C. `undefined`

D. Throws

4. `undefined == null` and `undefined === null` are…

A. `true`, `true`

B. `true`, `false`

C. `false`, `false`

D. `false`, `true`

5. `new Number(1) === 1` is…

A. `true`

B. `false` — the constructor creates an object; use `Number(x)` without `new`

C. A `TypeError`

D. `undefined`

ANSWER KEY · THE SEVEN PRIMITIVES, OBJECTS AND TYPEOF

1. B A 1995 bug preserved for compatibility; test with `=== null`.

2. B Arrays are objects; only functions get their own `typeof` result.

3. B Primitives are copied; objects are not.

4. B Loose equality treats the two nullish values as equal — the one idiomatic `==`.

5. B Wrapper objects are never what you want; the conversion functions are.

LESSON 2 · 14 MIN

Numbers — doubles, NaN, safe integers and BigInt

Every JavaScript `number` is a 64-bit floating-point value. There is no `int`, no `long`, no `decimal` — `7 / 2` is `3.5`, `2 ** 53 + 1` is `2 ** 53`, and `0.1 + 0.2` is `0.30000000000000004`. Most number bugs in the language follow from forgetting one of those three facts. This lesson covers the representation, the special values (`NaN`, the infinities, `-0`), the conversion functions and their differences, integer arithmetic and its limit, the `Math` toolkit, and `BigInt` for when 2^{53} is not enough.

One representation: IEEE-754 double

A double has 53 bits of significand, so it represents every integer up to `2 ** 53 = 9007199254740992` exactly and then starts skipping. `Number.MAX_SAFE_INTEGER` ($2^{53} - 1$) is the last integer where `n + 1 !== n` is guaranteed; `Number.isSafeInteger(x)` checks. Fractions are binary fractions, so decimal values like 0.1 are approximations — `0.1 + 0.2 === 0.3` is `false`, and `(0.1 + 0.2).toFixed(2)` is the honest way to print it. Compare doubles with a tolerance (`Math.abs(a - b) < Number.EPSILON * k` or a domain-appropriate epsilon), or work in integers (cents, not dollars).

The special values

`Infinity` and `-Infinity`: `1 / 0`, overflow (`1e308 * 10`). `Number.isFinite(x)` rejects them and `NaN`.

`NaN` (not a number): the result of any undefined operation — `0 / 0`, `Math.sqrt(-1)`, `Number("abc")`, `undefined + 1`. It is the only value in the language not equal to itself: `NaN === NaN` is `false`, so test with **`Number.isNaN(x)`** (strict: is it the value NaN?) — not the global `isNaN(x)`, which coerces first and reports `isNaN("abc")` as `true`. `Object.is(x, NaN)` also works.

`-0`: `-0 === 0` is `true` and `Object.is(-0, 0)` is `false`; `1 / -0` is `-Infinity`. It appears from `-1 * 0` or `Math.round(-0.4)` and matters almost never — except in `Object.is` and in printing (`String(-0)` is `"0"`).

Converting to number

From `"42px"`	From `""`	From `" 12 "`	From `"0x1f"`	From `"1e3"`	From `null`	From `undefined`	From `[]`	From `true`
`Number(...)` → `NaN`	`0`	`12`	`31`	`1000`	`0`	`NaN`	`0`	`1`
`parseInt(..., 10)` → `42`	`NaN`	`12`	`0` (radix 10)	`1`	`NaN`	`NaN`	`NaN`	`NaN`
`parseFloat(...)` → `42`	`NaN`	`12`	`0`	`1000`	`NaN`	`NaN`	`NaN`	`NaN`

`Number(x)` (identical to unary `+x`) converts the *whole* string or fails with `NaN`; `parseInt` / `parseFloat` read the longest valid **prefix** and ignore the rest. Always pass the radix to `parseInt` — `parseInt("08")` is `8` today but was `0` in old engines, and `parseInt("0x1f")` with no radix reads hex. For user input, `Number` plus a `Number.isFinite` check is the safe default; `parseInt` is for "the number at the start of this string".

Integer arithmetic, and where it lies

`+ - *` on integers are exact below 2^{53}. Division is not integer division: `7 / 2` is `3.5`; use `Math.floor(a / b)` (toward −∞), `Math.trunc(a / b)` (toward 0) or `~~(a / b)` (32-bit truncation, avoid). `%` is a **remainder** with the sign of the dividend: `-7 % 3` is `-1`; a true modulo is `((a % b) + b) % b`. `**` is exponentiation (`2 ** 10`); `Math.pow` is the same. Above 2^{53}, `+` silently loses precision — `9007199254740992 + 1` is `9007199254740992` — which is why IDs and money in large systems arrive as strings and why `BigInt` exists.

Bitwise operators (`& | ^ ~ << >> >>>`) work on 32-bit integers: the operand is converted to a signed 32-bit int first, so `2 ** 31 | 0` is `-2147483648` and `(2 ** 32) | 0` is `0`. Useful for flags and for the `x | 0` truncation idiom on small values; a trap for anything larger.

The `Math` toolkit

`Math.floor/ceil/round/trunc` (note `Math.round(-2.5)` is `-2` — rounds half toward +∞, unlike Java), `Math.abs`, `Math.min/max(...values)` (spread an array: `Math.max(...arr)`; empty gives `-Infinity`), `Math.sqrt`, `Math.hypot`, `Math.pow`, `Math.sign`, `Math.log2/log10`, `Math.random()` (0 ≤ r < 1, not cryptographic — `crypto.randomInt` for that), `Math.PI`, `Math.E`.

Formatting

`x.toFixed(d)` gives a string with `d` decimals (rounding is by decimal expansion; `(1.005).toFixed(2)` is `"1.00"` because 1.005 is really 1.00499…). `x.toPrecision(n)`, `x.toString(2)` (binary), `x.toString(16)`, `Number.parseFloat`. `toLocaleString("en-US")` for grouping. Large and tiny numbers print in exponent form (`1e21`, `1e-7`) — a surprise for money code that forgot to fix decimals.

BigInt

```
const big = 9007199254740993n;              // literal with n
const fromString = BigInt("123456789012345678901234567890");
big + 1n; big * 2n; big / 2n;                // / truncates toward zero: 7n / 2
big ** 20n;
typeof big;                                  // "bigint"
Number(big);                                 // back to double — may lose preci
big + 1                                      // TypeError: cannot mix BigInt ar
big > 1                                      // comparisons across the types ar
JSON.stringify({ big })                      // TypeError — BigInt has no JSON
```

Arbitrary precision, integers only, no `Math`, explicit conversion at every boundary. Use it for factorials, 64-bit IDs, cryptography-sized arithmetic; use `number` for everything else — it is far faster.

