Module 7 of 16

Errors and error handling

throw/try/catch/finally precisely; the built-in error types and custom hierarchies with codes and cause; where to catch — boundaries, per-item recovery, retries, result objects; validation and assertions; reading stack traces and decoding the common messages.

Error handling is the part of a program that runs when the plan failed, and it is judged by whether anyone can tell what happened. JavaScript gives you very little structure — `throw` takes any value, `catch` takes everything, messages are engine-specific text — so the discipline has to come from you: throw real `Error`s of the right type, give your own errors a name and a stable code, wrap with `cause` when crossing a layer, and catch only where a failure becomes an outcome.

The module starts with the mechanics, including the corners (`finally` overriding a `return`, the stack captured at construction, `JSON.stringify(err)` being `{}`). The type lesson maps each built-in to what triggers it and shows a small, useful custom hierarchy. The strategy lesson answers the real question — where to catch — with boundaries, expected-versus-bug, per-item recovery guarded by type, bounded retries, and result objects for ordinary failures. Validation and assertions follow: fail early at the edge with messages that name the value, collect all problems for a form, assert invariants inside. The last lesson teaches you to read a stack trace top to bottom and decodes the fourteen messages that account for most bugs.

The exercises trace `finally` order, classify real runtime errors by type, build an error hierarchy with `cause` chains, run a batch with per-item recovery that still aborts on a bug, implement retry, validate and normalise records, parse a stack trace, and decode error messages — then a transaction processor, a config loader with wrapped causes, and an RPN calculator that raises the right type for each bad input.

CONTENTS

1	throw, try, catch, finally — the mechanics, precisely	12 min
2	The built-in error types, and designing your own	12 min
3	Where to catch — strategies, boundaries and the result-object alternative	13 min
4	Validation, guard clauses and assertions — failing early with a useful message	12 min

5	Reading stack traces and decoding the errors you will see most	12 min
✓	Checkpoint — Errors	24 min

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

LESSON 1 · 12 MIN

throw, try, catch, finally — the mechanics, precisely

Errors in JavaScript are values that travel: `throw` hands one up the call stack until some `catch` takes it, running every `finally` on the way. The rules are small but the corners are sharp — a `finally` can override a `return`, you can throw anything (and people do), `catch` catches *everything* including bugs, and in async code the stack it unwinds is not the one you think (module 8). This lesson pins the synchronous mechanics down exactly, describes what an `Error` object carries, and sets the vocabulary — throw, propagate, catch, rethrow, wrap — the rest of the module uses.

`throw`

```
throw new Error("amount must be positive");
throw new RangeError(`age ${age} out of range`);   // a built-in subtype (nex
throw "oops";                                        // legal, and a bad idea
```

`throw` takes **any** value. Throwing a non-`Error` — a string, an object literal — loses the stack trace, breaks `instanceof Error` checks and `err.message` reads downstream, and shows up in logs as `Uncaught oops` with no location. Always throw an `Error` (or subclass). A linter rule (`no-throw-literal`) enforces it.

Execution stops at the `throw`: nothing after it in the function runs, and the same is true of every caller up the stack until a `catch` is found. If none is found the exception is **uncaught**: Node prints it and exits with code 1; a browser logs it and continues with the next event.

`try` / `catch`

```
try {
  const data = JSON.parse(text);          // may throw SyntaxError
  use(data);
} catch (err) {
  console.error("bad input:", err.message);
}
```

`catch` receives whatever was thrown. It catches **everything** thrown inside the `try` block — including `TypeError`s from your own bugs — so a `catch` that recovers silently can hide real defects. Check what you caught (`if (err instanceof SyntaxError) … else throw err`) or keep the `try` block small enough that only the expected failure can occur in it. Since 2019 the binding is optional: `catch { … }` when you do not need the error.

Variables declared with `let` / `const` inside `try` are not visible in `catch` or after — declare before the `try` if you need the value later.

`finally`

```
const handle = open(path);
try {
  return process(handle);
} finally {
  handle.close();                          // runs whether process returned, thr
}
```

`finally` runs after `try` / `catch` finish, **no matter how**: normal completion, `return`, `break`, `continue`, or a throw that is propagating. It is the place for cleanup — releasing a lock, closing a handle, restoring state — because it is the only construct guaranteed to run on every path. Two corners:

> A `return` or `throw` **inside** `finally` overrides whatever the `try` / `catch` was completing with — a `return` in `finally` swallows a propagating exception. Do not return from `finally`.
>
> `finally` runs even when `catch` rethrows — the order is `try` → `catch` → `finally` → the exception continues upward.

`try` / `finally` without a `catch` is common and correct: "I do not handle this, but I must clean up."

The `Error` object

```
const err = new Error("disk full", { cause: originalError });   // cause: ES2(
err.name;        // "Error" — subclasses set their own
err.message;     // "disk full"
err.stack;       // "Error: disk full\n    at save (app.js:12:9)\n    at …" —
err.cause;       // the wrapped error
String(err);     // "Error: disk full"   (name: message)
```

`stack` is captured when the object is **created**, not when thrown — creating errors ahead of time gives misleading traces. It is an ordinary string; there is no structured API in the language (Node's `Error.captureStackTrace` and `Error.prepareStackTrace` are V8 extras). `message` and `stack` are own, non-enumerable properties, so `JSON.stringify(err)` gives `{}` — serialise explicitly (`{ name, message, stack }`) for logs and APIs.

Rethrow and wrap

```
try { return JSON.parse(text); }
catch (err) {
  if (err instanceof SyntaxError) throw new Error(`config ${path} is not valid
  throw err;                                  // not ours to handle — propagate u
}
```

Rethrow (`throw err`) passes an error you cannot handle onward, keeping its stack. **Wrap** adds context at a layer boundary — the caller learns *which* config, and the original stays reachable as `cause` . Never `throw new Error(err.message)` — it discards the original type, stack and cause. Whether to rethrow or wrap is the subject of lesson 3.

Errors are values

An exception is only special while it is propagating. Once caught it is an ordinary object: store it, return it, put it in an array of failures, attach it to a result. `const results = items.map((x) ⇒ { try { return { ok: true, value: f(x) }; } catch (err) { return { ok: false, err }; } });` processes a batch without one failure aborting the rest — the **result object** pattern, again in lesson 3.

Common mistakes

Throwing strings or plain objects.

`catch (e) {}` — swallowing everything, including bugs.

Returning from `finally` (silently discards exceptions).

Wrapping an error by copying `message` into a new `Error` — lose type, stack and cause.

A huge `try` block around a whole function so the `catch` cannot know what failed.

Expecting `JSON.stringify(err)` to contain the message.

Interview angle

"What does `finally` guarantee?" It runs on every exit path — normal, `return`, `throw` — after `try` / `catch`; a `return` inside it overrides the outcome.

"Can you throw a non-Error?" Yes; you should not — no stack, no `message`, breaks `instanceof`.

"What is `Error.cause`?" An option (`new Error(msg, { cause })`) that keeps the original error when wrapping with context.

"When is the stack captured?" At construction, not at `throw`.

"Why is `catch (e) {}` dangerous?" It hides bugs (TypeErrors) alongside the expected failure.

Key takeaways

`throw` any value, but always an `Error`; propagation unwinds callers until a `catch`.

`catch` takes everything — check the type or keep `try` small; the binding is optional.

`finally` always runs; never `return` from it; `try` / `finally` alone is fine for cleanup.

`Error` has `name`, `message`, `stack` (captured at construction), optional `cause`; serialise explicitly.

Rethrow to propagate unchanged; wrap with `{ cause }` to add context; caught errors are ordinary values.

EXERCISE 1

In what order does it run?

Write `run(mode, log)` with a `try` / `catch` / `finally`. The `try` pushes `"try"` and, for modes `throw` and `rethrow`, throws `new Error("boom")`; otherwise it returns `"from try"`. The `catch` pushes `catch <message>`, rethrows in mode `rethrow`, otherwise returns `"from catch"`. The `finally` pushes `"finally"` and, in mode `override` only, returns `"from finally"`. For each input line (a mode) call `run` inside an outer `try` / `catch` and print `mode=<mode>: <log joined by ,> → <returned value>` or `→ outer caught <message>`.

Example: `ok`, `throw`, `override`, `rethrow` →

```
mode=ok: try,finally → from try
mode=throw: try,catch boom,finally → from catch
mode=override: try,finally → from finally
mode=rethrow: try,catch boom,finally → outer caught boom
```

`finally` ran every time, and its `return` replaced the `try`'s.

INPUT 1

```
ok
throw
override
rethrow
```

OUTPUT 1

```
mode=ok: try,finally → from try
mode=throw: try,catch boom,finally →
from catch
mode=override: try,finally → from
finally
mode=rethrow: try,catch boom,finally →
outer caught boom
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function run(mode, log) {
  try {
    log.push("try");
    if (mode === "throw" || mode === "rethrow") throw new Error("boom");
    return "from try";
  } catch (err) {
    log.push(`catch ${err.message}`);
    if (mode === "rethrow") throw err;
    return "from catch";
  } finally {
    log.push("finally");
    if (mode === "override") return "from finally";   // overrides whatever try/catc
  }
}
for (const mode of lines.map((l) => l.trim())) {
  const log = [];
  let outcome;
  try {
    outcome = run(mode, log);
  } catch (err) {
    outcome = `outer caught ${err.message}`;
  }
  console.log(`mode=${mode}: ${log.join(",")} -> ${outcome}`);
}
```

EXERCISE 2

Safe division with the right error types

Write `divide(a, b)` that throws `TypeError("operands must be finite numbers, got <a> and <b>")` unless both are finite numbers and `RangeError("division by zero")` when `b` is 0. For each input line `a b` (converted with `Number`) print `<a> / <b> = <result>` or `<name>: <message>`. Count failures in the `catch` and every line in the `finally`; end with `processed=<n> failed=<m>`.

