VALENTINO GAGLIARDI

Last updated: August 18, 2020 by Valentino Gagliardi - 24 minutes read

A mostly complete guide to error handling in JavaScript.

Learn how to deal with errors and exceptions in synchronous and asynchronous JavaScript code.

What is an error in programming?

Things don't go always well in our programs.

In particular, there are situations where we may want to **stop the program or inform the user if something bad happens**.

For example:

- the program tried to open a non-existent file.
- the network connection is broken.
- the user entered invalid input.

In all these cases we as programmers, create **errors**, or we let the programming engine create some for us.

After creating the error we can inform the user with a message, or we can halt the execution altogether.

What is an error in JavaScript?

An **error in JavaScript is an object**, which is later **thrown** to halt the program.

To create a new error in JavaScript we call the appropriate **constructor function**. For example, to create a new, generic error we can do:

```
const err = new Error("Something bad happened!");
```

When creating an error object it's also possible to omit the `new` keyword:

```
const err = Error("Something bad happened!");
```

Once created, the error object presents three properties:

- `message`: a string with the error message.
- `name`: the error's type.
- `stack`: a stack trace of functions execution.

For example, if we create a new `TypeError` object with the appropriate message, the `message` will carry the actual error string, while `name` will be `"TypeError"`:

```
const wrongType = TypeError("Wrong type given, expected number");

wrongType.message; // "Wrong type given, expected number"
wrongType.name; // "TypeError"
```

Firefox also implements a bunch on non-standard property like `columnNumber`, `filename`, and `lineNumber`.

Many types of errors in JavaScript

There are many types of errors in JavaScript, namely:

- `Error`
- `EvalError`
- `InternalError`
- `RangeError`
- `ReferenceError`
- `SyntaxError`
- `TypeError`
- `URIError`

Remember, all these error types are **actual constructor functions** meant to return a new error object.

In your code you'll mostly use `Error` and `TypeError`, two of the most common types, to create your own error object.

Most of the times, the majority of errors will come directly from the JavaScript engine, like `InternalError` or `SyntaxError`.

An example of `TypeError` occurs when you try to reassign `const`:

```
const name = "Jules";
name = "Caty";

// TypeError: Assignment to constant variable.
```

An example of `SyntaxError` is when you misspell language keywords:

```
va x = '33';
// SyntaxError: Unexpected identifier
```

Or when you use reserved keywords in wrong places, like `await` outside of an `async` function:

```
function wrong(){
    await 99;
```

```
}

wrong();

// SyntaxError: await is only valid in async function
```

Another example of `TypeError` occurs when we select non-existent HTML elements in the page:

```
Uncaught TypeError: button is null
```

In addition to these traditional error objects, an `AggregateError` object is going to land soon in JavaScript. `AggregateError` is convenient for wrapping multiple errors together, as we'll see later.

Besides these built-in errors, in the browser we can find also:

- `DOMException`.
- `DOMError`, deprecated and no longer used today.

`DOMException` is a family of errors related to Web APIs. They are thrown when we do silly things in the browser, like:

```
document.body.appendChild(document.cloneNode(true));
```

The result:

```
Uncaught DOMException: Node.appendChild: May not add a Document as a child
```

For a complete list see this page on MDN.

What is an exception?

Most developers think that error and exceptions are the same thing. In reality, an **error object becomes an exception only when it's thrown**.

To throw an exception in JavaScript we use `throw`, followed by the error object:

```
const wrongType = TypeError("Wrong type given, expected number");

throw wrongType;
```

The short form is more common, in most code bases you'll find:

```
throw TypeError("Wrong type given, expected number");
```

or

```
throw new TypeError("Wrong type given, expected number");
```

It's unlikely to throw exceptions outside of a function or a conditional block. Instead, consider the following example:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}
```

Here we check if the function argument is a string. If it's not we throw an exception.

Technically, you could throw anything in JavaScript, not only error objects:

```
throw Symbol();
throw 33;
throw "Error!";
throw null;
```

However, it's better to avoid these things: **always throw proper error objects, not primitives.**

By doing so you keep error handling consistent through the codebase. Other team members can always expect to access `error.message` or `error.stack` on the error object.

What happens when we throw an exception?

Exceptions are like an elevator going up: once you throw one, **it bubbles up in the program stack, unless it is caught somewhere.**

Consider the following code:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

toUppercase(4);
```

If you run this code in a browser or in Node.js, the program stops and reports the error:

```
Uncaught TypeError: Wrong type given, expected a string
    toUppercase http://localhost:5000/index.js:3
    <anonymous> http://localhost:5000/index.js:9
```

In addition, you can see the exact line where the error happened.

This report is a **stack trace**, and it's helpful for tracking down problems in your code.

The stack trace goes from bottom to top. So here:

```
    toUppercase http://localhost:5000/index.js:3
    <anonymous> http://localhost:5000/index.js:9
```

We can say:

- something in the program at line 9 called `toUppercase`
- `toUppercase` blew up at line 3

In addition to seeing this stack trace in the browser's console, you can access it on the `stack` property of the error object.

If the exception is **uncaught**, that is, nothing is done by the programmer to catch it, the program will crash.

When, and where you catch an exception in your code depends on the specific use case.

For example **you may want to propagate an exception up in the stack to crash the program altogether**. This could happen for fatal errors, when it's safer to stop the program rather than working with invalid data.

Having introduced the basics let's now turn our attention to **error and exception handling in both synchronous and asynchronous JavaScript code.**

Synchronous error handling

Synchronous code is most of the times straightforward, and so its error handling.

Error handling for regular functions

Synchronous code is executed in the same order in which is written. Let's take again the previous example:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

toUppercase(4);
```

Here the engine calls and executes `toUppercase`. All happens **synchronously**. To **catch** an exception originating by such synchronous function we can use `try/catch/finally`:

```
try {
  toUppercase(4);
} catch (error) {
  console.error(error.message);
```

```
  // or log remotely
} finally {
  // clean up
}
```

Usually, `try` deals with the happy path, or with the function call that could potentially throw.

`catch` instead, **captures the actual exception**. It **receives the error object**, which we can inspect (and send remotely to some logger in production).

The `finally` statement on the other hand runs regardless of the function's outcome: whether it failed or succeeded, any code inside `finally` will run.

Remember: `try/catch/finally` is a **synchronous** construct: **it has now way to catch exceptions coming from asynchronous code.**

Error handling for generator functions

A generator function in JavaScript is a special type of function.

It can be **paused and resumed at will**, other than providing a **bi-directional communication channel** between its inner scope and the consumer.

To create a generator function we put a star `*` after the `function` keyword:

```
function* generate() {
//
}
```

Once inside the function we can use `yield` to return values:

```
function* generate() {
  yield 33;
  yield 99;
}
```

The **return value from a generator function** is an **iterator object**. To **pull values out a generator** we can use two approaches:

- calling `next()` on the iterator object.

- **iteration** with `for...of`.

If we take our example, to get values from the generator we can do:

```
function* generate() {
  yield 33;
  yield 99;
}

const go = generate();
```

Here `go` becomes our iterator object when we call the generator function.

From now on we can call `go.next()` to advance the execution:

```
function* generate() {
  yield 33;
  yield 99;
}

const go = generate();

const firstStep = go.next().value; // 33
const secondStep = go.next().value; // 99
```

Generators also work the other way around: **they can accept values and exceptions back from the caller.**

In addition to `next()`, iterator objects returned from generators have a `throw()` method.

With this method we can halt the program by injecting an exception into the generator:

```
function* generate() {
  yield 33;
  yield 99;
}

const go = generate();

const firstStep = go.next().value; // 33
```

```
go.throw(Error("Tired of iterating!"));

const secondStep = go.next().value; // never reached
```

To catch such error you would wrap your code inside the generator with `try/catch` (and `finally` if needed):

```
function* generate() {
  try {
    yield 33;
    yield 99;
  } catch (error) {
    console.error(error.message);
  }
}
```

Generator functions can also throw exceptions to the outside. The mechanism for catching these exceptions is the same for catching synchronous exceptions: `try/catch/finally`.

Here's an example of a generator function consumed from the outside with `for...of`:

```
function* generate() {
  yield 33;
  yield 99;
  throw Error("Tired of iterating!");
}

try {
  for (const value of generate()) {
    console.log(value);
  }
} catch (error) {
  console.error(error.message);
}

/* Output:
33
99
Tired of iterating!
*/
```

Here we iterate the happy path inside a `try` block. If any exceptions occurs we stop it with `catch`.

Asynchronous error handling

JavaScript is synchronous by nature, being a single-threaded language.

Host environments like browsers engines augment JavaScript with a number of Web API for interacting with external systems, and for dealing with I/O bound operations.

Examples of asynchronicity in the browser are **timeouts, events, Promise**.

Error handling in the asynchronous world is distinct from its synchronous counterpart.

Let's see some examples.

Error handling for timers

In the beginning of your explorations with JavaScript, after learning about `try/catch/finally`, you might be tempted to put it around any block of code.

Consider the following snippet:

```
function failAfterOneSecond() {
  setTimeout(() => {
    throw Error("Something went wrong!");
  }, 1000);
}
```

This function throws after roughly 1 second. What's the right way to handle this exception?

The following example **does not work**:

```
function failAfterOneSecond() {
  setTimeout(() => {
    throw Error("Something went wrong!");
  }, 1000);
}
```

```
try {
  failAfterOneSecond();
} catch (error) {
  console.error(error.message);
}
```

As we said, `try/catch` is synchronous. On the other hand we have `setTimeout`, a browser API for timers.

By the time the callback passed to `setTimeout` runs, our `try/catch` is **long gone**. The program will crash because we failed to capture the exception.

They travel on two different tracks:

```
Track A: --> try/catch
Track B: --> setTimeout --> callback --> throw
```

If we don't want to crash the program, to handle the error correctly we must move `try/catch` inside the callback for `setTimeout`.

But, this approach doesn't make much sense most of the times. As we'll see later, **asynchronous error handling with Promises provides a better ergonomic.**

Error handling for events

HTML nodes in the Document Object Model are connected to `EventTarget`, the common ancestor for any event emitter in the browser.

That means we can listen for events on any HTML element in the page.

(Node.js will support `EventTarget` in a future release).

The **error handling mechanics for DOM events follows the same scheme of any asynchronous** Web API.

Consider the following example:

```
const button = document.querySelector("button");

button.addEventListener("click", function() {
```

```
  throw Error("Can't touch this button!");
});
```

Here we throw an exception as soon as the button is clicked. How do we catch it? This pattern **does not work**, and won't prevent the program from crashing:

```
const button = document.querySelector("button");

try {
  button.addEventListener("click", function() {
    throw Error("Can't touch this button!");
  });
} catch (error) {
  console.error(error.message);
}
```

As with the previous example with `setTimeout`, any callback passed to `addEventListener` is executed **asynchronously**:

```
Track A: --> try/catch
Track B: --> addEventListener --> callback --> throw
```

If we don't want to crash the program, to handle the error correctly we must move `try/catch` inside the callback for `addEventListener`.

But again, there's little of no value in doing this.

As with `setTimeout`, exception thrown by an asynchronous code path are **un-catchable** from the outside, and will crash your program.

In the next sections we'll see how Promises and `async/await` can ease error handling for asynchronous code.

How about onerror?

HTML elements have a number of event handlers like `onclick`, `onmouseenter`, `onchange` to name a few.

There is also `onerror`, but it has nothing to do with `throw` and friends.

The `onerror` event handler fires any time an HTML element like an `<img>` tag or a `<script>` hits a non-existent resource.

Consider the following example:

```
// omitted
<body>
<img src="nowhere-to-be-found.png" alt="So empty!">
</body>
// omitted
```

When visiting an HTML document with a missing or non-existent resource the browser's console records the error:

```
GET http://localhost:5000/nowhere-to-be-found.png
[HTTP/1.1 404 Not Found 3ms]
```

In our JavaScript we have the chance to "catch" this error with the appropriate event handler:

```
const image = document.querySelector("img");

image.onerror = function(event) {
  console.log(event);
};
```

Or better:

```
const image = document.querySelector("img");

image.addEventListener("error", function(event) {
  console.log(event);
});
```

This pattern is useful for **loading alternate resources in place of missing images or scripts.**

But remember: `onerror`, has nothing to do with `throw` or `try/catch`.

Error handling with Promise

To illustrate error handling with Promise we'll "promisify" one of our original examples. We tweak the following function:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

toUppercase(4);
```

Instead of returning a simple string, or an exception, we use respectively `Promise.reject` and `Promise.resolve` to handle error and success:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    return Promise.reject(TypeError("Wrong type given, expected a string"));
  }

  const result = string.toUpperCase();

  return Promise.resolve(result);
}
```

(Technically there's nothing asynchronous in this code, but it serves well to illustrate the point).

Now that the function is "promisified" we can attach `then` for consuming the result, and `catch` for **handling the rejected Promise**:

```
toUppercase(99)
  .then(result => result)
  .catch(error => console.error(error.message));
```

This code will log:

```
Wrong type given, expected a string
```

In the realm of Promise, `catch` is the construct for handling errors.

In addition to `catch` and `then` we have also `finally`, similar to the `finally` in `try/catch`.

As its synchronous "relative", Promise's `finally` runs **regardless** of the Promise outcome:

```
toUppercase(99)
  .then(result => result)
  .catch(error => console.error(error.message))
  .finally(() => console.log("Run baby, run"));
```

Always keep in mind that any callback passed to `then/catch/finally` is handled asynchronously by the Microtask Queue. They're **micro tasks** with precedence over macro tasks such as events and timers.

Promise, error, and throw

As a **best practice when rejecting a Promise** it's convenient to provide an error object:

```
Promise.reject(TypeError("Wrong type given, expected a string"));
```

By doing so you keep error handling consistent through the codebase. Other team members can always expect to access `error.message`, and more important you can inspect stack traces.

In addition to `Promise.reject`, we can exit from a Promise chain by **throwing** an exception.

Consider the following example:

```
Promise.resolve("A string").then(value => {
  if (typeof value === "string") {
    throw TypeError("Expected a number!");
  }
});
```

We resolve a Promise with a string, and then the chain is immediately broken with `throw`.

To stop the exception propagation we use `catch`, as usual:

```
Promise.resolve("A string")
```

```
  .then(value => {
    if (typeof value === "string") {
      throw TypeError("Expected a number!");
    }
  })
  .catch(reason => console.log(reason.message));
```

This pattern is common in `fetch`, where we check the response object in search for errors:

```
fetch("https://example-dev/api/")
  .then(response => {
    if (!response.ok) {
      throw Error(response.statusText);
    }

    return response.json();
  })
  .then(json => console.log(json));
```

Here the exception can be intercepted with `catch`. If we fail, or decide to not catch it there, **the exception is free to bubble up in the stack.**

This is not bad per-se, but different environments react differently to uncaught rejections.

Node.js for example in the future will let crash any program where Promise rejections are un-handled:

```
DeprecationWarning: Unhandled promise rejections are deprecated. In the future
```

Better you catch them!

Error handling for "promisified" timers

With timers or events it's not possible to catch exceptions thrown from a callback. We saw an example in the previous section:

```
function failAfterOneSecond() {
  setTimeout(() => {
```

```
    throw Error("Something went wrong!");
  }, 1000);
}

// DOES NOT WORK
try {
  failAfterOneSecond();
} catch (error) {
  console.error(error.message);
}
```

A solution offered by Promise consists in the "promisification" of our code. Basically, we wrap our timer with a Promise:

```
function failAfterOneSecond() {
  return new Promise((_, reject) => {
    setTimeout(() => {
      reject(Error("Something went wrong!"));
    }, 1000);
  });
}
```

With `reject` we set off a Promise rejection, which carries an error object.

At this point we can handle the exception with `catch`:

```
failAfterOneSecond().catch(reason => console.error(reason.message));
```

Note: it's common to use `value` as the returning value from a Promise, and `reason` as the returning object from a rejection.

Node.js has a utility called promisify to ease the "promisification" of old-style callback APIs.

Error handling in Promise.all

The static method `Promise.all` accepts an array of Promise, and returns an array of results from all resolving Promise:

```
const promise1 = Promise.resolve("All good!");
```

```
const promise2 = Promise.resolve("All good here too!");

Promise.all([promise1, promise2]).then((results) => console.log(results));

// [ 'All good!', 'All good here too!' ]
```

If any of these Promise rejects, `Promise.all` rejects with the error from the first rejected Promise.

To handle these situations in `Promise.all` we use `catch`, as we did in the previous section:

```
const promise1 = Promise.resolve("All good!");
const promise2 = Promise.reject(Error("No good, sorry!"));
const promise3 = Promise.reject(Error("Bad day ..."));

Promise.all([promise1, promise2, promise3])
  .then(results => console.log(results))
  .catch(error => console.error(error.message));
```

To run a function regardless of the outcome of `Promise.all`, again, we can use `finally`:

```
Promise.all([promise1, promise2, promise3])
  .then(results => console.log(results))
  .catch(error => console.error(error.message))
  .finally(() => console.log("Always runs!"));
```

Error handling in Promise.any

We can consider `Promise.any` (Firefox > 79, Chrome > 85) as the opposite of `Promise.all`.

Whereas `Promise.all` returns a failure even if a single Promise in the array rejects, `Promise.any` gives us always the first resolved Promise (if present in the array) regardless of any rejection occurred.

In case instead **all the Promise** passed to `Promise.any` reject, the resulting error is an `AggregateError`. Consider the following example:

```
const promise1 = Promise.reject(Error("No good, sorry!"));
```

```
const promise2 = Promise.reject(Error("Bad day ..."));

Promise.any([promise1, promise2])
  .then(result => console.log(result))
  .catch(error => console.error(error))
  .finally(() => console.log("Always runs!"));
```

Here we handle the error with `catch`. The output of this code is:

```
AggregateError: No Promise in Promise.any was resolved
Always runs!
```

The `AggregateError` object has the same properties of a basic `Error`, plus an `errors` property:

```
//
  .catch(error => console.error(error.errors))
//
```

This property is an array of each individual error produced by the reject:

```
[Error: "No good, sorry!, Error: "Bad day ..."]
```

Error handling in Promise.race

The static method `Promise.race` accepts an array of Promise:

```
const promise1 = Promise.resolve("The first!");
const promise2 = Promise.resolve("The second!");

Promise.race([promise1, promise2]).then(result => console.log(result));

// The first!
```

The result is **the first Promise who wins the "race"**.

How about rejections then? If the rejecting Promise is not the first to appear in the input array,

`Promise.race` resolves:

```
const promise1 = Promise.resolve("The first!");
const rejection = Promise.reject(Error("Ouch!"));
const promise2 = Promise.resolve("The second!");

Promise.race([promise1, rejection, promise2]).then(result =>
  console.log(result)
);

// The first!
```

If **the rejection instead appears as the first element of the array, `Promise.race` rejects**, and we must catch the rejection:

```
const promise1 = Promise.resolve("The first!");
const rejection = Promise.reject(Error("Ouch!"));
const promise2 = Promise.resolve("The second!");

Promise.race([rejection, promise1, promise2])
  .then(result => console.log(result))
  .catch(error => console.error(error.message));

// Ouch!
```

Error handling in Promise.allSettled

`Promise.allSettled` is an ECMAScript 2020 addition to the language.

There is not so much to handle with this static method since **the result will always be a resolved Promise, even if one or more input Promise rejects**.

Consider the following example:

```
const promise1 = Promise.resolve("Good!");
const promise2 = Promise.reject(Error("No good, sorry!"));

Promise.allSettled([promise1, promise2])
  .then(results => console.log(results))
```

```
  .catch(error => console.error(error))
  .finally(() => console.log("Always runs!"));
```

We pass to `Promise.allSettled` an array consisting of two Promise: one resolved and another rejected.

In this case `catch` will never be hit. `finally` instead runs.

The result of this code, logged in `then` is:

```
[
  { status: 'fulfilled', value: 'Good!' },
  {
    status: 'rejected',
    reason: Error: No good, sorry!
  }
]
```

Error handling for async/await

`async/await` in JavaScript denotes asynchronous functions, but from a reader standpoint they benefit from all the **readability** of synchronous functions.

To keep things simple we'll take our previous synchronous function `toUppercase`, and we transform it to an asynchronous function by putting `async` before the `function` keyword:

```
async function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}
```

Just by prefixing a function with `async` we cause the function to **return a Promise**. That means we can chain `then`, `catch`, and `finally` after the function call:

```
async function toUppercase(string) {
```

```
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

toUppercase("abc")
  .then(result => console.log(result))
  .catch(error => console.error(error.message))
  .finally(() => console.log("Always runs!"));
```

When we throw from an `async` function the exception becomes cause of rejection for the underlying Promise.

Any error can be intercepted with `catch` from the outside.

Most important, in **addition to this style we can use `try/catch/finally`**, much as we would do with a synchronous function.

In the following example we call `toUppercase` from another function, `consumer`, which conveniently wraps the function call with `try/catch/finally`:

```
async function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

async function consumer() {
  try {
    await toUppercase(98);
  } catch (error) {
    console.error(error.message);
  } finally {
    console.log("Always runs!");
  }
}

consumer(); // Returning Promise ignored
```

The output is:

```
Wrong type given, expected a string
Always runs!
```

On the same topic: How to Throw Errors From Async Functions in JavaScript?

Error handling for async generators

Async generators in JavaScript are **generator functions capable of yielding Promises** instead of simple values.

They combine generator functions with `async`. The result is a generator function whose iterator objects expose a Promise to the consumer.

To create an async generator we declare a generator function with the star `*`, prefixed with `async`:

```
async function* asyncGenerator() {
  yield 33;
  yield 99;
  throw Error("Something went wrong!"); // Promise.reject
}
```

Being based on Promise, the same rules for error handling apply here. `throw` inside an async generator causes a Promise rejection, which we intercept with `catch`.

To **pull Promises out an async generators** we can use two approaches:

- `then` handlers.
- **async iteration**.

From the above example we know for sure there will be an exception after the first two `yield`. This means we can do:

```
const go = asyncGenerator();

go.next().then(value => console.log(value));
go.next().then(value => console.log(value));
```

```
go.next().catch(reason => console.error(reason.message));
```

The output from this code is:

```
{ value: 33, done: false }
{ value: 99, done: false }
Something went wrong!
```

The other approach uses **async iteration** with `for await...of`. To use async iteration we need to wrap the consumer with an `async` function.

Here's the complete example:

```
async function* asyncGenerator() {
  yield 33;
  yield 99;
  throw Error("Something went wrong!"); // Promise.reject
}

async function consumer() {
  for await (const value of asyncGenerator()) {
    console.log(value);
  }
}

consumer();
```

And as with `async/await` we handle any potential exception with `try/catch`:

```
async function* asyncGenerator() {
  yield 33;
  yield 99;
  throw Error("Something went wrong!"); // Promise.reject
}

async function consumer() {
  try {
    for await (const value of asyncGenerator()) {
      console.log(value);
```

```
    }
  } catch (error) {
    console.error(error.message);
  }
}

consumer();
```

The output of this code is:

```
33
99
Something went wrong!
```

The iterator object returned from an asynchronous generator function has also a `throw()` method, much like its synchronous counterpart.

Calling `throw()` on the iterator object here won't throw an exception, but a Promise rejection:

```
async function* asyncGenerator() {
  yield 33;
  yield 99;
  yield 11;
}

const go = asyncGenerator();

go.next().then(value => console.log(value));
go.next().then(value => console.log(value));

go.throw(Error("Let's reject!"));

go.next().then(value => console.log(value)); // value is undefined
```

To handle this situation from the outside we can do:

```
go.throw(Error("Let's reject!")).catch(reason => console.error(reason.message)
```

But let's not forget that iterator objects `throw()` send the exception **inside the generator**.

This means we can also apply the following pattern:

```
async function* asyncGenerator() {
  try {
    yield 33;
    yield 99;
    yield 11;
  } catch (error) {
    console.error(error.message);
  }
}

const go = asyncGenerator();

go.next().then(value => console.log(value));
go.next().then(value => console.log(value));

go.throw(Error("Let's reject!"));

go.next().then(value => console.log(value)); // value is undefined
```

Error handling in Node.js

Synchronous error handling in Node.js

Synchronous error handling in Node.js does not differ too much from what we saw so far.

For **synchronous code**, `try/catch/finally` works fine.

However, things get interesting if we take a glance at the asynchronous world.

Asynchronous error handling in Node.js: the callback pattern

For asynchronous code, Node.js strongly relies on two idioms:

- the callback pattern.
- event emitters.

In the **callback pattern, asynchronous Node.js APIs** accept a function which is handled through the **event loop** and executed as soon as the **call stack** is empty.

Consider the following code:

```
const { readFile } = require("fs");

function readDataset(path) {
  readFile(path, { encoding: "utf8" }, function(error, data) {
    if (error) console.error(error);
    // do stuff with the data
  });
}
```

If we extract the callback from this listing, we can see how it is supposed to deal with errors:

```
//
function(error, data) {
    if (error) console.error(error);
    // do stuff with the data
  }
//
```

If any errors arises from reading the given path with `fs.readFile`, we get an error object.

At this point we can:

- simply log the error object as we did.
- throw an exception.
- pass the error to another callback.

To throw an exception we can do:

```
const { readFile } = require("fs");

function readDataset(path) {
  readFile(path, { encoding: "utf8" }, function(error, data) {
    if (error) throw Error(error.message);
    // do stuff with the data
  });
```

```
}
```

However, as with events and timers in the DOM, this exception will **crash the program**. The following attempt to stop it with `try/catch` won't work:

```
const { readFile } = require("fs");

function readDataset(path) {
  readFile(path, { encoding: "utf8" }, function(error, data) {
    if (error) throw Error(error.message);
    // do stuff with the data
  });
}

try {
  readDataset("not-here.txt");
} catch (error) {
  console.error(error.message);
}
```

Passing the error to another callback is the preferred option, if we don't want to crash the program:

```
const { readFile } = require("fs");

function readDataset(path) {
  readFile(path, { encoding: "utf8" }, function(error, data) {
    if (error) return errorHandler(error);
    // do stuff with the data
  });
}
```

Here `errorHandler` is what the name suggests, a simple function for error handling:

```
function errorHandler(error) {
  console.error(error.message);
  // do something with the error:
  // - write to a log.
  // - send to an external logger.
```

```
}
```

Asynchronous error handling in Node.js: event emitters

Much of what you do in Node.js is based on **events**. Most of the times you interact with an **emitter object** and some observers listening for messages.

Any event-driven module (like net for example) in Node.js extends a root class named `EventEmitter`.

`EventEmitter` in Node.js has two fundamental methods: `on` and `emit`.

Consider this simple HTTP server:

```
const net = require("net");

const server = net.createServer().listen(8081, "127.0.0.1");

server.on("listening", function () {
  console.log("Server listening!");
});

server.on("connection", function (socket) {
  console.log("Client connected!");
  socket.end("Hello client!");
});
```

Here we listen for two events: **listening** and **connection**.

In addition to these events, event emitters expose also an **error** event, fired in case of errors.

If you run this code listening on port 80 instead of the previous example, you'll get an exception:

```
const net = require("net");

const server = net.createServer().listen(80, "127.0.0.1");

server.on("listening", function () {
  console.log("Server listening!");
```

```
});

server.on("connection", function (socket) {
  console.log("Client connected!");
  socket.end("Hello client!");
});
```

Output:

```
events.js:291
      throw er; // Unhandled 'error' event
      ^

Error: listen EACCES: permission denied 127.0.0.1:80
Emitted 'error' event on Server instance at: ...
```

To catch it we can register an event handler for **error**:

```
server.on("error", function(error) {
  console.error(error.message);
});
```

This will print:

```
listen EACCES: permission denied 127.0.0.1:80
```

In addition, the program won't crash.

To learn more on the topic, consider also reading "Error handling in Node.js".

Wrapping up

In this guide we covered **error handling in JavaScript for the whole spectrum**, from simple synchronous code, to advanced asynchronous primitives.

There are many ways in which an exception can manifest in our JavaScript programs.

Exceptions from synchronous code are the most straightforward to catch. **Exceptions** from

asynchronous code paths instead can be **tricky** to deal with.

In the meantime, new JavaScript APIs in the browser are almost all headed toward `Promise`. This pervasive pattern makes easier to handle exceptions with `then/catch/finally`, or with `try/catch` for `async/await`.

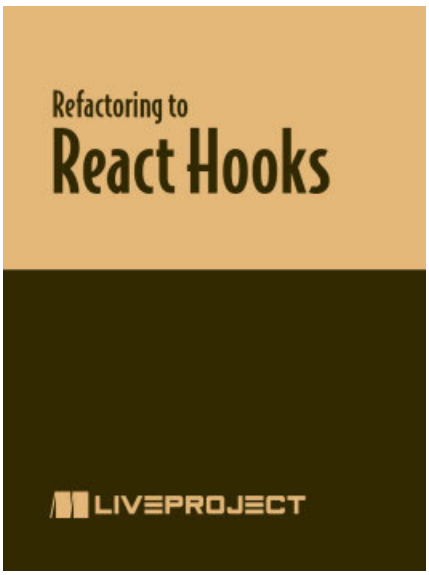
After reading this guide you should be **able to recognize all the different situations** that may arise in your programs, and catch your **exceptions** correctly.

Thanks for reading and stay tuned!

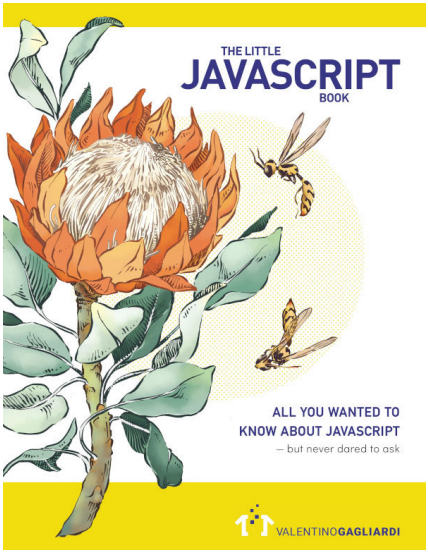
This post is also available in:

- Japanese
- Chinese

Refactoring to React Hooks



The Little JavaScript Book



Hi! I'm Valentino! Educator and consultant, I help people learning to code with on-site and remote workshops. Looking for JavaScript and Python training? Let's get in touch!

More from the blog:

- var, let, and const in JavaScript: a cheatsheet.
- How to build an URL and its search parameters with JavaScript
- You might not need switch in JavaScript

- How babel preset-env, core-js, and browserslistrc work together
- Code documentation for JavaScript with JSDoc: an introduction
- Cos'è JavaScript e perché studiarlo?
- A look at generator functions and asynchronous generators in JavaScript.
- All I need to know about ECMAScript modules
- What does it mean "event-driven" in JavaScript and Node.js?
- Formatting dates in JavaScript with Intl.DateTimeFormat