Interview angle

> *"Why is `0.1 + 0.2 ≠ 0.3`?"* Binary doubles cannot represent decimal tenths exactly; compare with a tolerance or use integer units.
>
> *"How do you check for `NaN`?"* `Number.isNaN(x)` — `NaN ≠ NaN`, and global `isNaN` coerces.

"`Number("42px")` versus `parseInt("42px")`?" `NaN` versus `42` — whole-string versus prefix parsing.

"Largest exact integer?" `Number.MAX_SAFE_INTEGER` = $2^{53} - 1$; beyond it use `BigInt`.

"What is `-7 % 3`?" `-1` — remainder keeps the dividend's sign; `((a % b) + b) % b` for a true modulo.

Key takeaways

One `number` type: a double. Exact integers to $2^{53} - 1$; decimal fractions are approximate.

`NaN` is unequal to itself — `Number.isNaN`; `Infinity` from overflow and `/ 0`; `-0` exists.

`Number` / `+x` converts whole strings (or `NaN`); `parseInt(s, 10)` / `parseFloat` read a prefix.

`/` is floating; `Math.floor` / `trunc` for integer division; `%` keeps the dividend's sign; bitwise ops are 32-bit.

`BigInt` for big whole numbers; never mixes with `number` implicitly; no `Math`, no JSON.

EXERCISE 1

Safe or not

Read `n` tokens. For each, convert with `Number` and print `<token>: value=<v> finite=<Number.isFinite> safeInteger=<Number.isSafeInteger>`; when the token is a plain integer literal (`/^-?\d+$/`) also append `bigint=<BigInt(token)>` — which keeps the digits the double lost.

Example: `3` then `42 9007199254740993 abc` →

```
42: value=42 finite=true safeInteger=true bigint=42
9007199254740993: value=9007199254740992 finite=true safeInteger=false bigint=900719
abc: value=NaN finite=false safeInteger=false
```

INPUT 1

```
3
42 9007199254740993 abc
```

OUTPUT 1

```
42: value=42 finite=true
safeInteger=true bigint=42
9007199254740993: value=9007199254740992
finite=true safeInteger=false
bigint=9007199254740993
abc: value=NaN finite=false
safeInteger=false
```

INPUT 2

```
2
4.5 1e3
```

OUTPUT 2

```
4.5: value=4.5 finite=true
safeInteger=false
1e3: value=1000 finite=true
safeInteger=true
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
for (let i = 1; i ⩽ n; i++) {
  const t = tokens[i];
  const v = Number(t);
  const parts = [`value=${v}`, `finite=${Number.isFinite(v)}`, `safeInteger=${Number
  if (/^-?\d+$/.test(t)) parts.push(`bigint=${BigInt(t).toString()}`);
  console.log(`${t}: ${parts.join(" ")}`);
}
```

EXERCISE 2

Division, five ways

Read `n` pairs `a b` (`b ≠ 0`). For each print `<a>/<b>: div=<a / b> floor=<Math.floor> trunc=<Math.trunc> rem=<a % b> mod=<((a % b) + b) % b>`. Watch the negatives: floor and trunc disagree, and `%` keeps the dividend's sign.

Example: `2` then `7 2`, `-7 2` →

```
7/2: div=3.5 floor=3 trunc=3 rem=1 mod=1
-7/2: div=-3.5 floor=-4 trunc=-3 rem=-1 mod=1
```

INPUT 1

```
2
7 2
-7 2
```

OUTPUT 1

```
7/2: div=3.5 floor=3 trunc=3 rem=1 mod=1
-7/2: div=-3.5 floor=-4 trunc=-3 rem=-1
mod=1
```

INPUT 2

```
1
-7 3
```

OUTPUT 2

```
-7/3: div=-2.3333333333333335 floor=-3
trunc=-2 rem=-1 mod=2
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
for (let i = 0; i < n; i++) {
  const a = Number(tokens[1 + 2 * i]), b = Number(tokens[2 + 2 * i]);
  const mod = ((a % b) + b) % b;
  console.log(`${a}/${b}: div=${a / b} floor=${Math.floor(a / b)} trunc=${Math.trunc
}
```

QUIZ

1. `0.1 + 0.2 === 0.3` is…

A. `true`

B. `false` — binary doubles cannot represent decimal tenths exactly

C. A `TypeError`

D. `true` only in strict mode

2. The correct test for `NaN` is…

A. `x === NaN`

B. `Number.isNaN(x)`

C. `x == NaN`

D. `isNaN(x)` — always

3. `Number.MAX_SAFE_INTEGER` is…

A. $2^{31} - 1$

B. $2^{53} - 1$ — the last integer that `n + 1` distinguishes

C. 2^{64}

D. Infinity

4. `1n + 1` evaluates to…

A. `2n`

B. `2`

C. A `TypeError` — BigInt and Number never mix implicitly

D. `"11"`

5. `-7 % 3` is…

A. `2`

B. `-1` — the remainder takes the dividend's sign

C. `1`

D. `-2`

ANSWER KEY · NUMBERS — DOUBLES, NAN, SAFE INTEGERS AND BIGINT

1. B Compare with a tolerance or compute in integer units.

2. B `NaN ≠ NaN`; global `isNaN` coerces (`isNaN("abc")` is `true`).

3. B Doubles have a 53-bit significand; beyond it use `BigInt`.

4. C Convert one side explicitly: `1n + BigInt(1)` or `Number(1n) + 1`.

5. B `((a % b) + b) % b` gives the mathematical modulo.

LESSON 3 · 14 MIN

Strings — immutable text, template literals and the method toolkit

Strings are the most-used values in most programs, and JavaScript's are simple in shape — immutable sequences of UTF-16 code units — with a rich, mostly consistent method set. The things that go wrong are predictable: forgetting immutability (`s.toUpperCase()` with no assignment), confusing `slice` and `substring` and `substr`, treating `length` as a character count when emoji are present, and reaching for the wrong search method. This lesson is the toolkit, organised by what you want to do, with the traps marked.

Literals and template literals

```
const a = 'single', b = "double";                  // identical; pick one style
const t = `sum: ${1 + 2}, upper: ${a.toUpperCase()}`;   // template literal: c
const multi = `line one
line two`;                                          // newlines are kept as wr
const esc = "tab\tnewline\nquote\"backslash\\ unicodeé";
```

Template literals also allow **tagged templates** — `tag`text ${x}`` calls `tag` with the literal parts and values (used by libraries for SQL, GraphQL, styled components) — a topic for the strings module later. `String.raw` is one such tag that leaves escapes alone.

Immutability

Every "modifying" method returns a new string; the original never changes.

```
let s = "hello";
s.toUpperCase();            // returns "HELLO", s is still "hello"
s = s.toUpperCase();        // now s is "HELLO"
s[0] = "j";                 // silently ignored (throws in strict mode? no — c
```

Because they are immutable, strings are safe to share, compare by value with `===`, and use as object keys; and building a long string in a loop with `+=` is fine in modern engines (they use ropes internally), unlike Java — `array.join` is still clearer for many pieces.

Indexing, length and Unicode

`s.length` counts **UTF-16 code units**, `s[i]` and `s.charAt(i)` read one unit, `s.at(-1)` reads from the end (2022). Characters outside the Basic Multilingual Plane — emoji, some CJK, mathematical symbols — take **two** units:

```
"héllo".length        // 5
"😅".length           // 2  — one character, two code units (a surrogate pair
[..."😅a"].length     // 2  — spreading iterates by code point
"😅".codePointAt(0)   // 128512
String.fromCodePoint(128512)   // "😅"
```

Iterate with `for (const ch of s)` or `[...s]` when characters matter; `s.split("")` splits by unit and corrupts emoji. `normalize("NFC")` before comparing text that may contain composed versus decomposed accents.

The toolkit, by task

Search

```
s.includes("lo")           // boolean
s.indexOf("l"), s.lastIndexOf("l")   // -1 when absent
s.startsWith("he"), s.endsWith("lo")
s.search(/l+/)             // regex, index or -1
s.match(/l/g)              // ["l", "l"] or null;   s.matchAll(/l/g) for itera
```

Slice

```
s.slice(1, 3)      // "el"   — end exclusive; negatives count from the end: s.
s.substring(1, 3)  // "el"   — swaps arguments if start > end; negatives becor
s.substr(1, 2)     // deprecated (start, length) — do not use
```

Transform

```
s.toUpperCase(), s.toLowerCase()          // toLocaleUpperCase for locale rules
s.trim(), s.trimStart(), s.trimEnd()
s.padStart(8, "0"), s.padEnd(8)           // "000hello"
s.repeat(3)
s.replace("l", "L")                       // FIRST occurrence only when given a
s.replaceAll("l", "L")                    // every occurrence (2021); or s.repl
s.split(","), s.split(/\s+/), s.split("", 3)
"a,b".concat("!")                         // same as + ; rarely used
```

The `replace`-with-a-string-replaces-once rule is the trap people hit weekly; `replaceAll` or a `/g` regex is the fix. In the replacement string `$&` is the match and `$1` a group; a function replacer `(match, g1) ⇒ …` computes the replacement.

Compare

```
"a" < "b"                         // true — code unit order; "Z" < "a" is true,
"ä".localeCompare("z", "de")      // -1: locale-aware order for sorting user-vi
arr.sort((x, y) ⇒ x.localeCompare(y))
```

Default `<` / `>` on strings compares code units, so upper case sorts before lower case and digit strings compare lexically. For human-facing sort use `localeCompare`, optionally with `{ numeric: true }` to make `"file10"` follow `"file9"`.

Conversions

`String(x)` handles every value (`String(null)` is `"null"`, `String([1,2])` is `"1,2"`, `String({})` is `"[object Object]"`); `x.toString()` fails on `null` / `undefined`; `` `${x}` `` is `String(x)`; `JSON.stringify(x)` gives a JSON encoding (quotes around strings, `"[1,2]"` for arrays). Number to string with a radix: `(255).toString(16)` is `"ff"`. String to number: previous lesson.

Building strings

For a few pieces, template literals. For many pieces in a loop, either `+=` (fine in V8) or push to an array and `join` — the join form makes the separator explicit and avoids the trailing-separator problem. `Array(n).fill("-").join("")` or `"-".repeat(n)` for runs.

Interview angle

"Are strings mutable?" No — every method returns a new string; assign the result.

"`slice` versus `substring` versus `substr`?" `slice` supports negatives and is the one to use; `substring` swaps and clamps; `substr` is deprecated.

"Why is `\"😀\".length` 2?" UTF-16 code units; a surrogate pair. Iterate with `for…of` or spread for code points.

"How do you replace all occurrences?" `replaceAll` or `replace` with a `/g` regex — plain `replace(string)` does one.

"Why does `["b", "a", "B"].sort()` put `B` first?" Code-unit order; use `localeCompare` for human sorting.

Key takeaways

Immutable UTF-16 sequences; template literals interpolate any expression.

`length` /indexing count code units; emoji are two; `for…of` /spread iterate characters.

`slice` (negatives ok), never `substr`; `includes` / `startsWith` / `indexOf` to search; `replaceAll` for every match.

`<` compares code units; `localeCompare` for people; `{ numeric: true }` for `file10`.

`String(x)` converts anything; `join` an array for many pieces.

EXERCISE 1

Code units and characters

Read one line. Print `length=<s.length> codePoints=<[...s].length>`, then `reversed=<the characters reversed>` — reverse the **spread** array so surrogate pairs stay intact — then `upper=<s.toUpperCase()>` and `first=<first character> last=<last character>` (empty for an empty line).

Example: `héllo 😀` →

```
length=8 codePoints=7
reversed=😀 olléh
upper=HÉLLO 😀
first=h last=😀
```

INPUT 1

```
héllo 😀
```

OUTPUT 1

```
length=8 codePoints=7
reversed=😀 olléh
upper=HÉLLO 😀
first=h last=😀
```

INPUT 2

```
abc
```

OUTPUT 2

```
length=3 codePoints=3
reversed=cba
upper=ABC
first=a last=c
```

REFERENCE SOLUTION

```
"use strict";
const s = require("fs").readFileSync(0, "utf8").replace(/\r?\n$/, "");
const chars = [...s];
console.log(`length=${s.length} codePoints=${chars.length}`);
console.log(`reversed=${chars.reverse().join("")}`);
console.log(`upper=${s.toUpperCase()}`);
console.log(`first=${[...s][0] ?? ""} last=${[...s].at(-1) ?? ""}`);
```

EXERCISE 2

Replace, count, pad

Read lines until end of input (skip empty lines). For each line `s` print four lines: `once=<s.replace("a", "_")>`, `all=<s.replaceAll("a", "_")>`, `words=<number of whitespace-separated words after trimming>` and `padded=<s.trim().padStart(12, ".")>`.