Example: `10 2`, `1 0`, `x 3`, `7 -2` →

```
10 / 2 = 5
RangeError: division by zero
TypeError: operands must be finite numbers, got NaN and 3
7 / -2 = -3.5
processed=4 failed=2
```

INPUT 1

```
10 2
1 0
x 3
7 -2
```

OUTPUT 1

```
10 / 2 = 5
RangeError: division by zero
TypeError: operands must be finite
numbers, got NaN and 3
7 / -2 = -3.5
processed=4 failed=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
let processed = 0, failed = 0;
function divide(a, b) {
  if (!Number.isFinite(a) || !Number.isFinite(b)) throw new TypeError(`operands must
  if (b === 0) throw new RangeError("division by zero");
  return a / b;
}
for (const line of lines) {
  const [a, b] = line.trim().split(/\s+/);
  try {
    console.log(`${a} / ${b} = ${divide(Number(a), Number(b))}`);
  } catch (err) {
    failed++;
    console.log(`${err.name}: ${err.message}`);
  } finally {
    processed++;                        // runs on both paths
  }
}
console.log(`processed=${processed} failed=${failed}`);
```

QUIZ

1. A `try` block throws and the `finally` block contains `return 1;`. The caller sees…

A. The exception

B. The value `1` — a `return` in `finally` overrides the propagating exception

C. Both

D. `undefined`

2. `throw "not found"` is discouraged because…

A. It is a syntax error

B. A string has no stack trace, fails `instanceof Error`, and has no `.message` — every consumer breaks

C. Strings cannot be caught

D. It is slower

3. When is `err.stack` captured?

A. At `throw`

B. At `new Error(...)` — construction, not throwing

C. At `catch`

D. Lazily when read

4. `JSON.stringify(new Error("x"))` gives…

A. `{"message":"x"}`

B. `{}` — `message` and `stack` are own but non-enumerable

C. `"Error: x"`

D. An error

5. `catch (e) {}` around a block is dangerous because…

A. It is slow

B. It swallows every error including `TypeError`s from bugs, and the program continues in a state it never expected

C. It prevents `finally`

D. It only catches strings

ANSWER KEY · THROW, TRY, CATCH, FINALLY — THE MECHANICS, PRECISELY

1. B Which is why returning from `finally` is a bug in real code.

2. B Always throw an `Error` or a subclass.

3. B An error created in a helper and thrown elsewhere points at the helper.

4. B Serialise explicitly with `{ name, message, code }`.

5. B Check the type or keep the `try` small enough that only the expected failure can happen inside.

LESSON 2 · 12 MIN

The built-in error types, and designing your own

The language ships seven error constructors, and the runtime throws each for a specific class of mistake — `TypeError` for "wrong kind of value", `RangeError` for "right kind, impossible value", `SyntaxError` for text that does not parse, `ReferenceError` for a name that does not exist. Knowing which is which tells you what a stack trace means before you read the message, and it tells you which to throw yourself when input is bad. Above them you build **your own** hierarchy — an `AppError` base with a machine-readable `code`, subclasses per failure kind, `cause` for wrapping — so callers can handle by type instead of parsing messages. This lesson covers both halves and how errors should cross a JSON boundary.

The built-ins

Type	The runtime throws it when…	You throw it when…
`TypeError`	a value is the wrong type for an operation: `undefined.x`, `null()`, `x is not a function`, assigning to a frozen property, `new` on an arrow	an argument is the wrong type (`expected a string, got number`)
`RangeError`	a value is the right type but out of range: `new Array(-1)`, `(1).toFixed(101)`, `"x".repeat(-1)`, `BigInt(1.5)`, stack overflow (`Maximum call stack size exceeded`)	a numeric argument is out of bounds (`age must be 0–150`)
`SyntaxError`	source or `JSON.parse` input does not parse: `JSON.parse("{")`, `eval("(")`, a bad regex literal	your parser rejects input
`ReferenceError`	an undeclared identifier is read: `undefinedVariable`, or a `let`/`const` in its temporal dead zone	rarely — it signals a bug, not bad input
`URIError`	`decodeURIComponent("%")` and friends get malformed escapes	almost never
`EvalError`	historically from `eval`; unused by modern engines	never
`AggregateError`	`Promise.any` rejects with every rejection in `.errors` (2021)	you collect several failures into one

All inherit from `Error`, so `instanceof Error` is true for each and `err.name` is the type's name. A **stack overflow** is a `RangeError` (V8), not a special crash — you can catch it, though the recovery options are few.

Messages are **not** part of the specification. `Cannot read properties of undefined (reading 'x')` is V8's wording; Firefox says `x is undefined`. Never branch on message text; branch on type (`instanceof`) or on a code you attached yourself.

Which to throw

Pick by *kind of wrongness*: wrong type → `TypeError`; right type, bad value → `RangeError`; unparsable text → `SyntaxError`; anything domain-specific → your own class. Reserve

`ReferenceError` for the engine. A plain `Error` is fine for one-off failures that no caller will distinguish.

Your own error classes

```
class AppError extends Error {
  constructor(message, { code, cause, ...extra } = {}) {
    super(message, { cause });
    this.name = this.constructor.name;      // "ValidationError" etc., withou
    this.code = code;                       // machine-readable: "E_VALIDATIO
    Object.assign(this, extra);             // any structured fields (field, s
  }
}
class ValidationError extends AppError {
  constructor(field, message, options) { super(message, { code: "E_VALIDATION"
}
class NotFoundError extends AppError {
  constructor(what, id) { super(`${what} ${id} not found`, { code: "E_NOT_FOU
}

try { … }
catch (err) {
  if (err instanceof ValidationError) respond(400, { field: err.field, message
  else if (err instanceof NotFoundError) respond(404, { message: err.message }
  else throw err;                           // unknown: not ours to handle he
}
```

Rules that make a hierarchy useful:

One base class per package or application so `err instanceof AppError` separates "expected, ours" from "unexpected, a bug".

A `code` — a stable string constant. Types can be checked with `instanceof` only inside one process with one copy of the class; a code survives serialisation, logging, a different bundle, and a different language on the other end of the wire.

Structured fields (`field`, `status`, `retryable`) instead of data embedded in the message.

Set `name`; V8 prints `${name}: ${message}` at the top of the stack, so a wrong name misleads every log line.

Keep the constructor signature small; a subclass that takes `(field, message)` is nicer to throw than one that takes an options bag.

Two to three levels is plenty. A twelve-class error tree is a smell; a base class plus a `code` per case usually does the job.

`cause` and chains

`new AppError("could not save order", { cause: dbError })` keeps the low-level error attached. Walk the chain when logging:

```
function describe(err) {
  const parts = [];
  for (let e = err; e; e = e.cause) parts.push(`${e.name}: ${e.message}`);
  return parts.join(" ← ");
}
```

`cause` is set only when passed as an option; `err.cause` is `undefined` otherwise. It can be any value, but keep it an `Error`.

Errors across a boundary

`JSON.stringify(err)` yields `{}` (own properties `message` / `stack` are non-enumerable; your added fields *are* enumerable and would appear alone). Serialise deliberately:

```
const toJSON = (err) ⇒ ({ name: err.name, code: err.code, message: err.messag
```

Never send `stack` to an end user (leaks paths and internals); do send it to your logs. On the receiving side rebuild by `code`, not by `name`, and treat the message as display text.

Checking the type

`err instanceof ValidationError` is the normal check. It fails when two copies of the class exist (duplicate package versions, iframes) — the same failure `instanceof` has everywhere. `err.code ⩵ "E_VALIDATION"` and `err.name ⩵ "ValidationError"` survive that; `code` is the robust one because nobody renames it by accident.

`AggregateError`

```
try { await Promise.any([a(), b()]); }
catch (err) { err instanceof AggregateError && err.errors.forEach(log); }

throw new AggregateError(failures, `${failures.length} items failed`);   // yo
```

The one built-in for "several things went wrong at once".

Common mistakes

- Branching on `err.message` text.
- Throwing `Error` with the type encoded in the message (`"VALIDATION: bad email"`) instead of a class or code.
- Forgetting `this.name` , so logs read `Error: …` for every custom class.
- `instanceof` across package copies; or `err.name` checks that break when a subclass is added.
- Sending `stack` to clients; or `JSON.stringify(err)` and wondering where the message went.

Interview angle

- *"TypeError versus RangeError?"* Wrong type of value versus right type but impossible value.
- *"What throws `ReferenceError` ?"* Reading an undeclared name or a `let` / `const` in its TDZ — a bug signal.
- *"How do you design custom errors?"* `extends Error` , set `name` , add a stable `code` and structured fields, one base class, `cause` for wrapping.
- *"Why a `code` when you have classes?"* It survives serialisation, duplicate class copies and other languages.
- *"Is a stack overflow catchable?"* Yes — it is a `RangeError` in V8.

Key takeaways

`TypeError` (wrong type), `RangeError` (bad value / stack overflow), `SyntaxError` (unparsable, incl. JSON), `ReferenceError` (unknown name), `AggregateError` (many at once).

Messages are engine-specific display text; branch on type or `code`, never on text.

Custom errors: `extends Error`, `this.name = this.constructor.name`, a `code`, structured fields, one base class, shallow tree.

Wrap with `{ cause }` and walk the chain when logging.

Serialise errors explicitly; stack to logs only; match by `code` across boundaries.

EXERCISE 1

Classify what the runtime throws

The starter holds twelve named snippets that each fail in a different way — reading a property of `undefined`, `new Array(-1)`, `JSON.parse("{")`, an undeclared name, `decodeURIComponent("%")`, `BigInt(1.5)`, `(1).toFixed(101)`, writing to a frozen object, assigning to a `const`, a negative `repeat` count, throwing a string. For each snippet name on input run it and print `<name>: <error.constructor.name> instanceofError=<bool>` (or `<name>: no error`).

Example: `read-undefined`, `negative-array`, `bad-json`, `unknown-name`, `repeat-negative`, `throw-string` →

```
read-undefined: TypeError instanceofError=true
negative-array: RangeError instanceofError=true
bad-json: SyntaxError instanceofError=true
unknown-name: ReferenceError instanceofError=true
repeat-negative: RangeError instanceofError=true
throw-string: String instanceofError=false
```

Print the constructor name only — messages are engine-specific text.

