ChatGPT

Error Handling in JavaScript (Synchronous)

JavaScript **errors** are problems in your code that prevent it from running normally. For example, referencing an undefined variable or performing an illegal operation will stop your script. When an error occurs, JavaScript creates an **Error object** containing details about the problem. These error objects include things like the error's name (type), message, and a stack trace [1] [2]. You can think of error handling like a safety net: you let your code **"try"** to run, and if something goes wrong, you **"catch"** the problem and handle it gracefully, instead of the script crashing.

Common Error Types

JavaScript has several built-in error types for different problems. Some common ones include:

- **SyntaxError** – Thrown when the code has invalid syntax (e.g. a missing parenthesis). These errors often occur before code runs (parse time) [3]. For example, `eval("foo bar");` will produce a SyntaxError.
- **ReferenceError** – Occurs when you reference a variable that doesn't exist or is out of scope [4]. Example:

```
try {
  console.log(x); // x is not defined
} catch (e) {
  console.log(e.name, e.message);
  // ReferenceError: x is not defined
}
```

- **TypeError** – Happens when a value is not of the expected type [5]. For instance, calling a non-function like `null.f()` causes a TypeError:

```
try {
  null.f();
} catch (e) {
  console.log(e.name, e.message);
  // TypeError: Cannot read properties of null (reading 'f')
}
```

- **RangeError** – Thrown when a numeric value is outside its allowed range [6]. For example, `new Array(-1)` triggers a RangeError.
- **URIError** – Triggered by invalid use of global URI functions (like `decodeURIComponent` with bad input) [7].
- (There are others like **EvalError**, **InternalError**, etc., for special cases.)

Each error object's **name** (like `"TypeError"`) tells you what kind of error it is, and the **message** describes what went wrong [8].

The `try...catch...finally` Structure

To handle errors, JavaScript provides `try`, `catch`, and `finally` blocks. You put code that might throw an error inside `try { … }`. If an error occurs, execution jumps to `catch`, where you can inspect the error and respond. A `finally` block (optional) runs **no matter what**, even if there was no error. MDN explains: *"The code in the try block is executed first, and if it throws an exception, the code in the catch block will be executed. The code in the finally block will always be executed before control flow exits the entire construct."* [9].

```
try {
  // Code that might throw an error
  let data = JSON.parse(userInput);
  console.log("Data is", data);
} catch (error) {
  // This runs only if the try block throws an error
  console.error("An error occurred:", error.message);
} finally {
  // This runs regardless of what happened above
  console.log("Cleanup can be done here.");
}
```

A few key points: - The `catch(error)` block receives the thrown error object as `error`. You can use `error.name` and `error.message` to understand it. - The `finally` block always runs, whether or not an error was thrown. It's ideal for cleanup tasks (like closing files) that must happen in either case [10]. - In JavaScript, a `try` block **must** be followed by either a `catch` or `finally` (or both). If you write a `try` without one of these, the code itself is invalid and will throw a SyntaxError [11].

Throwing Custom Errors

You can throw your own errors using the `throw` statement. Typically you throw an `Error` object (or a subclass) with a helpful message:

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

try {
  console.log(divide(10, 0));
} catch (e) {
  console.error(e.name + ":", e.message);
  // Output: Error: Cannot divide by zero
}
```

Technically JavaScript lets you throw anything (numbers, strings, etc.), but **don't** do that in practice. As the guide notes, throwing primitives is legal but *"doesn't provide details about the error, such as its type, name, or stack trace,"* so it's discouraged [12] [13]. Always throw an `Error` (or a custom error class) to get a proper stack trace and name.

You can also create **custom error classes** by extending `Error`. For example:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

throw new ValidationError("Invalid email address");
```

Now `catch` blocks can use `instanceof` to identify specific errors if needed.

