

Test a deployment on our modern App Hosting. For a limited time, your first \$20 is on us.

JAVASCRIPT TUTORIALS

# A Definitive Guide to Handling Errors in JavaScript

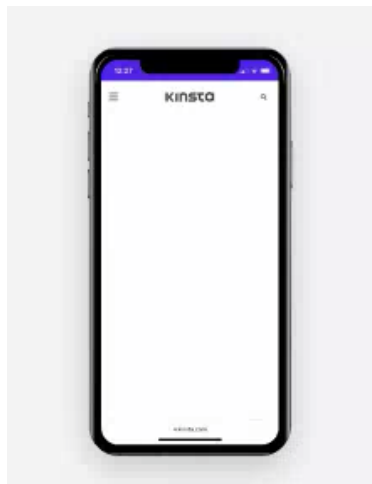
Kumar Harsh, October 26, 2022



 Shares   

Murphy's law states that whatever can go wrong will eventually go wrong. This applies a tad too well in the world of programming. If you create an application, chances are you'll create bugs and other issues. Errors in JavaScript are one such common issue!

A software product's success depends on how well its creators can resolve these issues before hurting their users. And [JavaScript](#), out of all [programming languages](#), is notorious for its average error handling design.



## In a hurry? Save this article as a PDF.

Tired of scrolling? Download a PDF version for easier offline reading and sharing with coworkers.

[Download](#)

If you're [building a JavaScript application](#), there's a high chance you'll mess up with data types at one point or another. If not that, then you might end up replacing an undefined with a null or a triple equals operator ( `===` ) with a double equals operator ( `==` ).

It's only human to make mistakes. This is why we will show you everything you need to know about handling errors in JavaScript.

This article will guide you through the basic errors in JavaScript and explain the various errors you might encounter. You'll then learn how to identify and fix these errors. There are also a couple of tips to handle errors effectively in production environments.

Without further ado, let's begin!

## Table of Contents

- [What Are JavaScript Errors?](#)
- [Types of Errors in JavaScript](#)
- [Creating Custom Error Types](#)
- [Top 10 Most Common Errors in JavaScript](#)
- [How to Identify and Prevent Errors in JavaScript](#)
- [Best Practices for Handling Errors in JavaScript](#)

## Check Out Our Video Guide to [Handling JavaScript Errors](#)



**See how Kinsta stacks up against the competition.**

Select your provider

**Compare**

## What Are JavaScript Errors?

Errors in programming refer to situations that don't let a program function normally. It can happen when a program doesn't know how to handle the job at hand, such as when trying to open a non-existent file or reaching out to a web-based API endpoint while there's no network connectivity.

These situations push the program to throw errors to the user, stating that it doesn't know how to proceed. The program collects as much information as possible about the error and

then reports that it can not move ahead.

“**Murphy's law states that whatever can go wrong will eventually go wrong 😬 This applies a bit too well in the world of JavaScript 😊 Get prepped with this guide 📌**

**CLICK TO TWEET**

Intelligent programmers try to predict and cover these scenarios so that the user doesn't have to figure out a [technical error message like "404"](#) independently. Instead, they show a much more understandable message: "The page could not be found."

Errors in JavaScript are objects shown whenever a programming error occurs. These objects contain ample information about the type of the error, the statement that caused the error, and the stack trace when the error occurred. JavaScript also allows programmers to create custom errors to provide extra information when debugging issues.



## Your new developer platform to build, deploy, host, and scale.

Time is money. Code needs to be shipped; yesterday. With [Application Hosting](#) we provide CTOs, engineers, and developers with solid infrastructure that is fast to set up, already tested, reliable and ready to scale.

**Try it for free**

### Properties of an Error

Now that the definition of a JavaScript error is clear, it's time to dive into the details.

Errors in JavaScript carry certain standard and custom properties that help understand the cause and effects of the error. By default, errors in JavaScript contain three properties:

1. **message:** A string value that carries the error message
2. **name:** The type of error that occurred (We'll dive deep into this in the next section)
3. **stack:** The stack trace of the code executed when the error occurred.

Additionally, errors can also carry properties like `columnNumber`, `lineNumber`, `fileName`, etc., to describe the error better. However, these properties are not standard and may or may not be

present in every error object generated from your JavaScript application.

## Understanding Stack Trace

A stack trace is the list of method calls a program was in when an event such as an exception or a warning occurs. This is what a sample stack trace accompanied by an exception looks like:

```
TypeError: Numeric argument is expected
    at Timeout.setTimeout [as _onTimeout] (/tmp/rWzN9KhDTV.js:5:10)
    at ontimeout (timers.js:436:11)
    at tryOnTimeout (timers.js:300:5)
    at listOnTimeout (timers.js:263:5)
    at Timer.processTimers (timers.js:223:10)
```

— Example of a Stack Trace.