INPUT 1

```
read-undefined
call-undefined
negative-array
bad-json
unknown-name
bad-uri
bigint-fraction
too-many-digits
frozen-write
const-assign
repeat-negative
throw-string
```

OUTPUT 1

```
read-undefined: TypeError
instanceofError=true
call-undefined: TypeError
instanceofError=true
negative-array: RangeError
instanceofError=true
bad-json: SyntaxError
instanceofError=true
unknown-name: ReferenceError
instanceofError=true
bad-uri: URIError instanceofError=true
bigint-fraction: RangeError
instanceofError=true
too-many-digits: RangeError
instanceofError=true
frozen-write: TypeError
instanceofError=true
const-assign: TypeError
instanceofError=true
repeat-negative: RangeError
instanceofError=true
throw-string: String
instanceofError=false
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const snippets = {
  "read-undefined": () => undefined.x,
  "call-undefined": () => { const f = undefined; f(); },
  "negative-array": () => new Array(-1),
  "bad-json": () => JSON.parse("{"),
  "unknown-name": () => notDeclaredAnywhere,
  "bad-uri": () => decodeURIComponent("%"),
  "bigint-fraction": () => BigInt(1.5),
  "too-many-digits": () => (1).toFixed(101),
  "frozen-write": () => { const o = Object.freeze({ a: 1 }); o.a = 2; },
  "const-assign": () => new Function('"use strict"; const a = 1; a = 2;')(),
  "repeat-negative": () => "x".repeat(-1),
  "throw-string": () => { throw "just text"; },
};
for (const name of lines.map((l) => l.trim())) {
  try {
    snippets[name]();
    console.log(`${name}: no error`);
  } catch (err) {
    console.log(`${name}: ${err === null || err === undefined ? String(err) : err.co
  }
}
```

EXERCISE 2

A small error hierarchy with cause

Given `AppError(message, { code, cause, ...extra })` (sets `name` from the constructor, stores `code` and any extra fields), write `ValidationError(field, message)` (code `E_VALIDATION`, stores `field`) and `NotFoundError(what, id)` (message `<what> <id> not found`, code `E_NOT_FOUND`, `status` 404), plus `describe(err)` that walks `cause` producing `Name: message ← Name: message`. Commands: `validate <field> <value>` throws a ValidationError `<field> is required` when the value is `-`, else prints `valid <field>=<value>`; `find <what> <id>` throws NotFoundError when the id ends in `0`, else prints `found <what> <id>`; `wrap <message...>` throws a `RangeError("inner value out of range")`, catches it and rethrows `AppError(message, { code: "E_WRAP", cause })`. Caught errors print `<name> [<code>] <message> field=<field or –> status=<status or –> isAppError=<bool> chain=<describe>`.

Example: `validate email -`, `find order 120`, `wrap could not save` →

```
ValidationError [E_VALIDATION] email is required field=email status=- isAppError=tru
NotFoundError [E_NOT_FOUND] order 120 not found field=- status=404 isAppError=true c
AppError [E_WRAP] could not save field=- status=- isAppError=true chain=AppError: co
```

INPUT 1

```
validate email -
validate email ada@x
find order 120
find order 121
wrap could not save
```

OUTPUT 1

```
ValidationError [E_VALIDATION] email is
required field=email status=-
isAppError=true chain=ValidationError:
email is required
valid email=ada@x
NotFoundError [E_NOT_FOUND] order 120
not found field=- status=404
isAppError=true chain=NotFoundError:
order 120 not found
found order 121
AppError [E_WRAP] could not save field=-
status=- isAppError=true chain=AppError:
could not save ← RangeError: inner
value out of range
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class AppError extends Error {
  constructor(message, { code, cause, ...extra } = {}) {
    super(message, { cause });
    this.name = this.constructor.name;
    this.code = code;
    Object.assign(this, extra);
  }
}
class ValidationError extends AppError {
  constructor(field, message) { super(message, { code: "E_VALIDATION", field }); }
}
class NotFoundError extends AppError {
  constructor(what, id) { super(`${what} ${id} not found`, { code: "E_NOT_FOUND", st
}
function describe(err) {
  const parts = [];
  for (let e = err; e; e = e.cause) parts.push(`${e.name}: ${e.message}`);
  return parts.join(" ← ");
}
for (const line of lines) {
  const [cmd, a, ...rest] = line.trim().split(/\s+/);
  const b = rest[0];
  try {
    if (cmd === "validate") {
      if (b === "-") throw new ValidationError(a, `${a} is required`);
      console.log(`valid ${a}=${b}`);
    } else if (cmd === "find") {
      if (b.endsWith("0")) throw new NotFoundError(a, b);
      console.log(`found ${a} ${b}`);
    } else if (cmd === "wrap") {
      try { throw new RangeError("inner value out of range"); }
      catch (inner) { throw new AppError([a, ...rest].join(" "), { code: "E_WRAP", c
    }
  } catch (err) {
    console.log(`${err.name} [${err.code}] ${err.message} field=${err.field ?? "-"}
  }
}
```

QUIZ

1. `(1).toFixed(101)` throws a…

A. `TypeError`

B. `RangeError` — the right type of value, but out of the allowed range

C. `SyntaxError`

D. `ReferenceError`

2. Why give custom errors a `code` string when they already have a class?

A. Classes cannot be caught

B. A code survives serialisation, logging, duplicate copies of the class and other languages; `instanceof` does not

C. Codes are faster

D. It is required by `Error`

3. Branching on `err.message` text is fragile because…

A. Messages are frozen

B. Messages are engine-specific and unspecified — V8 and Firefox word the same error differently

C. Messages are always empty

D. `message` is not a string

4. `new Error("save failed", { cause: dbErr })` — what does `cause` give you?

A. A retry

B. The original error stays attached for logging and diagnosis while the message gains context

C. A different `name`

D. Nothing — it is ignored

5. A `ReferenceError` in production most likely means…

A. Bad user input

B. A bug — an undeclared identifier or a `let` / `const` used before its line

C. A network failure

D. A JSON problem

ANSWER KEY · THE BUILT-IN ERROR TYPES, AND DESIGNING YOUR OWN

1. B `new Array(-1)`, `"x".repeat(-1)` and stack overflow are also `RangeError`s.

2. B Match by `code` across a boundary; `instanceof` inside one process.

3. B Branch on the type or on a code you attached.

4. B Never `throw new Error(err.message)` — that drops type, stack and cause.

5. B You almost never throw it yourself.

LESSON 3 · 13 MIN

Where to catch — strategies, boundaries and the result-object alternative

Knowing the syntax of `try` / `catch` is the easy part; knowing *where* to put one is the skill. Catch too early and you swallow information the caller needed; catch too late and one bad record aborts a batch of ten thousand; never catch and the process dies on a typo in a config file. The working rule: errors are handled at **boundaries** — the places where a failure has a meaning to someone (a request, a job, a file, a UI action) — and everything between the throw and the boundary lets them pass, adding context at most. This lesson gives that rule its details: expected failures versus bugs, rethrow versus wrap, per-item recovery, retries, result objects, and the two process-level nets.

Two kinds of failure

Expected failures are part of the domain: invalid input, a missing record, a network timeout, a file that is not there. They will happen in production every day; code must handle them and users must see something sensible. **Bugs** are defects: `undefined.x`, a wrong argument type, an impossible state. Handling them means *finding* them — logging with a stack trace and, usually, failing the current operation loudly so the defect is noticed. The most common error-handling mistake is treating both the same: a `catch` that shows "something went wrong" for a `TypeError` hides a bug behind a friendly message forever.

The type system from the previous lesson is how you tell them apart: `err instanceof AppError` (or a `code`) is expected; anything else is a bug.

Boundaries

A boundary is where an error stops being a control-flow event and becomes an *outcome*: an HTTP handler turning it into a status code, a CLI's main function printing a message and exiting non-zero, a UI event handler showing a toast, a job runner marking the job failed, a batch loop recording one item's failure. Catch there. Between the throw site and the boundary, functions should **not** catch unless they can do one of three things:

> **Recover** — try an alternative that satisfies the caller (a cache miss falls through to the database; a missing optional file yields defaults).

Add context — wrap with `{ cause }` and information only this layer has (which file, which record id, which step), then rethrow.

Clean up — with `finally`, which is not catching at all.

A `catch` that logs and rethrows at every layer produces the same error in the log five times; log once, at the boundary.

Rethrow versus wrap

Rethrow (`throw err`) when the caller would handle the original type just as well — most of the time. Wrap when crossing an abstraction: a `ConfigError("config.json invalid", { cause: syntaxError })` tells the operator what to fix; the raw `SyntaxError: Unexpected token }` does not. Wrapping also converts a third-party library's error types into yours, so callers depend on your API rather than on the library's. Keep the `cause`.

Per-item recovery

```
const outcomes = records.map((r) => {
  try { return { ok: true, value: importRecord(r) }; }
  catch (err) {
    if (!(err instanceof ValidationError)) throw err;     // a bug aborts the
    return { ok: false, id: r.id, error: err.message };   // an expected failu
  }
});
```

A batch has two natural boundaries: the item (expected failures are recorded, processing continues) and the batch (a bug stops everything). Note the `instanceof` guard — the pattern is *not* "catch everything per item".

Retries

Retry only failures that are **transient** (timeouts, 503s, connection resets) and only operations that are **idempotent** (safe to repeat). A validation error will fail identically every time; retrying a non-idempotent payment charges twice. Bound the attempts, back off between them, and wrap the final failure with the attempt count and the last cause:

```
function withRetry(fn, { attempts = 3, isTransient }) {
  let last;
  for (let i = 1; i ⩽ attempts; i++) {
    try { return fn(i); }
    catch (err) {
      if (!isTransient(err)) throw err;
      last = err;
    }
  }
  throw new Error(`gave up after ${attempts} attempts`, { cause: last });
}
```

(Asynchronous retries with real delays are the same shape with `await` — module 8.)

Result objects instead of exceptions

Some failures are so ordinary that throwing is noise: a parser that rejects half its inputs, a lookup that often finds nothing, a validator that reports many problems at once. Return a value that says so:

```
function parsePort(text) {
  const n = Number(text);
  return Number.isInteger(n) && n > 0 && n < 65536 ? { ok: true, value: n } :
}
const r = parsePort(input);
if (!r.ok) return usage(r.error);
```

Discriminated results (`{ ok: true, value } | { ok: false, error }`) make failure visible in the signature, compose with `map` / `filter`, and collect naturally (a list of all validation errors rather than the first). Exceptions remain right for failures that are exceptional, cross many layers, or would otherwise need `if (!r.ok) return r` on every line. Neither is "correct"; mixing them by layer is normal — parsers and validators return results, I/O and invariants throw.