Nesting try-catch Blocks

You can nest `try`/`catch` blocks inside each other to handle errors at different levels of your code. For example:

```
try {
  try {
    throw new Error("Inner error");
  } catch (inner) {
    console.log("Caught inside inner catch:", inner.message);
    // Optionally rethrow or handle it
    throw inner; // rethrow to outer catch
  }
} catch (outer) {
  console.log("Caught by outer catch:", outer.message);
}
```

In this example, the inner catch handles the error first, logs it, and then rethrows it so the outer catch can see it. This pattern lets each level add context or cleanup before passing the error along.

Inspecting Error Objects

When you catch an error, you usually get an `Error` object (or subclass instance). Useful properties include:

- `error.name` – the type of error (e.g. `"TypeError"`, `"ReferenceError"`) [14] [8].
- `error.message` – a descriptive message (the string you passed when throwing, or a default one).
- `error.stack` – a stack trace showing where the error occurred (browser support is widespread though technically non-standard) [15] [8].

For example:

```
try {
  undefinedFunction();
} catch (e) {
  console.log("Error name:", e.name);       // e.g. "ReferenceError"
  console.log("Error message:", e.message); // e.g. "undefinedFunction is not
defined"
  console.log("Stack trace:", e.stack);     // shows call stack
}
```

Modern engines include a stack trace by default, which is invaluable for debugging. Remember that error objects may also have extra info like line numbers or custom properties, but `name`, `message`, and `stack` are the standard ones [8] [15].

When and Why to Use Error Handling

Use error handling to make your code **robust** and user-friendly. Without catching errors, any runtime exception will stop the script immediately. By catching exceptions, you can prevent a small problem from crashing your whole application. For instance, if you're parsing user input or calling an external API, wrapping the code in a `try-catch` lets you show a friendly message or attempt a backup solution instead of the page going blank. In other words, catching an exception *"will not let it propagate up through the stack and crash your program"* [12].

Think of `try-catch` like a seat belt: you hope it's never needed, but it can save you from a crash if something goes wrong. Use it around code that might fail (file I/O, network requests, JSON parsing, etc.) so you can recover gracefully or at least log a clear error.

Common Pitfalls and Beginner Mistakes

- **Empty `catch` blocks.** Simply doing `catch (e) {}` and ignoring the error is dangerous. It silently swallows problems and makes bugs hard to find [16]. Always handle the error or at least log it.
- **Forgetting `catch` or `finally`**. In JavaScript, a `try` must be followed by a `catch` or `finally`. Leaving out both causes a syntax error in your code [11].

- **Catching too generally or too late.** Don't wrap huge chunks of code in one `try-catch` if only a small part might fail; that can hide which part caused the error. Also, catching an error without rethrowing can make debugging confusing (you might think it never happened).
- **Throwing non-Error values.** As noted, avoid `throw "oops"` or `throw 123`; always throw an `Error` object so the error has a meaningful name and stack [13].
- **Ignoring the `finally` block.** If you need cleanup (like closing a file or releasing resources), put it in `finally`. Remember it runs whether or not there was an error.
- **Assuming `catch` can handle syntax errors.** SyntaxErrors occur at compile/parse time, before the code runs, so `try-catch` *cannot* catch a missing parenthesis in your source code. Those must be fixed manually.
- **Overusing exceptions for flow control.** Exceptions should be for *unexpected* problems, not normal conditions. Don't use `try-catch` to handle regular control flow (e.g. using it to break loops or return early) – it makes code hard to read and can hurt performance.

By understanding these basics and common mistakes, beginners can use JavaScript's synchronous error handling (`try`, `catch`, `finally`, and `throw`) effectively. Proper error handling leads to more resilient code that fails gracefully and is easier to debug.

Sources: Official documentation and guides [1] [17] [9] [12] [13] [10] [16].

---

[1] [2] [8] [10] [11] [12] [13] [16] A Definitive Guide to Handling Errors in JavaScript
https://kinsta.com/blog/errors-in-javascript/

[3] [4] [5] [6] [7] [14] [15] [17] Error - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

[9] try...catch - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch