JULY 11, 2025 / #JAVASCRIPT

The JavaScript Error Handling Handbook

Tapas Adhikary

Errors and exceptions are inevitable in application development. As programmers, it is our responsibility to handle these errors gracefully so that the user experience of the application is not compromised. Handling errors correctly also helps programmers debug and understand what caused them so they can deal with them.

JavaScript has been a popular programming language for over three decades. We build web, mobile, PWA, and server-side applications using JavaScript and various popular JavaScript-based libraries (like ReactJS) and frameworks (like Next.js, Remix, and so on).

Being a loosely typed language, JavaScript imposes the challenge of handling type safety correctly. TypeScript is useful for managing types, but we still need to handle runtime errors efficiently in our code.

Errors like `TypeError`, `RangeError`, `ReferenceError` are probably pretty familiar to you if you've been building with JavaScript for a while. All these errors may cause invalid data, bad page transitions, unwanted results, or even the entire application to crash – none of which will make end users happy!

In this handbook, you'll learn everything you need to know about error handling in JavaScript. We will start with an understanding of errors, their types, and occurrences. Then you'll learn how to deal with these errors so that they don't cause a bad user experience. At the end, you'll also learn to build your own custom error types and clean-up methodologies to handle your code flow better for optimizations and performance.

I hope you enjoy reading along and practising the tasks I have provided at the end of the article.

This handbook is also available as a video session as part of the 40 Days of JavaScript initiative. Please check it out.

Table of Contents

Errors in JavaScript

Errors and exceptions are the events that disrupt program execution. JavaScript parses and executes code line by line. The source code gets evaluated with the grammar of the programming language to ensure it is valid and executable. If there is a mismatch, JavaScript encounters a `parsing error`. You'll need to make sure you follow the right syntax and grammar of the language to stay away from parsing errors.

Take a look at the code snippet below. Here, we have made the mistake of not closing the parentheses for the console.log.

```
console.log("hi"
```

This will lead to a Syntax Error like this:

Other types of errors can happen due to wrong data input, trying to read a value or property that doesn't exist, or acting on inaccurate data. Let's see some examples:

```
console.log(x); // Uncaught ReferenceError: x is not defined

let obj = null;
console.log(obj.name); // Uncaught TypeError: Cannot read properties of null

let arr = new Array(-1) // Uncaught RangeError: Invalid array length

decodeURIComponent("%"); // Uncaught URIError: URI malformed

eval("var x = ;"); // Uncaught EvalError
```

Here is the list of possible runtime errors you may encounter, along with their descriptions:

- `ReferenceError` – Occurs when trying to access a variable that is not defined.
- `TypeError` – Occurs when an operation is performed on a value of the wrong type.
- `RangeError` – Occurs when a value is outside the allowable range.
- `SyntaxError` – Occurs when there is a mistake in the syntax of the JavaScript code.

JavaScript code.

- `URIError` – Occurs when an incorrect URI function is used in encoding and decoding URIs.
- `EvalError` – Occurs when there is an issue with the eval() function.
- `InternalError` – Occurs when the JavaScript engine runs into an internal limit (stack overflow).
- `AggregateError` – Introduced in ES2021, used for handling multiple errors at once.
- `Custom Errors` – These are user-defined errors, and we will learn how to create and use them soon.

Have you noticed that all the code examples we used above result in a message explaining what the error is about? If you look at those messages closely, you will find a word called `Uncaught`. It means the error occurred, but it was not caught and managed. That's exactly what we will now go for – so you know how to handle these errors.

Handling Errors With the `try` and `catch`

JavaScript applications can break for various reasons, like invalid syntax, invalid data, missing API responses, user mistakes, and so on. Most of these reasons may lead to an application crash, and your users will see a blank white page.

Rather than letting the application crash, you can gracefully handle these situations using `try…catch`.

```
try {
    // logic or code
} catch (err) {
    // handle error
}
```

The `try` Block

The `try` block contains the code – the business logic – which might throw an error. Developers always want their code to be error-free. But at the same time, you should be aware that the code might throw an error for several possible reasons, like:

- Parsing JSON
- Running API logic
- Accessing nested object properties
- DOM manipulations
- … and many more

When the code inside the try block throws an error, the code execution of the remaining code in the try block will be suspended, and the control moves to the nearest catch block. If no error occurs, the catch block is skipped completely.

```
try {
  // Code that might throw an error
} catch (error) {
  // Handle the error here
}
```