Do not swallow

`catch (e) {}` and `catch (e) { console.log(e) }` (then continuing as if it worked) are the two forms of the worst bug: the program proceeds in a state it never expected, and the

failure surfaces later somewhere unrelated. If you truly cannot handle an error, let it propagate. If a failure genuinely does not matter (best-effort telemetry, optional cache warm-up), say so in a comment *and* keep the `try` block to the single call that may fail.

The last nets

`process.on("uncaughtException", handler)` and `process.on("unhandledRejection", handler)` catch what nothing else did. They are for **logging and exiting cleanly** — flush the log, close connections, `process.exit(1)` — not for continuing: after an uncaught exception the process may be in an inconsistent state, and Node's documentation says to restart. A supervisor (systemd, a container orchestrator, PM2) restarts the process; that is the recovery. In a browser, `window.onerror` / `unhandledrejection` are the equivalents, used to report to telemetry.

Messages that help

A good message says **what** failed, **which** (the id, path or field), and if possible **what would fix it**: `config db.port must be 1–65535, got 70000` rather than `invalid config`. Include values, never secrets. Messages are for developers and operators; the boundary decides what a user sees.

Common mistakes

- One `try` / `catch` around everything, showing a generic message for bugs and expected failures alike.
- Catching at every layer to log — five copies of one error.
- Catching per item without a type guard, so a bug in the loop body silently fails every record.
- Retrying non-transient or non-idempotent operations, or without a bound.
- Continuing after `uncaughtException`.
- Messages without the identifying value.

Interview angle

"Where should errors be caught?" At boundaries — where a failure becomes an outcome; intermediate layers recover, add context, or clean up, otherwise let it pass.

"Expected failure versus bug?" Domain failures are handled and shown; bugs are logged with a stack and fail loudly; `instanceof AppError` / `code` separates them.

"When would you return a result object instead of throwing?" Ordinary, frequent, or multiple failures — parsers and validators — where the signature should show it.

"When is a retry appropriate?" Transient failure, idempotent operation, bounded attempts with backoff.

"What should `uncaughtException` do?" Log, clean up, exit; let a supervisor restart.

Key takeaways

Separate expected failures (handle, show) from bugs (log with stack, fail loudly) by type or code.

Catch at boundaries; in between, recover, wrap-with-cause, or `finally` — otherwise propagate.

Per-item recovery needs a type guard; a bug should still abort the batch.

Retry only transient + idempotent, bounded, with backoff; wrap the final failure.

Result objects for ordinary failures; exceptions for exceptional ones; never swallow; `uncaughtException` means exit.

EXERCISE 1

A batch with two boundaries

Records arrive one per line: `ok <id>`, `invalid <id> <reason>` (the starter's `importRecord` throws a `ValidationError`) or `crash <id>` (it throws a `TypeError` — a bug). Process every record inside an item-level `try` / `catch` that recovers **only** from `ValidationError` (print `skipped <id>: <message>`), rethrowing anything else; successes print `imported <id>`. Wrap the loop in a batch-level `try` / `catch`: on normal completion print `done: imported=<n> skipped=<m>`; when a bug escapes print `fatal: <name>: <message> - batch aborted after <records seen> records (imported=<n> skipped=<m>)`.

Example: `ok r1`, `invalid r2 bad`, `crash r3`, `ok r4` →

```
imported r1
skipped r2: r2: bad
fatal: TypeError: unexpected null record at r3 - batch aborted after 3 records (impo
```

`r4` was never processed — a bug stops the batch, an expected failure does not.

INPUT 1

```
ok r1
invalid r2 missing name
ok r3
invalid r4 age out of range
ok r5
```

OUTPUT 1

```
imported r1
skipped r2: r2: missing name
imported r3
skipped r4: r4: age out of range
imported r5
done: imported=3 skipped=2
```

INPUT 2

```
ok r1
invalid r2 bad
crash r3
ok r4
```

OUTPUT 2

```
imported r1
skipped r2: r2: bad
fatal: TypeError: unexpected null record
at r3 - batch aborted after 3 records
(imported=1 skipped=1)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class AppError extends Error {
  constructor(message, { code, cause, ...extra } = {}) {
    super(message, { cause });
    this.name = this.constructor.name;
    this.code = code;
    Object.assign(this, extra);
  }
}
class ValidationError extends AppError { constructor(id, reason) { super(`${id}: ${r
function importRecord(kind, id, reason) {
  if (kind === "invalid") throw new ValidationError(id, reason);
  if (kind === "crash") throw new TypeError(`unexpected null record at ${id}`);   //
  return id;
}
let imported = 0, skipped = 0, seen = 0;
try {
  for (const line of lines) {
    const [kind, id, ...rest] = line.trim().split(/\s+/);
    seen++;
    try {
      importRecord(kind, id, rest.join(" "));
      imported++;
      console.log(`imported ${id}`);
    } catch (err) {
      if (!(err instanceof ValidationError)) throw err;   // not an expected failure
      skipped++;
      console.log(`skipped ${err.id}: ${err.message}`);
    }
  }
  console.log(`done: imported=${imported} skipped=${skipped}`);
} catch (err) {
  console.log(`fatal: ${err.name}: ${err.message} - batch aborted after ${seen} reco
}
```

EXERCISE 2

Bounded retry with a wrapped final failure

Implement `withRetry(fn, { attempts, isTransient })`: call `fn(i)` for `i = 1…attempts`; on success print `attempt <i>: ok` and return the value; on a transient error print `attempt <i>: failed (transient)` and try again; on a non-transient error print `attempt <i>: failed (permanent) - not retrying` and rethrow; after the last attempt throw `Error("gave up after <attempts> attempts", { cause: lastError })`. Input `attempts failures kind`: the flaky function fails for the first `failures` calls with a `TransientError("attempt i timed out")` (or `PermanentError("attempt`

`i rejected")` when kind is `permanent`), then returns `ok on attempt i`. Print `result=<value>` or `failed: <cause chain joined by " ← ">`.

Example: `3 2 transient` →

```
attempt 1: failed (transient)
attempt 2: failed (transient)
attempt 3: ok
result=ok on attempt 3
```

`3 5 transient` → three failures then `failed: Error: gave up after 3 attempts ← TransientError: attempt 3 timed out`.

INPUT 1

```
3 2 transient
```

OUTPUT 1

```
attempt 1: failed (transient)
attempt 2: failed (transient)
attempt 3: ok
result=ok on attempt 3
```

INPUT 2

```
3 5 transient
```

OUTPUT 2

```
attempt 1: failed (transient)
attempt 2: failed (transient)
attempt 3: failed (transient)
failed: Error: gave up after 3 attempts
← TransientError: attempt 3 timed out
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class TransientError extends Error { constructor(m) { super(m); this.name = "Transie
class PermanentError extends Error { constructor(m) { super(m); this.name = "Permane
function withRetry(fn, { attempts, isTransient }) {
  let last;
  for (let i = 1; i <= attempts; i++) {
    try {
      const value = fn(i);
      console.log(`attempt ${i}: ok`);
      return value;
    } catch (err) {
      if (!isTransient(err)) { console.log(`attempt ${i}: failed (permanent) - not r
      console.log(`attempt ${i}: failed (transient)`);
      last = err;
    }
  }
  throw new Error(`gave up after ${attempts} attempts`, { cause: last });
}
const [attempts, failures, kind] = lines[0].trim().split(/\s+/);
const flaky = (i) => {
  if (i <= Number(failures)) {
    if (kind === "permanent") throw new PermanentError(`attempt ${i} rejected`);
    throw new TransientError(`attempt ${i} timed out`);
  }
  return `ok on attempt ${i}`;
};
try {
  console.log(`result=${withRetry(flaky, { attempts: Number(attempts), isTransient:
} catch (err) {
  const chain = [];
  for (let e = err; e; e = e.cause) chain.push(`${e.name}: ${e.message}`);
  console.log(`failed: ${chain.join(" ← ")}`);
}
```

QUIZ

1. Errors should be caught…

A. In every function, to be safe

B. At boundaries — where a failure becomes an outcome (a request, a job, a UI action, a batch item); layers in between recover, add context, or clean up

C. Only at the top level

D. Never — let the process crash

2. Per-item recovery in a batch should…

A. Catch everything and continue

B. Catch only the expected failure types and rethrow the rest — a bug should still abort the batch loudly

C. Skip the `catch`

D. Retry each item forever

3. A retry is appropriate when the failure is…

A. Any failure

B. Transient (timeout, 503) **and** the operation is idempotent, with bounded attempts and backoff

C. A validation error

D. A `TypeError`

4. A result object `{ ok, value } | { ok: false, error }` is preferable to throwing when…

A. Never

B. The failure is ordinary and frequent (a parser, a validator reporting many problems) and belongs in the signature

C. The failure is a bug

D. Performance is critical

5. `process.on("uncaughtException", …)` should…

A. Resume normal operation

B. Log, clean up and exit — the process may be inconsistent; a supervisor restarts it

C. Retry the request

D. Be avoided entirely

ANSWER KEY · WHERE TO CATCH — STRATEGIES, BOUNDARIES AND THE RESULT-OBJECT ALTERNATIVE

1. B Catching to log at every layer produces the same error five times.

2. B Without the type guard a bug in the loop body silently fails every record.

3. B Retrying a non-idempotent payment charges twice; a validation error fails identically every time.

4. B Exceptions remain right for exceptional failures that cross many layers.

5. B Node's documentation says exactly this.

LESSON 4 · 12 MIN

Validation, guard clauses and assertions — failing early with a useful message

Most exceptions worth having are thrown on purpose, at the edge, the moment a bad value arrives — because the alternative is a `TypeError` twelve calls later in code that had nothing to do with the mistake. This lesson is the craft of that early failure: validating inputs where they enter, guard clauses that keep the happy path flat, `TypeError` / `RangeError` with messages that name the value, collecting *all* problems for a form, invariants asserted with `node:assert`, and the specific traps around `JSON.parse`, numbers and property lookups that make JavaScript validation different from other languages.