Example: `banana` →

```
once=b_nana
all=b_n_n_
words=1
padded=......banana
```

INPUT 1

```
banana
```

OUTPUT 1

```
once=b_nana
all=b_n_n_
words=1
padded=......banana
```

INPUT 2

```
  a cat
```

OUTPUT 2

```
once=  _ cat
all=  _ c_t
words=2
padded=.......a cat
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l =
for (const s of lines) {
  console.log(`once=${s.replace("a", "_")}`);
  console.log(`all=${s.replaceAll("a", "_")}`);
  const words = s.trim() === "" ? 0 : s.trim().split(/\s+/).length;
  console.log(`words=${words}`);
  console.log(`padded=${s.trim().padStart(12, ".")}`);
}
```

QUIZ

1. Predict: `let s = "hi"; s.toUpperCase(); console.log(s)`

A. `HI`

B. `hi` — strings are immutable; the result was discarded

C. `undefined`

D. Throws

2. `"😀".length` is…

A. `1`

B. `2` — one character, two UTF-16 code units

C. `4`

D. `0`

3. `"a-b-c".replace("-", "+")` gives…

A. `a+b+c`

B. `a+b-c` — only the first match

C. `a-b-c`

D. An error

4. Which slicing method should you avoid?

A. `slice`

B. `substring`

C. `substr` — deprecated `(start, length)` form

D. `at`

5. `["b", "a", "B"].sort()` gives…

A. `["a", "b", "B"]`

B. `["B", "a", "b"]` — code-unit order puts upper case first

C. `["a", "B", "b"]`

D. `["b", "B", "a"]`

ANSWER KEY · STRINGS — IMMUTABLE TEXT, TEMPLATE LITERALS AND THE METHOD TOOLKIT

1. B Assign the result: `s = s.toUpperCase()`.

2. B Spread or `for…of` to count characters.

3. B `replaceAll` or a `/g` regex for every occurrence.

4. C `slice` handles negatives and is the standard choice.

5. B Use `localeCompare` for human ordering.

LESSON 4 · 14 MIN

Truthiness, coercion and the two equalities

JavaScript will happily compare a string to a number, add a number to a string, and treat an empty array as true and an empty string as false. The rules behind those conversions are precise, small in number, and mostly ignored — which is why `==` has a reputation and `"1" + 1` is a meme. This lesson states the rules: what is falsy, how `==` decides, what `+` does, what the comparison operators do to mixed types, and the modern operators (`??`, `?.`, `||=`) that let you write the common cases without thinking about any of it.

Truthy and falsy

Exactly **eight** values are falsy: `false`, `0`, `-0`, `0n`, `""`, `null`, `undefined`, `NaN`. Everything else is truthy — including `"0"`, `"false"`, `[]`, `{}`, and any function. Conditions (`if`, `while`, `? :`, `!`, `&&`, `||`) apply this rule, and `Boolean(x)` or `!!x` makes it explicit.

```
if ([]) console.log("an empty array is truthy");     // prints
if ("") console.log("never");
if ("0") console.log("a non-empty string is truthy"); // prints
```

The trap: `if (count)` is wrong when `0` is a valid count; write `if (count !== undefined)` or `if (count != null)`.

`&&`, `||`, `??` return operands, not booleans

`a && b` returns `a` if `a` is falsy, else `b`. `a || b` returns `a` if truthy, else `b`. Both short-circuit — the right side is not evaluated when the left decides. This is why they double as control flow:

```
const name = input || "anonymous";        // default — but "" and 0 are replac
const name2 = input ?? "anonymous";       // nullish coalescing (2020): only r
user && user.save();                      // call only if user is truthy
user?.save();                             // optional call (2020): only if use
config.retries ??= 3;                     // logical assignment (2021): set ij
flags.verbose ||= false;  cache.hit &&= true;
```

Use `??` for defaults whenever `0`, `""` or `false` are legitimate values — which is most of the time. `?.` works for properties (`a?.b`), computed access (`a?.[k]`) and calls (`f?.()`), and short-circuits the *rest of the chain*: `a?.b.c` is `undefined` when `a` is nullish, without evaluating `.c`.

`===` and `==`

Strict equality `===`: same type and same value; no conversion. Objects by identity. `NaN !== NaN`; `+0 === -0`. `Object.is` is `===` except that it distinguishes `-0` and treats `NaN` as equal to itself.

Loose equality `==` converts operands by these rules, in order:

Same type → same as `===`.

`null == undefined` → `true` (and neither equals anything else).

number vs string → convert the string to a number: `1 == "1"`.

boolean vs anything → convert the boolean to a number: `true == 1`, `true == "1"`, `false == ""`, `false == "0"`.

object vs primitive → convert the object with `valueOf` / `toString`: `[1] == 1`, `[] == ""`, `[] == 0`, `{} == "[object Object]"`.

Hence the famous table: `"" == 0` (true), `"0" == 0` (true), `"" == "0"` (false — same type, different value), `null == 0` (false — rule 2 stops null), `[] == ![]` (true — `![]` is `false`, `false` → `0`, `[]` → `""` → `0`). None of this is random; all of it is avoidable: **use** `===`, and the single idiomatic `==`: `x == null` to test for null-or-undefined.

Relational operators

`< > ⩽ ⩾` convert to numbers **unless both sides are strings**, in which case they compare code units:

```
"10" < "9"        // true — string comparison
10 < "9"          // false — "9" becomes 9
"abc" < 5         // false — "abc" becomes NaN, and every comparison with NaN
null < 1          // true — null becomes 0
undefined < 1     // false — undefined becomes NaN
```

Sorting an array of numeric strings with `<` or the default `sort()` gives `["1", "10", "2", "9"]`; convert or pass a comparator.

`+` is the odd one out

If **either** operand is a string, `+` concatenates; otherwise it adds numbers. Every other arithmetic operator (`- * / %`) converts to numbers.

```
1 + "2"        // "12"
"3" - 1        // 2
"3" * "4"      // 12
1 + 2 + "3"    // "33" — left to right: 3 + "3"
"1" + 2 + 3    // "123"
[] + {}        // "[object Object]" — both become strings
+"42"          // 42 — unary plus converts
```

The conversion algorithm, briefly

Converting an object to a primitive calls `Symbol.toPrimitive` if present, else `valueOf` then `toString` (for numeric contexts) or `toString` then `valueOf` (for string contexts). `Date` prefers strings in `+`; arrays' `toString` is `join(",")`; plain objects give `"[object Object]"`. You can define `toString` / `valueOf` on your own classes to control this — rarely wise, occasionally elegant (a `Money` class that formats itself).

Conditions that read well

```
if (list.length === 0)            // not: if (!list.length)  — explicit intent
if (value == null)                // the one accepted ==
if (typeof x === "string" && x !== "")
const port = Number(process.env.PORT ?? 3000);
const label = user?.profile?.displayName ?? user?.email ?? "guest";
```

Reviewers read `===` as competence and `==` as a bug unless it is the null check. Reach for `??` over `||` unless you specifically want to replace empty strings and zeros.

Interview angle

"What is falsy in JavaScript?" `false, 0, -0, 0n, "", null, undefined, NaN` — and nothing else; `[]` and `{}` are truthy.

"`==` versus `===`?" Loose equality converts by fixed rules (null/undefined equal; strings to numbers; booleans to numbers; objects to primitives); strict compares type and value. Use `===`; `== null` is the one idiom.

"`||` versus `??`?" `||` replaces any falsy left side; `??` only `null` / `undefined`.

"Why is `1 + \"2\"` `\"12\"` but `\"3\" - 1` is `2`?" `+` concatenates if either side is a string; the other operators convert to numbers.

"Is `[] == ![]` true?" Yes — and the derivation is the interviewer's point; the answer is to never write it.

Key takeaways

Eight falsy values; everything else truthy — including `[]`, `{}` and `"0"`.

`&&` / `||` / `??` return operands and short-circuit; `??` and `?.` handle only nullish values.

`===` always; `== null` is the one idiomatic loose equality; know the coercion rules to explain, not to use.

`+` concatenates when a string is involved; `- * / %` convert to numbers; strings compare by code unit.

`Object.is` for `NaN` and `-0`; `Number.isNaN` for `NaN`.

EXERCISE 1

Falsy filter

Read `n` tokens and convert each with the `parse` helper (`empty` is the empty string, `0n` a bigint zero, `[]` an empty array…). Print `<token>: truthy` or `<token>: falsy` using `Boolean(value)`, then `falsy=<count>`. Exactly eight values are falsy — check your list against the output.

Example: `5` then `0 empty [] NaN hello` →

```
0: falsy
empty: falsy
[]: truthy
NaN: falsy
hello: truthy
falsy=3
```

INPUT 1

```
5
0 empty [] NaN hello
```

OUTPUT 1

```
0: falsy
empty: falsy
[]: truthy
NaN: falsy
hello: truthy
falsy=3
```

INPUT 2

```
4
null undefined false 0n
```

OUTPUT 2

```
null: falsy
undefined: falsy
false: falsy
0n: falsy
falsy=4
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
// Turn a token into the value it names: null, undefined, NaN, booleans, numbers, bi
// empty string), [] and {} as JSON, anything else as a plain string.
function parse(t) {
  if (t === "null") return null;
  if (t === "undefined") return undefined;
  if (t === "NaN") return NaN;
  if (t === "true") return true;
  if (t === "false") return false;
  if (t === "empty") return "";
  if (/^-?\d+n$/.test(t)) return BigInt(t.slice(0, -1));
  if (t.startsWith("[") || t.startsWith("{")) { try { return JSON.parse(t); } catch
  if (t.trim() !== "" && Number.isFinite(Number(t))) return Number(t);
  return t;
}
const n = Number(tokens[0]);
let falsy = 0;
for (let i = 1; i <= n; i++) {
  const v = parse(tokens[i]);
  const truthy = Boolean(v);
  if (!truthy) falsy++;
  console.log(`${tokens[i]}: ${truthy ? "truthy" : "falsy"}`);
}
console.log(`falsy=${falsy}`);
```

EXERCISE 2

Loose, strict, or the same?

Read `n` pairs of tokens, convert each with `parse`, and print `<a> <b>: ==<a == b> ===<a === b> is=<Object.is(a, b)>`. Every difference between the three columns is one of the rules from the lesson.

Example: `3` then `null undefined`, `0 empty`, `NaN NaN` →

```
null undefined: ==true ===false is=false
0 empty: ==true ===false is=false
NaN NaN: ==false ===false is=true
```

INPUT 1

```
3
null undefined
0 empty
NaN NaN
```

OUTPUT 1

```
null undefined: ==true ===false is=false
0 empty: ==true ===false is=false
NaN NaN: ==false ===false is=true
```

INPUT 2

```
2
-0 0
[] 0
```

OUTPUT 2

```
-0 0: ==true ===true is=false
[] 0: ==true ===false is=false
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
// Turn a token into the value it names: null, undefined, NaN, booleans, numbers, bi
// empty string), [] and {} as JSON, anything else as a plain string.
function parse(t) {
  if (t === "null") return null;
  if (t === "undefined") return undefined;
  if (t === "NaN") return NaN;
  if (t === "true") return true;
  if (t === "false") return false;
  if (t === "empty") return "";
  if (/^-?\d+n$/.test(t)) return BigInt(t.slice(0, -1));
  if (t.startsWith("[") || t.startsWith("{")) { try { return JSON.parse(t); } catch
  if (t.trim() !== "" && Number.isFinite(Number(t))) return Number(t);
  return t;
}
const n = Number(tokens[0]);
for (let i = 0; i < n; i++) {
  const ta = tokens[1 + 2 * i], tb = tokens[2 + 2 * i];
  const a = parse(ta), b = parse(tb);
  // eslint-disable-next-line eqeqeq
  console.log(`${ta} ${tb}: ==${a == b} ===${a === b} is=${Object.is(a, b)}`);
}
```

QUIZ

1. Which of these is truthy?

A. `""`

B. `0`

C. `[]`

D. `NaN`

2. `count || 10` versus `count ?? 10` when `count` is `0`…

A. Both give `10`

B. `||` gives `10`, `??` gives `0` — nullish coalescing only replaces null/undefined

C. Both give `0`

D. `??` throws

3. `"" == 0`, `"0" == 0`, `"" == "0"` are…

A. true, true, true

B. true, true, false

C. false, true, false

D. false, false, false

4. `1 + 2 + "3"` is…

A. `6`

B. `"33"` — left to right: 3 then `3 + "3"` concatenates

C. `"123"`

D. `"6"`

5. `"10" < "9"` is…

A. `false`

B. `true` — both are strings, so the comparison is by code units

C. A `TypeError`

D. `undefined`

ANSWER KEY · TRUTHINESS, COERCION AND THE TWO EQUALITIES

1. C Every object is truthy, including empty arrays and objects.

2. B Use `??` for defaults whenever zero or an empty string is a valid value.

3. B Strings become numbers against a number; two strings compare as strings.

4. B `"1" + 2 + 3` would be `"123"`.

5. B `10 < "9"` (one number) converts the string and is `false`.

LESSON 5 · 13 MIN

let, const and var — scope, hoisting and the temporal dead zone

Three keywords declare variables, and the oldest one, `var`, is the reason the other two exist. `var` is function-scoped and hoisted with a value of `undefined`, which produced a generation of bugs (the loop-closure classic, accidental globals, variables used before their line); `let` and `const` (2015) are block-scoped, hoisted but *uninitialised*, and `const` cannot be reassigned. This lesson explains scope and hoisting precisely, the **temporal dead zone**, what `const` does and does not freeze, and the conventions modern code follows.