The `catch` Block

The catch block runs only if an error was thrown in the try block. It receives the `Error object` as a parameter to give us more information about the error. In the example shown below, we are using something called `abc` without declaring it. JavaScript will throw an error like this:

```
try {
    console.log("execution starts here");
    abc;
    console.log("execution ends here");
} catch (err) {
    console.error("An Error has occured", err);
}
```

JavaScript executes code line by line. The execution sequence of the above code will be:

- First, the string "execution starts here" will be logged to the console.
- Then the control will move to the next line and find the `abc` there. What is it? JavaScript doesn't find any definition of it anywhere. It's time to raise the alarm and throw an error. The control doesn't move to the next line (the next console log), but rather moves to the catch block.
- In the catch block, we handle the error by logging it to the console. We could do many other things like show a toast message, send the user an email, or switch off a toaster (why not if your customer needs it).

Without try...catch, the error would crash the app.

Error Handling: Real-World Use Cases

Let's now see some of the real-world use cases of error handling with try…catch.

Handling Division by Zero

Here is a function that divides a number by another number. So, we have parameters passed to the function for both numbers. We want to make sure that the division never encounters an error for dividing a number by zero (0).

As a proactive measure, we have written a condition that if the divisor is zero, we will throw an error saying that division by zero is not allowed. In every other case, we will proceed with the division operation. In case of an

error, the catch block will handle the error and do what's needed (in this case, logging the error to the console).

```
function divideNumbers(a, b) {
    try {
        if (b === 0) {
            const err = new Error("Division by zero is not allowed.");
            throw err;
        }
        const result = a/b;
        console.log(`The result is ${result}`);
    } catch(error) {
        console.error("Got a Math Error:", error.message)
    }
}
```

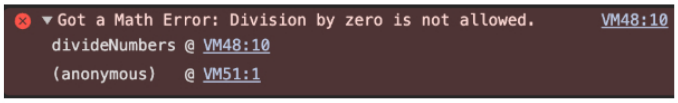
Now, if we invoke the function with the following arguments, we will get a result of 5, and the second argument is a non-zero value.

```
divideNumbers(15, 3); // The result is 5
```

But if we pass the 0 value for the second argument, the program will throw an error, and it will be logged into the console.

```
divideNumbers(15, 0);
```

Output:



Handling JSON

Often, you will get JSON as a response to an API call. You need to parse this JSON in your JavaScript code to extract the values. What if the API sends you some malformed JSON by mistake? You can't afford to let your user interface crash because of this. You need to handle it gracefully – and here comes the try...catch block again to the rescue:

```
function parseJSONSafely(str) {
  try {
    return JSON.parse(str);
  } catch (err) {
    console.error("Invalid JSON:", err.message);
    return null;
  }
}

const userData = parseJSONSafely('{"name": "tapaScript"}'); // Parsed
const badData = parseJSONSafely('name: tapaScript');        // Handled gracefully
```

Without try...catch, the second call will crash the app.

Anatomy of the Error Object

Getting errors in programming can be a scary feeling. But `Error`s in JavaScript aren't just some scary, annoying messages – they are structured objects that carry a lot of helpful information about what went wrong, where, and why.

As developers, we need to understand the anatomy of the `Error` object to help us better with faster debugging and smarter recovery in production-level application issues.

Let's deep dive into the Error object, its properties, and how to use it effectively with examples.

What is the Error Object?

The JavaScript engine throws an `Error` object when something goes wrong during runtime. This object contains helpful information like:

- An error message: This is a human-readable error message.
- The error type: TypeError, ReferenceError, SyntaxError, and so on that we discussed above.
- The stack trace: This helps you navigate to the root of the error. It is a string containing the stack trace at the point the error was thrown.

Let's take a look at the code snippet below. The JavaScript engine will throw an error in this code, as the variable `y` is not defined. The error object contains the error name (type), message, and the stack trace information.

```
function doSomething() {
  const x = y + 1; // y is not defined
}
try {
  doSomething();
} catch (err) {
  console.log(err.name);    // ReferenceError
  console.log(err.message); // y is not defined
  console.log(err.stack); // ReferenceError: y is not defined
                          // at doSomething (<anonymous>:2:13)
                          // at <anonymous>:5:3
}
```

Tip: If you need any specific properties from the error object, you can use destructuring to do that. Here is an example where we are only interested in the error name and message, not the stack.

```
try {
  JSON.parse("{ invalid json }");
} catch ({name, message}) {
  console.log("Name:", name);       // Name: SyntaxError
  console.log("Message:", message); // Message: Expected property name or '}' in JSON at po
}
```

Throwing Errors and Re-throwing Errors

JavaScript provides a `throw` statement to trigger an error manually. It is very helpful when you want to handle an invalid condition in your code (remember the divide by zero problem?).