Validate at the edge

Data enters a program at a few places: request bodies, CLI arguments, files, environment variables, third-party responses, user events. Validate **there**, once, and convert to well-typed values; code behind the edge then trusts its inputs. Validation sprinkled through internal functions is repetitive, inconsistent and still misses the edges. The internal functions get *assertions* instead (below) — cheap checks of things that must already be true.

Guard clauses

```
function withdraw(account, amount) {
  if (typeof amount !== "number" || Number.isNaN(amount)) throw new TypeError(
  if (amount <= 0) throw new RangeError(`amount must be positive, got ${amount
  if (amount > account.balance) throw new InsufficientFundsError(account.id, a
  account.balance -= amount;                // the happy path, unindented, la
}
```

Check and throw at the top, in order from "wrong type" to "wrong value" to "domain rule"; the real work follows without nesting. Each guard names **the parameter, the rule, and the actual value** — `got ${amount}` is the difference between a message that fixes the bug and one that starts a search. Interpolate `typeof x` for type failures, `JSON.stringify(x)` for small values, never entire request bodies or secrets.

JavaScript-specific checks

Numbers. `typeof x === "number"` admits `NaN` and `Infinity`. Use `Number.isFinite(x)` for "a real number", `Number.isInteger(x)` for integers, `Number.isSafeInteger` for ids. `Number("")` is `0` and `Number(" 12 ")` is `12` — parse then check, do not trust `Number()` alone; `parseInt("12px")` is `12`, which may or may not be what you want.

Strings. Check `typeof`, then `trim()`, then length and pattern. An empty string is falsy — `if (!name)` rejects it, which is usually right, but `if (!count)` rejects a legitimate `0`. Compare against `undefined` / `""` explicitly when zero is valid.

Objects and arrays. `typeof [] === "object"` and `typeof null === "object"`: test `Array.isArray(x)` and `x !== null && typeof x === "object"`. Test fields with `Object.hasOwn(x, "k")` or `"k" in x`; `x.k !== undefined` cannot tell "absent" from "present as undefined".

Optional values. `x ?? default` treats `null` and `undefined` as missing and keeps `0` / `""` / `false`; `x || default` does not. Choose deliberately.

`JSON.parse`. Always inside `try` / `catch` for external text; it throws `SyntaxError`. Parsing succeeds for `"null"`, `"42"` and `"[]"` — check the *shape* after parsing, not just that parsing worked.

Prototype keys. Data keyed by user strings: reject or skip `__proto__`, `constructor`, `prototype`, or use a `Map`.

Collect everything, or fail on the first?

A function called from code fails on the first problem — one throw, one message. A **form** or **config file** should report *all* problems at once, or the user fixes one and discovers the next on every round trip. Collect into an array and throw one error (or return a result) at the end:

```
function validateUser(u) {
  const errors = [];
  if (typeof u.name !== "string" || u.name.trim() === "") errors.push("name: r
  if (!Number.isInteger(u.age) || u.age < 0 || u.age > 150) errors.push(`age:
  if (typeof u.email !== "string" || !u.email.includes("@")) errors.push("emai
  if (errors.length) throw new ValidationError(errors);
  return { name: u.name.trim(), age: u.age, email: u.email.toLowerCase() };
}
```

Validation is also **normalisation**: return the cleaned value (trimmed, lower-cased, defaults applied) so the rest of the program handles one canonical shape. Schema libraries (zod, Joi, Ajv) do exactly this at scale; the hand-rolled version teaches what they generate.

Assertions: invariants, not input

```
const assert = require("node:assert/strict");
function take(queue) {
  assert(queue.length > 0, "take() on an empty queue");        // AssertionErr
  assert.equal(typeof queue[0].id, "string");
  …
}
```

An **assertion** states something the code believes is already true — a precondition guaranteed by the caller, a postcondition of your own algorithm, a loop invariant. It is not validation of external input: if it fails, the bug is in the program, not in the data, and the right outcome is a loud `AssertionError` (`err.code === "ERR_ASSERTION"`, `err.actual` / `err.expected` populated) that stops the operation. `node:assert/strict` uses `===` semantics for `equal` / `deepEqual`; the legacy `assert.equal` uses `==` — prefer strict. Assertions document assumptions for the reader as much as they check them; they should never have side effects, because some codebases strip them in production.

Fail fast

The principle behind all of this: detect a bad state as **early** and as **close to its cause** as possible, and stop rather than continue with corrupted data. A crash with a good message at the edge costs minutes; a wrong number written to a database because a `NaN` slipped through costs a weekend. Fail fast is not "crash the server" — the boundary decides the outcome (400 to the client, a logged failure for the job); it is "do not proceed past a check you know has failed".

Common mistakes

- `if (!value)` when `0`, `""` or `false` are valid inputs.
- Trusting `typeof x === "number"` (NaN passes) or `Number(text)` (empty string gives 0).

`JSON.parse` without `try` / `catch`, or without checking the parsed shape.

Validating deep inside internal functions instead of at the edge; or asserting external input instead of validating it.

Messages without the actual value; messages that leak secrets.

Returning the first error from a form validator.

Interview angle

"Where do you validate input?" At the edges, once, converting to trusted, normalised values; internals assert invariants.

"How do you check for a real number in JavaScript?" `Number.isFinite(x)` (or `Number.isInteger`), not `typeof` alone.

"Assertion versus validation?" Validation guards external data and produces a user-facing outcome; an assertion checks an internal invariant and its failure means a bug.

"Why collect all validation errors?" One round trip for the user; the caller sees the whole picture.

"What is fail fast?" Detect bad state as early and as near its cause as possible and stop, rather than continue and corrupt.

Key takeaways

Validate at the edge, normalise, and trust inside; guard clauses first, happy path last, flat.

Messages name the parameter, the rule and the actual value (`got …`); never secrets.

`Number.isFinite` / `isInteger`, `Array.isArray`, `x !== null && typeof x === "object"`, `Object.hasOwn`, `??` versus `||`, `JSON.parse` in `try`, guard prototype keys.

Forms and configs collect every error; functions fail on the first.

`node:assert/strict` for invariants — a failed assertion is a bug, not bad input; fail fast.

EXERCISE 1

Validate, collect and normalise

Write `validateUser(u)`: throw a `TypeError("user must be an object, got <null|array|typeof>")` when `u` is not a plain object; otherwise collect **every** problem — `name: required` (non-empty string after trim), `age: must be an integer 0-150, got <JSON>`, `email:`

`must contain @` — and throw one `Error` with the problems joined by `; `; when valid return the normalised record (trimmed name, lower-cased email). Input lines are JSON users, or `assert <a> <b>`, which runs `assert.equal(Number(a), Number(b))` from `node:assert/strict` and prints `assert <a> <b>: passed` or `AssertionError code=<code> actual=<> expected=<>`. Print `ok: <JSON>`, `invalid: <message>` or `type error: <message>`.

Example: `{"name":" Ada ","age":36,"email":"ADA@X.io"}`, `{"name":"","age":-1,"email":"nope"}`, `assert 2 3`, `[1,2]` →

```
ok: {"name":"Ada","age":36,"email":"ada@x.io"}
invalid: name: required; age: must be an integer 0-150, got -1; email: must contain
AssertionError code=ERR_ASSERTION actual=2 expected=3
type error: user must be an object, got array
```

INPUT 1

```
{"name":" Ada
","age":36,"email":"ADA@X.io"}
{"name":"","age":-1,"email":"nope"}
{"name":"Bo","age":30.5,"email":"b@c"}
assert 1 1
assert 2 3
[1,2]
```

OUTPUT 1

```
ok:
{"name":"Ada","age":36,"email":"ada@x.io"}
invalid: name: required; age: must be an
integer 0-150, got -1; email: must contain
@
invalid: age: must be an integer 0-150,
got 30.5
assert 1 1: passed
AssertionError code=ERR_ASSERTION actual=2
expected=3
type error: user must be an object, got
array
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const assert = require("node:assert/strict");
function validateUser(u) {
  const errors = [];
  if (u === null || typeof u !== "object" || Array.isArray(u)) throw new TypeError(`
  if (typeof u.name !== "string" || u.name.trim() === "") errors.push("name: require
  if (!Number.isInteger(u.age) || u.age < 0 || u.age > 150) errors.push(`age: must b
  if (typeof u.email !== "string" || !u.email.includes("@")) errors.push("email: mus
  if (errors.length) throw new Error(errors.join("; "));
  return { name: u.name.trim(), age: u.age, email: u.email.toLowerCase() };
}
for (const line of lines) {
  const text = line.trim();
  if (text.startsWith("assert ")) {
    const [, a, b] = text.split(/\s+/);
    try { assert.equal(Number(a), Number(b)); console.log(`assert ${a} ${b}: passed`
    catch (err) { console.log(`${err.name} code=${err.code} actual=${err.actual} exp
    continue;
  }
  try {
    console.log(`ok: ${JSON.stringify(validateUser(JSON.parse(text)))}`);
  } catch (err) {
    console.log(`${err instanceof TypeError ? "type error" : "invalid"}: ${err.messa
  }
}
```

EXERCISE 2

Result objects for parsing and lookups

Write `parseJson(text)` returning `{ ok: true, value }` or `{ ok: false, error: "parse failed (<error name>)" }` — never throwing — and `getPath(obj, "a.b.c")` that walks the path with `Object.hasOwn`, returning `{ ok: true, value }` or `{ ok: false, error: "missing '<key>' at <path so far or <root>>" }` (a non-object along the way counts as missing). Line 1 is the JSON text; every following line is a path. If parsing failed print its error and nothing else; otherwise print `<path> = <JSON of value>` or `<path>: <error>` per path.

Example: `{"db":{"host":"x","port":5432},"tags":["a"]}` then `db.port`, `db.user`, `tags`, `db.host.length` →

```
db.port = 5432
db.user: missing 'user' at db
tags = ["a"]
db.host.length: missing 'length' at db.host
```

`length` is a property of the string, but not an **own** property of a plain-data object — the lookup is deliberately strict.