Three declarations

	`var`	`let`	`const`
scope	function (or global)	block	block
hoisted	yes, initialised to `undefined`	yes, uninitialised (TDZ)	yes, uninitialised (TDZ)
redeclare in same scope	allowed	`SyntaxError`	`SyntaxError`
reassign	yes	yes	**no**
creates a global property when top-level in a script	yes (`globalThis.x`)	no	no
use	never in new code	when the binding changes	default

Block scope

```
{
  let a = 1;
  const b = 2;
  var c = 3;
}
console.log(typeof a, typeof b, c);   // "undefined" "undefined" 3
```

`let` and `const` live in the nearest enclosing `{ }` — an `if` body, a loop body, a bare block. `var` ignores blocks and attaches to the enclosing **function** (or the whole script), which is how a `var i` in a `for` loop leaks out and how two `var x` in one function silently share.

Hoisting and the temporal dead zone

All declarations are processed before the code runs — that is **hoisting** — but they are initialised differently:

```
console.log(v);   // undefined — var is hoisted and initialised to undefined
var v = 1;

console.log(l);   // ReferenceError: Cannot access 'l' before initialization
let l = 2;
```

From the start of the block until the `let` / `const` line executes, the binding exists but is in the **temporal dead zone**: any read or write throws. This is a feature — it turns "used before defined" into an immediate, precise error instead of a silent `undefined` that surfaces three functions later. `typeof l` in the TDZ also throws (the one case where `typeof` throws). Function *declarations* are hoisted with their body, so you may call a function above its definition; function *expressions* assigned to `const` are not (module 3).

The loop-closure classic

```
for (var i = 0; i < 3; i++) setTimeout(() => console.log(i), 0);   // 3 3 3
for (let j = 0; j < 3; j++) setTimeout(() => console.log(j), 0);   // 0 1 2
```

With `var`, one `i` is shared by every callback and equals `3` when they run. With `let`, **each iteration gets a fresh binding** — the specification copies the loop variable into a new scope

per iteration — so each closure sees its own `j`. This example alone justifies `let`; module 3 explains closures properly.

`const` is about the binding, not the value

```
const n = 1;
n = 2;                  // TypeError: Assignment to constant variable
const o = { x: 1 };
o.x = 2;                // fine — the object is mutable; the binding still poi
o = {};                 // TypeError
const arr = [];
arr.push(1);            // fine
```

`const` prevents **reassignment**, nothing more. Immutable *contents* need `Object.freeze` (shallow) or discipline (module 4). Style-wise this is exactly right: most bindings are never reassigned, so `const` documents that; a `let` then signals "this one changes — watch it."

Global scope and accidental globals

In a script (not a module), top-level `var` and function declarations become properties of the global object (`globalThis`, `window` in browsers); top-level `let` / `const` do not. Assigning to an undeclared name in sloppy mode creates a global — `total = 0` inside a function with a typo becomes a silent shared variable — which strict mode turns into a `ReferenceError`. Modules (ESM) have their own top-level scope and are strict by default, another reason to prefer them.

Shadowing and redeclaration

An inner block may declare a name that exists outside — **shadowing** — and the inner one wins inside the block. It is legal, occasionally useful (a loop's `const item` inside a function that also has `item`), and usually confusing; linters flag it. Redeclaring with `let` / `const` in the *same* scope is a `SyntaxError`; `var` allows it silently, another way bugs hide.

Naming conventions

`camelCase` for variables and functions; `PascalCase` for classes; `UPPER_SNAKE` for module-level constants that are truly constant values (`MAX_RETRIES`, `API_URL`) — not for every `const`. Booleans read as predicates (`isReady`, `hasItems`); collections plural (`users`); a `let` deserves a name that says what changes (`remaining`, `cursor`).

Practical rules

- `const` by default; `let` when reassignment is needed; `var` never.
- Declare at first use, in the smallest block that needs the name.
- One declaration per line; initialise on the declaration line.
- Turn on ESLint `no-var`, `prefer-const`, `no-shadow`, `no-undef`.
- In a `for` loop, `let i`; in `for…of` / `for…in`, `const item`.

Interview angle

- *"`var` versus `let` versus `const`?"* Function scope and `undefined`-hoisting versus block scope with a TDZ; `const` additionally forbids reassignment.
- *"What is hoisting?"* Declarations are processed before execution; `var` initialises to `undefined`, `let` / `const` remain uninitialised until their line.
- *"What is the temporal dead zone?"* The window between entering a scope and executing a `let` / `const` declaration, during which access throws `ReferenceError`.
- *"Why does `for (var …)` with `setTimeout` print the last value?"* One shared `var` binding; `let` creates a binding per iteration.
- *"Does `const` make an object immutable?"* No — only the binding; use `Object.freeze` for shallow immutability.

Key takeaways

- `const` default, `let` for change, `var` never.
- Block scope for `let` / `const`; function scope for `var`; `var` leaks from blocks and redeclares silently.
- Hoisting: `var` → `undefined`; `let` / `const` → TDZ `ReferenceError` on early access.
- `for (let …)` gives each iteration its own binding — the fix for the closure classic.

`const` freezes the binding, not the object; strict mode/modules stop accidental globals.

EXERCISE 1

The loop-closure classic, both ways

Read `n`. Fill one array with `n` arrow functions created in a `for (var i …)` loop, each returning `i`, and another in a `for (let j …)` loop returning `j`. **After** both loops, call every function and print `var: <results>` and `let: <results>` space-separated. The first line shows one shared binding; the second, a fresh binding per iteration.