To throw an error, you need to create an instance of the `Error` object with details and then throw it.

```
throw new Error("Something is bad!");
```

When the code execution encounters a `throw` statement,

- It stops the execution of the current code block immediately.
- The control moves to the nearest catch block (if any).
- If the catch block is not found, the error will not be caught. The error gets bubbled up, and may end up crashing the program. You can learn more in-depth about events and event bubbling from here.

Rethrowing

At times, catching the error itself in the catch block is not enough. Sometimes, you may not know how to handle the error completely, and you might want to do additional things, like:

- Adding more context to the error.
- Logging the error into a file-based logger.
- Passing the error to someone more specialized to handle it.

This is where `rethrow` comes in. With rethrowing, you catch an error, do something else with it, and then throw it again.

```
function processData() {
  try {
    parseUserData();
  } catch (err) {
    console.error("Error in processData:", err.message);
    throw err; // Rethrow so the outer function can handle it too
  }
}

function main() {
  try {
    processData();
  } catch (err) {
    handleErrorBetter(err);
  }
}
```

In the code above, the `processData()` function catches an error, logs it, and then `throw`s it again. The outer `main()` function can now catch it and do something more to handle it better.

In the real-world application development, you would want to separate the concerns for errors, like:

- **API Layer** – In this layer, you can detect HTTP failures

```
async function fetchUser(id) {
  const res = await fetch(`/users/${id}`);
  if (!res.ok) throw new Error("User not found"); // throw it here
  return res.json();
}
```

- **Service Layer** – In this layer, you handle business logic. So the error will be handled for invalid conditions.

```
async function getUser(id) {
  try {
    const user = await fetchUser(id);
    return user;
  } catch (err) {
    console.error("Fetching user failed:", err.message);
    throw new Error("Unable to load user profile"); // rethrowing
  }
}
```

- **UI Layer** – Show a user-friendly error message.

```
async function showUserProfile() {
  try {
    const user = await getUser(123);
    renderUser(user);
  } catch (err) {
    displayError(err.message); // A proper message show to the user
  }
}
```

The `finally` with try-catch

The try...catch block gives us a way to handle errors gracefully. But you may always want to execute some code irrespective of whether an error occurred or not. For example, closing the database connection, stopping a loader, resetting some states, and so on. That's where `finally` comes in.

```
try {
  // Code might throw an error
} catch (error) {
  // Handle the error
} finally {
  // Always runs, whether an error occured or not
}
```

Let's take an example:

```
function performTask() {
  try {
    console.log("Doing something cool...");
    throw new Error("Oops!");
  } catch (err) {
    console.error("Caught error:", err.message);
  } finally {
    console.log("Cleanup: Task finished (success or fail).");
```

```
}

performTask();
```

In the `performTask()` function, the error is thrown after the first console log. So, the control will move to the catch block and log the error. After that, the `finally` block will execute its console log.

Output:

```
Doing something cool...
Caught error: Oops!
Cleanup: Task finished (success or fail).
```

Let's take a more real-world use case of making an API call and handling the loading spinner:

```
async function loadUserData() {
  showSpinner(); // Show the spinner here

  try {
    const res = await fetch('/api/user');
    const data = await res.json();
    displayUser(data);
  } catch (err) {
    showError("Failed to load user.");
  } finally {
    hideSpinner(); // Hide spinner for both success and fail cases.
  }
}
```

Usually, we show a loading spinner while making an API (asynchronous) call from the browser. Irrespective of the successful response or an error from the API call, we must stop showing the loading spinner. Instead of executing the code logic twice to stop the spinner (once inside the `try` block and then again inside the `catch` block), you can do it inside the `finally` block.

Caution with `finally`

The `finally` block can override return values or a thrown error. This behaviour may be confusing and can lead to bugs as well.

```
function test() {
  try {
    return 'from try';
  } finally {
    return 'from finally';
  }
}

console.log(test());
```

What do you think the above code returns?

It will return `'from finally'`. The return `'from try'` is completely ignored. The return from finally overrides it silently.

