ScopeTreasure Courses

JAVASCRIPT COURSES · LESSON 3.23 · CODE SCOPE SERIES

Error Handling & localStorage

Apps that survive mistakes: try/catch/finally, throw, and localStorage — saving data in the browser so a refresh loses nothing.

Introduction

Two things separate a practice script from a real app. First: real apps **survive mistakes** — bad input, broken data, a failed calculation — instead of dying on the first error. Second: real apps **remember** — close the page, press refresh, and the data is still there.

This lesson delivers both: `try` / `catch` / `finally` and `throw` for surviving, and `localStorage` — the browser's small permanent drawer — for remembering. *(Roman Urdu: Achi app galti par marti nahi, sambhalti hai — aur localStorage ki wajah se refresh ke baad bhi data yaad rehta hai.)*

Key points

- **`try` holds code that can fail** — put the risky lines inside `try { ... }`. If any line inside throws an error, the rest of the `try` block is skipped at once.
- **`catch (error)` receives the error object** — the program lands here instead of dying, and it **survives**. The error object carries `error.message` — the human-readable line that explains what went wrong. Show it, log it, or decide with it.

  (Roman Urdu: catch woh jaal hai jo error ko pakar leta hai — program girta nahi, aage chalta rehta hai.)
- **`finally` runs either way** — success or failure, the `finally { ... }` block always runs last. It is the place for cleanup: hiding a loading spinner, closing what you opened, resetting a button.
- **`throw` raises your own error** — `throw new Error("Name cannot be empty");` stops the current function and throws, exactly like a built-in error. Use it for **validation with teeth**: input that is impossible (dividing by zero, an empty name) gets refused loudly, not silently turned into nonsense.
- **`JSON.parse` is the perfect live demo** — parsing broken JSON text throws a real error every time. In a `try` block, `catch` catches it and the app continues — in the open, the same line kills the script.
- **`localStorage` is a small permanent drawer for this site, in this browser** — `localStorage.setItem(key, value)` saves, `localStorage.getItem(key)` reads back (or `null` if the key is not there), `localStorage.removeItem(key)` deletes one key, `localStorage.clear()` empties the whole drawer.
- **Values are strings only** — `localStorage` cannot hold an object or an array directly. Objects travel as JSON: `JSON.stringify` on the way in, `JSON.parse` on the way out — Lesson 3.18 paying off.

  (Roman Urdu: localStorage mein sirf text jata hai — object ko JSON bana kar rakho, nikaal kar wapas parse karo.)
- **Honest limits** — about **5 MB** of space, saved in **this browser on this device only** (nothing syncs to a server or another phone), and it is easy for anyone with the browser to read — so **never store passwords** or secret data in it.

Code examples

Example 1 — try, catch, finally, and your own throw

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero");
  }
  return a / b;
}

try {
  const result = divide(10, 0);
  console.log("Result: " + result);
} catch (error) {
  console.log("Caught an error: " + error.message);
} finally {
  console.log("Division attempt finished");
}

console.log("Program still running");
```

What it does: `divide` refuses impossible input with its own `throw`. The call sits inside `try`, so the thrown error jumps straight to `catch`, which prints the error's `message`. `finally` runs after that, and — the whole point — the program carries on to its last line instead of dying at the bad division.

What you will see: three lines, in this order: `Caught an error: Cannot divide by zero`, then `Division attempt finished`, then `Program still running`. The `Result:` line never appears, because the error skipped the rest of the `try` block. Change the call to `divide(10, 2)` and run again: now you see `Result: 5`, then `Division attempt finished`, then `Program still running` — `finally` and survival on both paths.

Example 2 — JSON.parse on broken text: the app survives

```
const brokenText = "this is not json";

try {
  const data = JSON.parse(brokenText);
  console.log("Parsed name: " + data.name);
} catch (error) {
  console.log("The data could not be read.");
  console.log("Reason: " + error.message);
}