Example: `3` →

```
var: 3 3 3
let: 0 1 2
```

INPUT 1

```
3
```

OUTPUT 1

```
var: 3 3 3
let: 0 1 2
```

INPUT 2

```
1
```

OUTPUT 2

```
var: 1
let: 0
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
const withVar = [], withLet = [];
for (var i = 0; i < n; i++) withVar.push(() => i);      // one shared binding
for (let j = 0; j < n; j++) withLet.push(() => j);      // a fresh binding per iterc
console.log(`var: ${withVar.map((f) => f()).join(" ")}`);
console.log(`let: ${withLet.map((f) => f()).join(" ")}`);
```

EXERCISE 2

Hoisting and the dead zone, observed

Read `n` scenario names and run each inside `try` / `catch`, printing `<name>: <result>`: `var-before` (read `typeof v` before the `var v` line → `undefined`), `let-before` (read `l` before its `let` → the error class), `const-reassign` (assign to a `const` → the error class), `const-mutate` (change a

property of a `const` object → `mutated to 2`), `block-leak` (declare `var leaked` and `let kept` inside a block, then report `var=<typeof leaked> let=<typeof kept>` outside it).

Example: `3` then `var-before let-before const-mutate` →

```
var-before: undefined
let-before: ReferenceError
const-mutate: mutated to 2
```

INPUT 1

```
3
var-before let-before const-mutate
```

OUTPUT 1

```
var-before: undefined
let-before: ReferenceError
const-mutate: mutated to 2
```

INPUT 2

```
2
const-reassign block-leak
```

OUTPUT 2

```
const-reassign: TypeError
block-leak: var=number let=undefined
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);

function scenario(name) {
  try {
    switch (name) {
      case "var-before": {
        const seen = typeof v;      // hoisted, initialised to undefined
        var v = 1;
        return seen;
      }
      case "let-before": {
        const seen = l;             // temporal dead zone
        let l = 1;
        return String(seen);
      }
      case "const-reassign": {
        const c = 1;
        // eslint-disable-next-line no-const-assign
        c = 2;
        return "reassigned";
      }
      case "const-mutate": {
        const o = { x: 1 };
        o.x = 2;
        return `mutated to ${o.x}`;
      }
      case "block-leak": {
        { var leaked = 1; let kept = 2; }
        return `var=${typeof leaked} let=${typeof kept}`;
      }
      default:
        return "unknown";
    }
  } catch (e) {
    return e.constructor.name;
  }
}

for (let i = 1; i <= n; i++) console.log(`${tokens[i]}: ${scenario(tokens[i])}`);
```

QUIZ

1. Predict: `for (var i = 0; i < 3; i++) setTimeout(() ⇒ console.log(i))`

A. 0 1 2

B. 3 3 3 — one shared `var` binding read after the loop finished

C. 0 0 0

D. 1 2 3

2. Reading a `let` variable before its declaration line…

A. Gives `undefined`

B. Throws `ReferenceError` — the temporal dead zone

C. Gives `null`

D. Is a syntax error

3. `const o = {}; o.x = 1;` …

A. Throws

B. Is fine — `const` prevents reassigning `o`, not mutating the object

C. Freezes `o`

D. Creates a new object

4. A `var` declared inside an `if` block is visible…

A. Only inside the block

B. Throughout the enclosing function — `var` ignores blocks

C. Nowhere

D. Only after the block

5. Assigning to an undeclared name in a strict-mode function…

A. Creates a global

B. Throws `ReferenceError`

C. Creates a local

D. Is ignored

ANSWER KEY · LET, CONST AND VAR — SCOPE, HOISTING AND THE TEMPORAL DEAD ZONE

1. B `let` creates a binding per iteration and prints 0 1 2.

2. B `var` would give `undefined`; the TDZ turns the mistake into an immediate error.

3. B `Object.freeze` for (shallow) immutability.

4. B Block scope is what `let` and `const` added.

5. B In sloppy mode it silently creates a global — the accidental-global bug.

CHECKPOINT · 24 MIN

Checkpoint — Values, types and coercion

This checkpoint covers the seven primitives and `typeof`, numbers as doubles with `NaN`, safe integers and `BigInt`, strings and their toolkit, truthiness, the coercion rules behind `==`, `+` and the comparison operators, the nullish operators, and `let` / `const` / `var` with hoisting and the temporal dead zone.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- The two `typeof` traps and how to test for `null` and for arrays.
- Why `0.1 + 0.2 !== 0.3`, how to test for `NaN`, and where exact integers end.
- `Number("42px")` versus `parseInt("42px", 10)`; `slice` versus `substring`; why an emoji has length 2.
- The eight falsy values; `||` versus `??`; the rules `==` applies in order.
- What `+` does when one side is a string, and how `<` compares two strings.
- `var` versus `let` in a loop with callbacks, and what the temporal dead zone is.

The programs are a type-inspector that classifies input tokens the way the language would, a safe-arithmetic calculator that switches to `BigInt` when a result would lose precision, and a coercion table generator that prints what `==`, `===` and `+` produce for pairs of values.

EXERCISE 1

Type inspector

Read `n` tokens and classify each with the `parse` helper: print `<token>: <kind>` where the kind is `null` for null, `array` for arrays, and otherwise `typeof`; for numbers append `nan=<Number.isNaN> safe=<Number.isSafeInteger>`.

Example: `4` then `null [1] 3.5 NaN` →

```
null: null
[1]: array
3.5: number nan=false safe=false
NaN: number nan=true safe=false
```

INPUT 1

```
4
null [1] 3.5 NaN
```

OUTPUT 1

```
null: null
[1]: array
3.5: number nan=false safe=false
NaN: number nan=true safe=false
```

INPUT 2

```
4
12 x 9n true
```

OUTPUT 2

```
12: number nan=false safe=true
x: string
9n: bigint
true: boolean
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
// Turn a token into the value it names: null, undefined, NaN, booleans, numbers, bi
// empty string), [] and {} as JSON, anything else as a plain string.
function parse(t) {
  if (t === "null") return null;
  if (t === "undefined") return undefined;
  if (t === "NaN") return NaN;
  if (t === "true") return true;
  if (t === "false") return false;
  if (t === "empty") return "";
  if (/^-?\d+n$/.test(t)) return BigInt(t.slice(0, -1));
  if (t.startsWith("[") || t.startsWith("{")) { try { return JSON.parse(t); } catch
  if (t.trim() !== "" && Number.isFinite(Number(t))) return Number(t);
  return t;
}
const n = Number(tokens[0]);
for (let i = 1; i <= n; i++) {
  const v = parse(tokens[i]);
  const kind = v === null ? "null" : Array.isArray(v) ? "array" : typeof v;
  const extra = typeof v === "number" ? ` nan=${Number.isNaN(v)} safe=${Number.isSaf
  console.log(`${tokens[i]}: ${kind}${extra}`);
}
```

EXERCISE 2

A calculator that never loses digits

Read lines `<a> <op> <b>` (integers as text, `op` one of `+ - *`) until end of input. Compute with `Number` when both operands **and the result** are safe integers; otherwise recompute with `BigInt`. Print `<a> <op> <b> = <result> mode=<number|bigint>`.