Let's see one more example of the same problem:

```
function willThrow() {
  try {
    throw new Error("Original Error");
  } finally {
    throw new Error("Overriding Error"); // The original error is lost
  }
}

try {
  willThrow();
} catch (err) {
  console.log(err.message); // "Overriding Error"
}
```

Here, the original error (`"Original Error"`) is swallowed. The finally block overrides the actual root cause.

When using `finally`:

- Avoid returns and throws from `finally` as much as possible.
- Avoid performing logic in the `finally` block that may impact the actual outcome. The `try` block is the best place for that.
- Any critical decision-making must be avoided inside the `finally` block
- Use `finally` for cleanup activities, such as closing files, connections, and stopping loading spinners, etc.
- Ensure the `finally` block contains side-effect-free code.

Custom Errors

Using the generic Error and its existing types, like ReferenceError, SyntaxError, and so on, can be a bit vague in complex applications. JavaScript lets you create custom errors that are more related to your business use cases. The custom errors can provide:

- Additional contextual information about the error.
- Clarity about the error
- More readable logs
- The ability to handle multiple error cases conditionally.

A custom error in JavaScript is a user-defined error type that extends the built-in Error class. The custom error should be an ES6 Class that extends JavaScript's Error class. We can use the `super()` in the constructor function to inherit the message property of the Error class. You can optionally assign a name and clean the stack trace for the custom error.

```
class MyCustomError extends Error {
  constructor(message) {
    super(message);         // Inherit message property
    this.name = this.constructor.name; // Optional but recommended
    Error.captureStackTrace(this, this.constructor); // Clean stack trace
  }
```

```
}
```

Let's now see a real-world use case for a custom error.

A Real-World Use Case of Custom Errors

Using a form on a web page is an extremely common use case. A form may contain one or more input fields. It is recommended to validate the user inputs before we process the form data for any server actions.

Let's create a custom validation error we can leverage for validating multiple form input data, like the user's email, age, phone number, and more.

First, we'll create a class called `ValidationError` that extends the `Error` class. The constructor function sets up the `ValidationError` class with an error message. We can also instantiate additional properties, like name, field, and so on.

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

Now, let's see how to use `ValidationError`. We can validate a user model to check its properties and throw a `ValidationError` whenever the expectations mismatch.

```
function validateUser(user) {
  if (!user.email.includes("@")) {
    throw new ValidationError("email", "Invalid email format");
  }
  if (!user.age || user.age < 18) {
    throw new ValidationError("age", "User must be 18+");
  }
}
```

In the code snippet above,

- We throw an invalid email format validation error if the user's email doesn't include the `@` symbol.
- We throw another validation error if the age information of the user is missing or is below 18.

A custom error gives us the power to create domain/usage-specific error types to keep the code more manageable and less error-prone.

Task Assignments for You

If you have read the handbook this far, I hope you now have a solid understanding of JavaScript Error Handling. Let's try out some assignments

based on what we have learned so far. It's going to be fun.

Find the Output

What will be the output of the following code snippet and why?

```
try {
    let r = p + 50;
    console.log(r);
} catch (error) {
    console.log("An error occurred:", error.name);
}
```

Options are:

- ReferenceError
- SyntaxError
- TypeError
- No error, it prints 10

Payment Process Validation

Write a function `processPayment(amount)` that verifies if the amount is positive and does not exceed the balance. If any condition fails, throw appropriate errors.

Hint: You can think of creating a Custom Error here.

40 Days of JavaScript Challenge Initiative

There are 101 ways of learning something. But nothing can beat structured and progressive learning methodologies. After spending more than two decades in Software Engineering, I've been able to gather the best of JavaScript together to create the 40 Days of JavaScript challenge initiative.

Check it out if you want to learn JavaScript with fundamental concepts, projects, and assignments for free (forever). Focusing on the fundamentals of JavaScript will prepare you well for a future in web development.

Before We End...

That's all! I hope you found this article insightful.

Let's connect:

- Subscribe to my YouTube Channel.
- Follow on LinkedIn if you don't want to miss the daily dose of up-skilling tips.
- Check out and follow my open-source work on GitHub.

See you soon with my next article. Until then, please take care of yourself and keep learning.

Tapas Adhikary

Co-Founder, @CreoWis | Teacher, @tapaScript | Founder, @ReactPlay | YouTuber | Writer | Human

If you read this far, thank the author to show them you care. Say Thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started