INPUT 1

```
{"db":{"host":"x","port":5432},"tags":
["a"]}
db.port
db.user
tags
db.host.length
```

OUTPUT 1

```
db.port = 5432
db.user: missing 'user' at db
tags = ["a"]
db.host.length: missing 'length' at
db.host
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function parseJson(text) {
  try { return { ok: true, value: JSON.parse(text) }; }
  catch (err) { return { ok: false, error: `parse failed (${err.name})` }; }
}
function getPath(obj, path) {
  let current = obj;
  const walked = [];
  for (const key of path.split(".")) {
    if (current === null || typeof current !== "object" || !Object.hasOwn(current, k
      return { ok: false, error: `missing '${key}' at ${walked.length ? walked.join(
    }
    walked.push(key);
    current = current[key];
  }
  return { ok: true, value: current };
}
const parsed = parseJson(lines[0]);
if (!parsed.ok) {
  console.log(parsed.error);
} else {
  for (const path of lines.slice(1).map((l) => l.trim())) {
    const r = getPath(parsed.value, path);
    console.log(r.ok ? `${path} = ${JSON.stringify(r.value)}` : `${path}: ${r.error}
  }
}
```

QUIZ

1. The check for "a real number" in JavaScript is…

A. `typeof x === "number"`

B. `Number.isFinite(x)` — `typeof` lets `NaN` and `Infinity` through

C. `x === +x`

D. `!isNaN(x)`

2. A form validator should…

A. Throw on the first problem

B. Collect every problem and report them together, so the user fixes all of them in one round

C. Return `false`

D. Log and continue

3. Assertion versus validation:

A. Synonyms

B. Validation guards external input and yields a user-facing outcome; an assertion checks an internal invariant and its failure means a bug

C. Assertions are for input

D. Validation is only in tests

4. `if (!count) throw …` is wrong when…

A. Never

B. `0` is a legitimate value — `0`, `""` and `false` are falsy; compare against `undefined` explicitly

C. `count` is a string

D. In strict mode

5. `JSON.parse(text)` succeeded. What else must be checked?

A. Nothing

B. The **shape** — `"null"`, `"42"` and `"[]"` all parse successfully

C. The text length

D. The encoding

ANSWER KEY · VALIDATION, GUARD CLAUSES AND ASSERTIONS — FAILING EARLY WITH A USEFUL MESSAGE

1. B `Number.isInteger` for integers; `Number("")` is 0, so parse then check.

2. B A function called from code fails on the first; a form or config reports all.

3. B `node:assert/strict` throws an `AssertionError` with `code: "ERR_ASSERTION"`.

4. B `x ?? default` keeps `0` / `""` / `false`; `x || default` does not.

5. B Parsing in `try` / `catch` handles syntax; validation handles meaning.

LESSON 5 · 12 MIN

Reading stack traces and decoding the errors you will see most

A stack trace is the single most useful diagnostic you will ever get for free, and most developers read only its first line. This lesson teaches the whole thing — the frame format, top-down order, which frames are yours, why async traces are short, what source maps do — and then decodes the dozen error messages that account for the majority of JavaScript bugs, each with its one-sentence cause and the fix. It closes with the debugging tools that beat `console.log` and the habit that beats all of them: reproduce, then read.

Anatomy of a stack trace

```
TypeError: Cannot read properties of undefined (reading 'name')
    at greet (/app/src/greet.js:12:22)
    at Array.map (<anonymous>)
    at renderList (/app/src/list.js:40:15)
    at main (/app/src/index.js:8:3)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
```

Line one: `name: message`. Then one **frame** per active function, **innermost first** — the top frame is where the error was created, the bottom is the entry point. Each frame is `at <function> (<file>:<line>:<column>)`; frames with no function name read `at <file>:line:col`; native frames say `<anonymous>` or `node:internal/…`. Reading order: start at the top, skip down to the first frame in **your** code (here `greet.js:12:22`), open that line and column, and ask what was `undefined`. The column matters — `a.b.c` has two property reads and the column tells you which one failed.

Node prints frames from `node:internal` for its own machinery; browsers show framework frames. Neither is where the bug is, but the frames *between* yours show how you got there — the `Array.map` frame above says `greet` was a callback, so the bad element came from `renderList`'s array.

Traces you cannot trust

The stack is captured at `new Error()`, not at `throw`. An error created in a helper and thrown elsewhere points at the helper.

Async gaps. After an `await` or inside a `.then` callback, the frames that led to the original call are gone — the callback runs from the event loop. V8 stitches some `await` frames back together ("async stack traces", default since Node 12), which helps for `async` / `await` and not at all for callbacks (module 8).

Minified or bundled code gives `main.js:1:48213`. A **source map** (`//# sourceMappingURL=`) maps that back to the original file, line and function; browsers apply it automatically in devtools, Node needs `--enable-source-maps`. Production error trackers upload source maps for this reason.

`Error.stackTraceLimit` defaults to 10 frames; deep recursion traces are cut. Raise it while debugging.

A wrapped error shows the wrapper's trace; the original is in `err.cause` — log the chain.

The errors you will see most

Message (V8)	What it means	Look for
`Cannot read properties of undefined (reading 'x')`	something before `.x` is `undefined`	the expression to the *left* of `.x`; an unawaited promise, a missing key, a misspelled property, `find` returning `undefined`
`x is not a function`	you called something that is not a function	the wrong import (`default` vs named), a property that is data, a method on the wrong object, calling the result instead of the function
`x is not defined`	an identifier does not exist in scope	a typo, a missing `import` / `require`, a variable declared in another block
`Cannot access 'x' before initialization`	`let` / `const` / `class` used in its temporal dead zone	code order; circular imports
`Assignment to constant variable.`	reassigning a `const`	you meant `let`, or you meant to mutate a property
`Maximum call stack size exceeded`	unbounded recursion (a `RangeError`)	a missing base case; a getter/setter calling itself; `toJSON` returning `this`
`Unexpected token } in JSON at position 42`	`JSON.parse` on text that is not JSON	an HTML error page, a trailing comma, single quotes, an empty response
`Unexpected end of JSON input`	`JSON.parse("")` or truncated text	an empty body
`Cannot set properties of undefined (setting 'x')`	assigning through `undefined`	an uninitialised object, `arr[i]` past the end
`Converting circular structure to JSON`	`JSON.stringify` on a cycle	parent/child links; use a replacer or `toJSON`
`Class constructor X cannot be invoked`	called a class as a function	add `new`; or you passed the class where an instance was expected

Message (V8)	What it means	Look for
`without 'new'`		
`Invalid array length`	`new Array(-1)` or `length = 2**32`	a negative or huge computed size
`Cannot use import statement outside a module`	ESM syntax in a CommonJS file	`"type": "module"`, `.mjs`, or `require` (module 9)
`EADDRINUSE`, `ENOENT`, `ECONNREFUSED`	Node I/O errors with a `code`	a port in use, a missing path, nothing listening

Two habits: read the *identifier* in the message (it is the name from your source), and read the **column**.

`console.error`, `util.inspect` and friends

`console.log(err)` prints the stack and any own enumerable properties (your `code`, `field`); `console.log(err.message)` loses everything else. Use `console.error` for errors — it writes to stderr, which lets you separate diagnostics from program output. `util.inspect(obj, { depth: null, colors: true })` prints nested objects fully where `console.log` stops at depth 2 and prints `[Object]`. `console.table(rows)` for arrays of records; `console.dir(obj, { depth: null })` as a shortcut.

Beyond `console.log`

`debugger;` statement + `node inspect app.js` or `node --inspect-brk app.js` and Chrome's `chrome://inspect` — breakpoints, step, watch, call stack, scope variables. Faster than log-and-rerun once the bug takes more than two prints to find.

Conditional breakpoints (right-click a breakpoint in devtools) — stop only when `id === 42`.

`Error.captureStackTrace(obj)` — V8: attach a `stack` to any object, hiding frames above a given function; used by libraries to make custom errors point at the caller.

`node --trace-uncaught`, `--trace-warnings`, `--enable-source-maps`.

`process.on("warning")` and `NODE_OPTIONS=--trace-deprecation` to find deprecated calls.

The habit

Reproduce first — a failing input, a test, a one-line script. Then read the top frame in your code and the column. Then form a hypothesis and check it with one print or breakpoint. Random edits without a reproduction are how a one-hour bug becomes a one-day bug.

Common mistakes

Reading only the first line of a trace; ignoring the column.

Searching the message text on the web before opening the line it points at.

Blaming the framework frame at the top when your frame is three lines down.

`console.log(err.message)` in a catch — the stack and fields are gone.

Debugging minified code without source maps.

Interview angle

"How do you read a stack trace?" Top line is type and message; frames are innermost first; find the first frame in your code, open the line and column.

"Why are async stack traces short?" The callback runs from the event loop; the original call frames have returned — V8 reconstructs some for `await`.

"`Cannot read properties of undefined (reading 'x')` — what is wrong?" The expression left of `.x` is `undefined`, not `x`.

"What is a source map?" A file mapping minified positions back to original source; devtools apply it, Node needs `--enable-source-maps`.

"`x is not a function` — likely causes?" Wrong import shape, data instead of method, wrong receiver, calling a result.

Key takeaways

Frames are innermost first; find your first frame, read line **and column**; native frames explain the route, not the bug.

Stacks are captured at construction; async and minified traces need `await`-stitching and source maps; log the `cause` chain.

Know the fourteen messages: most say "something to the left is undefined", "this is not a function", or "this name does not exist".

`console.error(err)` (not `.message`), `util.inspect` with `depth: null`, `debugger` + `--inspect-brk`.

Reproduce, read the frame, hypothesise, verify — in that order.

EXERCISE 1

Parse a stack trace

The input is a V8-style stack trace: a header `<Name>: <message>` and frames `at fn (file:line:col)` or `at file:line:col` (function-less; treat the function as `<anonymous>`) or `at fn (<anonymous>)` (native). Print `error: <Name> - <message>`; `frames=<count> top=<fn>@<file>:<line>:<col>`; `yours=<fn> <file>:<line> col <col>` for the first frame whose file is neither `<anonymous>` nor starts with `node:` (or `yours=none`); and `route=<function names of those non-native frames joined by " ← ">`.