console.log("The app continues with default settings");
```

What it does: It tries to parse text that is not JSON at all. `JSON.parse` throws — parsing broken text is impossible, and it says so. The `catch` block turns that crash into a calm decision: tell the user the data could not be read, show the reason, and let the app continue with defaults.

What you will see: first line: `The data could not be read.` Second line: `Reason: ` followed by the browser's own JSON error message — the exact wording of this line differs a little between browsers, because each browser writes its parse errors in its own words. Third line: `The app continues with default settings`. Remove the `try` / `catch` and run the parse alone: the script dies on that line and the last message never appears — that difference is the whole lesson.

Example 3 — localStorage: the refresh-restore page

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Remember My Name — Code Scope</title>
</head>
<body>
  <h1 id="greeting">Hello, guest</h1>

  <input type="text" id="nameInput" placeholder="Write your name">
  <button id="saveBtn">Save name</button>
  <button id="removeBtn">Forget name</button>

  <script>
    const greeting = document.getElementById("greeting");
    const nameInput = document.getElementById("nameInput");

    function showSavedName() {
      const savedName = localStorage.getItem("username");
      if (savedName !== null) {
        greeting.textContent = "Hello, " + savedName;
      } else {
        greeting.textContent = "Hello, guest";
      }
    }

    document.getElementById("saveBtn").addEventListener("click", () => {
      const name = nameInput.value.trim();
      if (name === "") {
        alert("Please write a name first");
        return;
      }
      localStorage.setItem("username", name);
      showSavedName();
    });

    document.getElementById("removeBtn").addEventListener("click", () => {
      localStorage.removeItem("username");
      showSavedName();
    });

    showSavedName();
  </script>
```

```
</body>
</html>
```

What it does: On every page load, the script reads the `username` key from `localStorage` and greets that name — or `guest` if nothing is saved. The Save button stores the typed name with `setItem`; the Forget button deletes just that key with `removeItem` and the greeting falls back to guest. No variable holds the name between loads — the drawer does.

What you will see: open the page: `Hello, guest`. Type `Ali` and click Save name: the heading changes to `Hello, Ali`. Now press **F5** to refresh — the heading still says `Hello, Ali`. A normal variable would have died with the refresh; `localStorage` did not. Click Forget name: the heading returns to `Hello, guest`, and a refresh now keeps it as guest, because the key itself is gone.

Quick recap

- **`try`** holds risky code, **`catch (error)`** receives the error (read `error.message`) and the program survives, **`finally`** runs either way for cleanup; **`throw new Error("...")`** raises your own error for impossible input.
- **`localStorage`** saves per site, in this browser, across refreshes: `setItem`, `getItem`, `removeItem`, `clear`.
- Values are **strings only** — objects go in with `JSON.stringify` and come out with `JSON.parse`. About 5 MB, this browser only, **never passwords**.

Common mistakes

1. **Putting the whole program inside one giant `try`** — when everything is "protected", the `catch` cannot tell you *what* failed, and real bugs get quietly swallowed. The fix: wrap only the specific risky lines, and always use `error.message` in the `catch` so failures stay visible.
2. **Saving an object straight into localStorage** — `localStorage.setItem("user", userObject)` stores the useless text `[object Object]`, and your data is gone. The fix: `setItem("user", JSON.stringify(userObject))` in, `JSON.parse(getItem("user"))` out.
3. **Trusting `getItem` to always find the key** — on the first ever visit, the key does not exist and `getItem` returns **`null`**, which crashes code that expects text or an object. The fix: check for `null` first (or use `JSON.parse(localStorage.getItem("tasks")) || []` for a safe default, as the Todo project does).

Practice

1. Write a `setAge(age)` function that throws `new Error("Age cannot be negative")` when the age is below 0, and otherwise returns the age. Call it inside `try`/`catch` with `-5` and then with `25`, and print a message from `finally` both times.
2. Build a tiny page with one colour button: clicking it saves `"dark"` under the key `theme` in `localStorage` and paints the page background dark. On load, read the key — if it says `"dark"`, start dark. Refresh the

page and prove the choice survived. Add a second button that removes the key and returns the page to light.

Practice quiz

1. What do try, catch and finally each do?

- try catches, catch tries, finally throws
- try holds risky code, catch receives the error and the program survives, finally runs either way for cleanup
- All three run only on success
- finally runs only when there is an error

Answer: try holds risky code, catch receives the error and the program survives, finally runs either way for cleanup

2. What kind of values can localStorage hold directly?

- Objects and arrays, as they are
- Strings only, so objects travel as JSON: JSON.stringify on the way in, JSON.parse on the way out
- Numbers only
- Anything, with no limit

Answer: Strings only, so objects travel as JSON: JSON.stringify on the way in, JSON.parse on the way out

3. What does localStorage.getItem return for a key that was never saved, and what are its honest limits?

- An empty string, unlimited space, synced to all devices
- null, about 5 MB, this browser on this device only, and never store passwords in it
- An error, 5 GB, synced everywhere
- undefined, and it is safe for passwords

Answer: null, about 5 MB, this browser on this device only, and never store passwords in it

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/js-error-handling-localstorage · More free tools and notes at https://scopetreasure.com