Example input

```
2 + 3
9007199254740991 + 1
```

→

```
2 + 3 = 5 mode=number
9007199254740991 + 1 = 9007199254740992 mode=bigint
```

INPUT 1

```
2 + 3
9007199254740991 + 1
```

OUTPUT 1

```
2 + 3 = 5 mode=number
9007199254740991 + 1 = 9007199254740992
mode=bigint
```

INPUT 2

```
123456789012345678 * 2
-5 - 7
```

OUTPUT 2

```
123456789012345678 * 2 =
246913578024691356 mode=bigint
-5 - 7 = -12 mode=number
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim
const apply = (x, op, y) ⇒ (op ≡ "+" ? x + y : op ≡ "-" ? x - y : x * y);
for (const line of lines) {
  const [a, op, b] = line.trim().split(/\s+/);
  const na = Number(a), nb = Number(b);
  const asNumber = apply(na, op, nb);
  if (Number.isSafeInteger(na) && Number.isSafeInteger(nb) && Number.isSafeInteger(a
    console.log(`${a} ${op} ${b} = ${asNumber} mode=number`);
  } else {
    console.log(`${a} ${op} ${b} = ${apply(BigInt(a), op, BigInt(b))} mode=bigint`);
  }
}
```

EXERCISE 3

The coercion table

Read `n` pairs of tokens (via `parse`) and print, for each, `<a> <b> | ==:<a == b> ===:<a === b> +:<repr of a + b> (<typeof a + b>)` where the repr is `JSON.stringify` for strings and arrays and `String` otherwise. Predict every line before running.

Example: `3` then `1 one`, `null 0`, `[] 0` →

```
1 one | ==:false ===:false +:"1one" (string)
null 0 | ==:false ===:false +:0 (number)
[] 0 | ==:true ===:false +:"0" (string)
```

INPUT 1

```
3
1 one
null 0
[] 0
```

OUTPUT 1

```
1 one | ==:false ===:false +:"1one"
(string)
null 0 | ==:false ===:false +:0 (number)
[] 0 | ==:true ===:false +:"0" (string)
```

INPUT 2

```
2
true 1
empty 0
```

OUTPUT 2

```
true 1 | ==:true ===:false +:2 (number)
empty 0 | ==:true ===:false +:"0"
(string)
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
// Turn a token into the value it names: null, undefined, NaN, booleans, numbers, bi
// empty string), [] and {} as JSON, anything else as a plain string.
function parse(t) {
  if (t === "null") return null;
  if (t === "undefined") return undefined;
  if (t === "NaN") return NaN;
  if (t === "true") return true;
  if (t === "false") return false;
  if (t === "empty") return "";
  if (/^-?\d+n$/.test(t)) return BigInt(t.slice(0, -1));
  if (t.startsWith("[") || t.startsWith("{")) { try { return JSON.parse(t); } catch
  if (t.trim() !== "" && Number.isFinite(Number(t))) return Number(t);
  return t;
}
const show = (v) => (typeof v === "string" ? JSON.stringify(v) : Array.isArray(v) ?
const n = Number(tokens[0]);
for (let i = 0; i < n; i++) {
  const ta = tokens[1 + 2 * i], tb = tokens[2 + 2 * i];
  const a = parse(ta), b = parse(tb);
  const sum = a + b;
  // eslint-disable-next-line eqeqeq
  console.log(`${ta} ${tb} | ==:${a == b} ===:${a === b} +:${show(sum)} (${typeof su
}
```

CHECKPOINT QUESTIONS

1. JavaScript's primitive types number…

A. Five

B. Seven — undefined, null, boolean, number, bigint, string, symbol

C. Six

D. Eight

2. `typeof null` and `typeof []` …

A. `"null"` and `"array"`

B. Both `"object"`

C. `"object"` and `"array"`

D. `"undefined"` and `"object"`

3. `9007199254740992 + 1` in JavaScript equals…

A. `9007199254740993`

B. `9007199254740992` — past 2^{53} integers are no longer exact

C. `Infinity`

D. `NaN`

4. `Number("")` and `Number(" ")` are…

A. `NaN` and `NaN`

B. `0` and `0`

C. `0` and `NaN`

D. Throws

5. `parseInt("08", 10)` is…

A. `0`

B. `8`

C. `NaN`

D. `"08"`

6. `"😜".split("").length` is…

A. `1`

B. `2` — split by code unit tears the surrogate pair

C. `0`

D. Throws

7. How many values are falsy?

A. Five

B. Eight

C. Ten

D. Infinitely many

8. `[] == ![]` is…

A. `false`

B. `true` — `![]` is `false` → 0; `[]` → `""` → 0

C. A `TypeError`

D. `undefined`

9. `x ?? y` returns `y` when `x` is…

A. Any falsy value

B. `null` or `undefined` only

C. `0`

D. `""`

10. `"3" * "4"` is…

A. `"34"`

B. `12` — every arithmetic operator except `+` converts to numbers

C. `NaN`

D. `"12"`

11. `let` versus `var` hoisting:

A. Neither is hoisted

B. Both are hoisted; `var` is initialised to `undefined`, `let` stays uninitialised until its line (TDZ)

C. Only `var` is hoisted

D. Only `let` is hoisted

12. Which should be the default declaration in modern code?

A. `var`

B. `let`

C. `const`

D. None — implicit globals

ANSWER KEY · CHECKPOINT — VALUES, TYPES AND COERCION

1. B Plus object, the eighth kind of value.

2. B The two `typeof` traps; use `=== null` and `Array.isArray`.

3. B `BigInt` for exact big integers.

4. B Empty and whitespace-only strings convert to 0 — a common surprise in form handling.

5. B Always pass the radix; historically a leading zero meant octal.

6. B `[..."😜"].length` is 1.

7. B `false, 0, -0, 0n, "", null, undefined, NaN`.

8. B Derivable from the rules, and never to be written.

9. B `||` is the one that replaces every falsy value.

10. B `"3" + "4"` would be `"34"`.

11. B Hence `undefined` versus `ReferenceError` on early access.

12. C `const` documents that a binding never changes; `let` when it must; `var` never.