Example:

```
TypeError: Cannot read properties of undefined (reading 'name')
    at greet (/app/src/greet.js:12:22)
    at Array.map (<anonymous>)
    at renderList (/app/src/list.js:40:15)
    at main (/app/src/index.js:8:3)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
```

→

```
error: TypeError - Cannot read properties of undefined (reading 'name')
frames=5 top=greet@/app/src/greet.js:12:22
yours=greet /app/src/greet.js:12 col 22
route=greet ← renderList ← main
```

INPUT 1

```
TypeError: Cannot read properties of
undefined (reading 'name')
    at greet (/app/src/greet.js:12:22)
    at Array.map (<anonymous>)
    at renderList
(/app/src/list.js:40:15)
    at main (/app/src/index.js:8:3)
    at Module._compile
(node:internal/modules/cjs/loader:1105:14)
```

OUTPUT 1

```
error: TypeError - Cannot read
properties of undefined (reading
'name')
frames=5
top=greet@/app/src/greet.js:12:22
yours=greet /app/src/greet.js:12 col
22
route=greet ← renderList ← main
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const [header, ...frameLines] = input.split("\n").filter((l) ⇒ l.trim() ≠ "");
const frames = frameLines.map((l) ⇒ {
  const m = /^\s*at (?:(.+?) \((.+):(\d+):(\d+)\)|(.+):(\d+):(\d+)|(.+?) \((<anonymo
  if (!m) return null;
  if (m[1] ≠ undefined) return { fn: m[1], file: m[2], line: Number(m[3]), col: Nu
  if (m[5] ≠ undefined) return { fn: "<anonymous>", file: m[5], line: Number(m[6])
  return { fn: m[8], file: m[9], line: 0, col: 0 };
}).filter(Boolean);
const [, name, message] = /^(\w+): (.*)$/.exec(header.trim()) ?? [null, "Error", hea
console.log(`error: ${name} - ${message}`);
console.log(`frames=${frames.length} top=${frames[0].fn}@${frames[0].file}:${frames[
const own = frames.find((f) ⇒ !f.file.startsWith("node:") && f.file ≠ "<anonymous
console.log(own ? `yours=${own.fn} ${own.file}:${own.line} col ${own.col}` : "yours=
console.log(`route=${frames.filter((f) ⇒ f.file ≠ "<anonymous>" && !f.file.starts
```

EXERCISE 2

Decode the message

Build a table of `[regex, category, hint(match)]` rules and classify each input line (an error message) into: `undefined-read` (`Cannot read properties of undefined/null (reading 'k')` → `the expression before .k is undefined`), `undefined-write` (`Cannot set properties of null` → `assigning through null - the object was never created`), `not-a-function` (`x is not a function` → `x holds data, a wrong import, or the wrong receiver`), `not-defined` (`x is not defined` → `no x in scope: typo or missing import`), `tdz` (`Cannot access 'x' before initialization` → `x is used before its let/const/class line runs`), `const-assign`, `stack-overflow`, `json` (any `Unexpected token` / `Unexpected end of JSON` / `in JSON at position`), `circular`, `class-without-new` (→ `call new X(...)`), else `unknown: read the top frame of the stack`. Print `<category>: <hint>` per line.

Example: `Cannot read properties of undefined (reading 'name')`, `user.save is not a function`, `foo is not defined`, `Class constructor Point cannot be invoked without 'new'` →

```
undefined-read: the expression before .name is undefined
not-a-function: user.save holds data, a wrong import, or the wrong receiver
not-defined: no foo in scope: typo or missing import
class-without-new: call new Point(...)
```

INPUT 1

```
Cannot read properties of undefined (reading 'name')
user.save is not a function
foo is not defined
Cannot access 'x' before initialization
Assignment to constant variable.
Maximum call stack size exceeded
Unexpected token } in JSON at position 42
Converting circular structure to JSON
Class constructor Point cannot be invoked without 'new'
something entirely different
```

OUTPUT 1

```
undefined-read: the expression before .name is undefined
not-a-function: user.save holds data, a wrong import, or the wrong receiver
not-defined: no foo in scope: typo or missing import
tdz: x is used before its let/const/class line runs
const-assign: you reassigned a const - use let, or mutate a property
stack-overflow: unbounded recursion: check the base case
json: JSON.parse got text that is not JSON - log the raw text
circular: JSON.stringify met a cycle - break it or use a replacer
class-without-new: call new Point(...)
unknown: read the top frame of the stack
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const rules = [
  [/Cannot read propert(?:y|ies) of (undefined|null) \(reading '(.+?)'\)/, "undefine
  [/Cannot set propert(?:y|ies) of (undefined|null)/, "undefined-write", (m) => `ass
  [/^(.+?) is not a function$/, "not-a-function", (m) => `${m[1]} holds data, a wror
  [/^(.+?) is not defined$/, "not-defined", (m) => `no ${m[1]} in scope: typo or mis
  [/Cannot access '(.+?)' before initialization/, "tdz", (m) => `${m[1]} is used bef
  [/Assignment to constant variable/, "const-assign", () => "you reassigned a const
  [/Maximum call stack size exceeded/, "stack-overflow", () => "unbounded recursion:
  [/Unexpected (?:token|end of JSON)|in JSON at position/, "json", () => "JSON.parse
  [/Converting circular structure to JSON/, "circular", () => "JSON.stringify met a
  [/Class constructor (.+?) cannot be invoked without 'new'/, "class-without-new", (
];
for (const line of lines) {
  const text = line.trim();
  const hit = rules.map(([re, category, hint]) => { const m = re.exec(text); return
  console.log(hit ? `${hit.category}: ${hit.hint}` : `unknown: read the top frame of
}
```

QUIZ

1. In a stack trace, the frames are listed…

A. Outermost first

B. Innermost first — the top frame is where the error was created

C. Alphabetically

D. Randomly

2. `Cannot read properties of undefined (reading 'x')` means…

A. `x` is undefined

B. The expression to the **left** of `.x` evaluated to `undefined`

C. `x` is not declared

D. `x` is not a function

3. Stack traces through a `setTimeout` callback are short because…

A. Callbacks have no stack

B. The callback runs later from the event loop; the frames that scheduled it have already returned

C. V8 truncates them

D. They are hidden for security

4. A source map…

A. Compresses code

B. Maps positions in minified or bundled output back to the original files and lines; devtools apply it, Node needs `--enable-source-maps`

C. Lists dependencies

D. Is only for CSS

5. `console.log(err.message)` in a `catch` is worse than `console.error(err)` because…

A. It is slower

B. It discards the stack, the name and any `code` /fields, and writes to stdout instead of stderr

C. It throws

D. Messages are private

ANSWER KEY · READING STACK TRACES AND DECODING THE ERRORS YOU WILL SEE MOST

1. B Find the first frame in your own code and read its line **and column**.

2. B Look at what produced the object — a `find` that returned `undefined`, an unawaited promise, a misspelled key.

3. B V8 reconstructs `await` chains (async stack traces), not callback chains.

4. B Without it a production trace says `main.js:1:48213`.

5. B Log the error object; walk `cause` for wrapped errors.

CHECKPOINT · 24 MIN

Checkpoint — Errors

This checkpoint covers the mechanics of `throw` / `try` / `catch` / `finally` (including what `finally` overrides and when the stack is captured), the built-in error types and what triggers each, designing custom errors with `name`, `code`, fields and `cause`, the boundary strategy for where to catch, per-item recovery with type guards, retries, result objects, validation at the edge with guard clauses, assertions for invariants, and reading a stack trace.

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:

- What a `return` inside `finally` does to a propagating exception; why `catch (e) {}` is dangerous.
- `TypeError` versus `RangeError` versus `SyntaxError` versus `ReferenceError`; what a stack overflow is.
- Why custom errors carry a `code` as well as a class; what `cause` is for; why `JSON.stringify(err)` is `{}`.
- Where to catch (boundaries) and what the layers in between may do; why per-item recovery needs a type guard.
- `Number.isFinite` versus `typeof`; validation versus assertion; how to read the top frame and column of a trace.

The programs are a transaction processor with a custom error hierarchy and all-or-nothing transfers, a configuration loader that wraps failures with `cause` and reports the chain, and an RPN calculator that raises the right error type for each kind of bad input.

EXERCISE 1

A transaction processor with all-or-nothing transfers

Using the starter's `AppError`, define `AccountNotFound(id)` (code `E_NOT_FOUND`, message `account <id> not found`), `InvalidAmount(text)` (code `E_INVALID_AMOUNT`, `invalid amount <text>` — for anything not a finite positive number) and `InsufficientFunds(id, need, have)` (code `E_INSUFFICIENT`, `<id> has <have>, needs <need>`). Commands: `open <id> <amount>`, `deposit <id> <amount>`, `withdraw <id> <amount>`, `transfer <from> <to> <amount>` — which must

validate **both** accounts and the balance before changing anything, so a failed transfer leaves every balance untouched — and `show` (prints `balances: id=balance ...` in creation order). Print `<code>: <message>` for each `AppError`; rethrow anything else. End with `errors: <code=count sorted by code>` (or `errors: none`).

Example (excerpt): `open ada 100`, `open bob 20`, `transfer ada bob 60`, `transfer ada cy 10`, `transfer bob ada 500`, `show` →

```
E_NOT_FOUND: account cy not found
E_INSUFFICIENT: bob has 80, needs 500
balances: ada=40 bob=80
errors: E_INSUFFICIENT=1 E_NOT_FOUND=1
```

INPUT 1

```
open ada 100
open bob 20
deposit ada 50
withdraw bob 30
transfer ada bob 60
transfer ada cy 10
transfer bob ada 500
deposit ada -5
withdraw zed 1
show
```

OUTPUT 1

```
E_INSUFFICIENT: bob has 20, needs 30
E_NOT_FOUND: account cy not found
E_INSUFFICIENT: bob has 80, needs 500
E_INVALID_AMOUNT: invalid amount -5
E_NOT_FOUND: account zed not found
balances: ada=90 bob=80
errors: E_INSUFFICIENT=2
E_INVALID_AMOUNT=1 E_NOT_FOUND=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class AppError extends Error {
  constructor(message, { code, cause, ...extra } = {}) {
    super(message, { cause });
    this.name = this.constructor.name;
    this.code = code;
    Object.assign(this, extra);
  }
}
class AccountNotFound extends AppError { constructor(id) { super(`account ${id} not
class InvalidAmount extends AppError { constructor(amount) { super(`invalid amount $
class InsufficientFunds extends AppError { constructor(id, need, have) { super(`${id
const accounts = new Map();
const errorCounts = new Map();
function account(id) {
  if (!accounts.has(id)) throw new AccountNotFound(id);
  return accounts.get(id);
}
function amountOf(text) {
  const n = Number(text);
  if (!Number.isFinite(n) || n <= 0) throw new InvalidAmount(text);
  return n;
}
function withdraw(id, amount) {
  const acc = account(id);
  if (acc.balance < amount) throw new InsufficientFunds(id, amount, acc.balance);
  acc.balance -= amount;
}
function transfer(from, to, amount) {
  const src = account(from), dst = account(to);       // validate everything before
  if (src.balance < amount) throw new InsufficientFunds(from, amount, src.balance);
  src.balance -= amount;
  dst.balance += amount;
}
for (const line of lines) {
  const [cmd, a, b, c] = line.trim().split(/\s+/);
  try {
    if (cmd === "open") accounts.set(a, { balance: amountOf(b) });
    else if (cmd === "deposit") account(a).balance += amountOf(b);
    else if (cmd === "withdraw") withdraw(a, amountOf(b));
    else if (cmd === "transfer") transfer(a, b, amountOf(c));
    else if (cmd === "show") console.log(`balances: ${[...accounts].map(([id, acc])
  } catch (err) {
    if (!(err instanceof AppError)) throw err;
    errorCounts.set(err.code, (errorCounts.get(err.code) ?? 0) + 1);
    console.log(`${err.code}: ${err.message}`);
  }
```

```
}
console.log(`errors: ${[...errorCounts].sort(([x], [y]) => x.localeCompare(y)).map((
```

EXERCISE 2

A config loader that keeps the cause

Each line is a JSON config. `loadConfig(text)` parses it (a `SyntaxError` is wrapped as `ConfigError("config is not valid JSON", { cause })`), requires a plain object (`ConfigError("config must be an object")` with a `TypeError("got <null|array|typeof>")` cause), then validates `host` (non-empty string, trimmed; `TypeError("host must be a non-empty string, got <JSON>")`), `port` (integer 1–65535; `RangeError("port must be 1-65535, got <JSON>")`) and `retries` (default 3; non-negative integer; `RangeError("retries must be a non-negative integer, got <JSON>")`) — each field failure wrapped as `ConfigError("invalid config at <field>", { cause })`. Print `ok: host=<> port=<> retries=<>` or `error chain: <Name: message ← Name: message>`, showing a `SyntaxError` by name only (its message differs between engines).

Example: `{"host":"db.local","port":5432}`, `{"host":"x","port":70000}`, `{oops`, `[1,2]` →

```
ok: host=db.local port=5432 retries=3
error chain: ConfigError: invalid config at port ← RangeError: port must be 1-65535
error chain: ConfigError: config is not valid JSON ← SyntaxError
error chain: ConfigError: config must be an object ← TypeError: got array
```

INPUT 1

```
{"host":"db.local","port":5432}
{"host":"","port":5432}
{"host":"x","port":70000}
{"host":"x","port":80,"retries":-1}
{oops
[1,2]
```

OUTPUT 1

```
ok: host=db.local port=5432 retries=3
error chain: ConfigError: invalid config at host ← TypeError: host must be a non-empty string, got ""
error chain: ConfigError: invalid config at port ← RangeError: port must be 1-65535, got 70000
error chain: ConfigError: invalid config at retries ← RangeError: retries must be a non-negative integer, got -1
error chain: ConfigError: config is not valid JSON ← SyntaxError
error chain: ConfigError: config must be an object ← TypeError: got array
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class ConfigError extends Error { constructor(message, options) { super(message, opt
function loadConfig(text) {
  let raw;
  try { raw = JSON.parse(text); }
  catch (err) { throw new ConfigError("config is not valid JSON", { cause: err }); }
  if (raw === null || typeof raw !== "object" || Array.isArray(raw)) throw new Confi
  const check = (field, fn) => {
    try { return fn(raw[field]); }
    catch (err) { throw new ConfigError(`invalid config at ${field}`, { cause: err }
  };
  return {
    host: check("host", (v) => { if (typeof v !== "string" || v.trim() === "") throw
    port: check("port", (v) => { if (!Number.isInteger(v) || v < 1 || v > 65535) thr
    retries: check("retries", (v) => { if (v === undefined) return 3; if (!Number.is
  };
}
function describe(err) {
  const parts = [];
  for (let e = err; e; e = e.cause) parts.push(e instanceof SyntaxError ? e.name :
  return parts.join(" ← ");
}
for (const line of lines) {
  try {
    const cfg = loadConfig(line);
    console.log(`ok: host=${cfg.host} port=${cfg.port} retries=${cfg.retries}`);
  } catch (err) {
    console.log(`error chain: ${describe(err)}`);
  }
}
```

EXERCISE 3

An RPN calculator with typed failures

Evaluate each line as a reverse-Polish expression over `+ - * /` and decimal numbers. Throw `SyntaxError("unknown token '<t>'")` for anything else, `SyntaxError("not enough operands for '<op>'")` when an operator finds fewer than two values, `SyntaxError("too many operands: <n> left on the stack")` when more than one value remains, and `RangeError("division by zero")`. Print `<expression> = <result>` or `<name>: <message>`; count every line in a `finally` and failures in the `catch`; end with `evaluated=<n> failed=<m>`.

Example: `3 4 +`, `5 1 2 + 4 * + 3 -`, `1 0 /`, `1 +`, `1 2 3`, `2 x *` →

```
3 4 + = 7
5 1 2 + 4 * + 3 - = 14
RangeError: division by zero
SyntaxError: not enough operands for '+'
SyntaxError: too many operands: 3 left on the stack
SyntaxError: unknown token 'x'
evaluated=6 failed=4
```

INPUT 1

```
3 4 +
5 1 2 + 4 * + 3 -
1 0 /
1 +
1 2 3
2 x *
```

OUTPUT 1

```
3 4 + = 7
5 1 2 + 4 * + 3 - = 14
RangeError: division by zero
SyntaxError: not enough operands for '+'
SyntaxError: too many operands: 3 left
on the stack
SyntaxError: unknown token 'x'
evaluated=6 failed=4
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const OPS = {
  "+": (a, b) => a + b,
  "-": (a, b) => a - b,
  "*": (a, b) => a * b,
  "/": (a, b) => { if (b === 0) throw new RangeError("division by zero"); return a /
};
function evaluate(expression) {
  const stack = [];
  for (const token of expression.trim().split(/\s+/)) {
    if (Object.hasOwn(OPS, token)) {
      if (stack.length < 2) throw new SyntaxError(`not enough operands for '${token}
      const b = stack.pop(), a = stack.pop();
      stack.push(OPS[token](a, b));
    } else if (/^-?\d+(\.\d+)?$/.test(token)) {
      stack.push(Number(token));
    } else {
      throw new SyntaxError(`unknown token '${token}'`);
    }
  }
  if (stack.length !== 1) throw new SyntaxError(`too many operands: ${stack.length}
  return stack[0];
}
let evaluated = 0, failed = 0;
for (const line of lines) {
  try {
    console.log(`${line.trim()} = ${evaluate(line)}`);
  } catch (err) {
    failed++;
    console.log(`${err.name}: ${err.message}`);
  } finally {
    evaluated++;
  }
}
console.log(`evaluated=${evaluated} failed=${failed}`);
```

CHECKPOINT QUESTIONS

1. Order of execution for `try { A } catch { B } finally { C }` when `A` throws:

A. A, C

B. A (until the throw), B, C

C. A, C, B

D. B, A, C

2. `let` / `const` declared inside `try` are…

A. Visible in `catch` and after

B. Block-scoped to the `try` block — declare before the `try` if you need them later

C. Hoisted to the function

D. Global

3. `JSON.parse("{")` throws a…

A. `TypeError`

B. `SyntaxError`

C. `RangeError`

D. `JSONError`

4. `Promise.any` rejects with…

A. The first rejection

B. An `AggregateError` whose `.errors` holds every rejection

C. `undefined`

D. An array

5. Why set `this.name = this.constructor.name` in an error base class?

A. For `instanceof` to work

B. So logs and `String(err)` show the subclass name instead of `Error`, without repeating it in every subclass

C. It is required

D. To make it enumerable

6. `err instanceof ValidationError` can be `false` for a genuine `ValidationError` when…

A. It was thrown asynchronously

B. Two copies of the class exist (duplicate package versions, iframes) — a `code` check survives this

C. It was caught twice

D. It has a `cause`

7. Wrapping an error by `throw new Error(err.message)` loses…

A. Nothing

B. The original type, stack and any `cause`/fields — use `{ cause: err }` instead

C. Only the stack

D. Only the name

8. A per-item `catch` in a batch loop without a type guard will…

A. Work correctly

B. Silently record a bug (a `TypeError` in the loop body) as a failed item for every record

C. Abort the batch

D. Retry

9. Which value does `x ?? 10` **not** replace?

A. `null`

B. `undefined`

C. `0` — nullish coalescing only replaces `null` and `undefined`

D. Both `null` and `undefined`

10. `node:assert/strict` differs from legacy `assert` in that…

A. It is slower

B. `equal` / `deepEqual` use `===` semantics instead of `==`

C. It cannot throw

D. It only works in tests

11. The column number in a stack frame tells you…

A. Nothing useful

B. Which expression on the line failed — `a.b.c` has two reads, and the column says which

C. The indentation

D. The error code

12. `console.error` versus `console.log` for errors:

A. Identical

B. `console.error` writes to stderr so diagnostics can be separated from program output

C. `console.error` throws

D. `console.log` is for errors

ANSWER KEY · CHECKPOINT — ERRORS

1. B `finally` runs after `catch`, even when `catch` rethrows.

2. B A common source of `ReferenceError` after a `try`.

3. B Any unparsable text — source or JSON — is a `SyntaxError`.

4. B The one built-in for several failures at once; you can construct it yourself for batches.

5. B V8 prints `${name}: ${message}` at the top of the stack.

6. B Same failure mode `instanceof` has everywhere.

7. B Add context; keep the original attached.

8. B Recover from expected failures; let bugs escape to the batch boundary.

9. C `x || 10` would replace `0`, `""` and `false` too.

10. B Prefer strict.

11. B Read line **and** column.

12. B And log the error object, not just `.message`.

CodeKairo · JavaScript study plan · Module 7: Errors and error handling. Solve the exercises online to track your progress and earn XP.