As you can see, it starts by printing the error name and message, followed by a list of methods that were being called. Each method call states the location of its source code and the line at which it was invoked. You can use this data to navigate through your codebase and identify which piece of code is causing the error.

This list of methods is arranged in a stacked fashion. It shows where your exception was first thrown and how it propagated through the stacked method calls. Implementing a catch for the exception will not let it propagate up through the stack and crash your program. However, you might want to leave fatal errors uncaught to crash the program in some scenarios intentionally.

## Errors vs Exceptions

Most people usually consider errors and exceptions as the same thing. However, it's essential to note a slight yet fundamental difference between them.

An exception is an error object that has been thrown.

To understand this better, let's take a quick example. Here is how you can define an error in JavaScript:

```
const wrongTypeError = TypeError("Wrong type found, expected")
```

And this is how the `wrongTypeError` object becomes an exception:

```
throw wrongTypeError
```

However, most people tend to use the shorthand form which defines error objects while throwing them:

```
throw TypeError("Wrong type found, expected character")
```

This is standard practice. However, it's one of the reasons why developers tend to mix up exceptions and errors. Therefore, knowing the fundamentals is vital even though you use shorthands to get your work done quickly.

## Types of Errors in JavaScript

There's a range of predefined error types in JavaScript. They are automatically chosen and defined by the JavaScript runtime whenever the programmer doesn't explicitly handle errors in the application.

This section will walk you through some of the most common types of errors in JavaScript and understand when and why they occur.

### RangeError

A `RangeError` is thrown when a variable is set with a value outside its legal values range. It usually occurs when passing a value as an argument to a function, and the given value doesn't lie in the range of the function's parameters. It can sometimes get tricky to fix when using poorly documented third-party libraries since you need to know the range of possible values for the arguments to pass in the correct value.

Some of the common scenarios in which `RangeError` occurs are:

- Trying to create an array of illegal lengths via the `Array` constructor.
- Passing bad values to numeric methods like `toExponential()`, `toPrecision()`, `toFixed()`, etc.
- Passing illegal values to string functions like `normalize()`.

### ReferenceError

A `ReferenceError` occurs when something is wrong with a variable's reference in your code. You might have forgotten to define a value for the variable before using it, or you might be trying to use an inaccessible variable in your code. In any case, going through the stack trace provides ample information to find and fix the variable reference that is at fault.

Some of the common reasons why `ReferenceErrors` occur are:

- Making a typo in a variable name.
- Trying to access block-scoped variables outside of their scopes.
- Referencing a global variable from an external library (like `$` from `jQuery`) before it's loaded.

### SyntaxError

These errors are one of the simplest to fix since they indicate an error in the syntax of the code. Since JavaScript is a [scripting language](#) that is interpreted rather than compiled, these are thrown when the app executes the script that contains the error. In the case of compiled languages, such errors are identified during compilation. Thus, the app binaries are not created until these are fixed.

Some of the common reasons why `SyntaxErrors` might occur are:

- Missing inverted commas
- Missing closing parentheses
- Improper alignment of curly braces or other characters

It's a good practice to use a linting tool in your IDE to identify such errors for you before they hit the browser.

## **TypeError**

`TypeError` is one of the most common errors in JavaScript apps. This error is created when some value doesn't turn out to be of a particular expected type. Some of the common cases when it occurs are:

- Invoking objects that are not methods.
- Attempting to access properties of null or undefined objects
- Treating a string as a number or vice versa

There are a lot more possibilities where a `TypeError` can occur. We'll look at some famous instances later and learn how to fix them.

## **InternalError**

The `InternalError` type is used when an exception occurs in the JavaScript runtime engine. It may or may not indicate an issue with your code.

More often than not, `InternalError` occurs in two scenarios only:

- When a patch or an update to the JavaScript runtime carries a bug that throws exceptions (this happens very rarely)
- When your code contains entities that are too large for the JavaScript engine (e.g. too many switch cases, too large array initializers, too much recursion)

The most appropriate approach to solve this error is to identify the cause via the error message and restructure your app logic, if possible, to eliminate the sudden spike of workload on the JavaScript engine.

## **URIError**

`URIError` occurs when a global URI handling function such as `decodeURIComponent` is used illegally. It usually indicates that the parameter passed to the method call did not conform to URI standards and thus was not [parsed by the method properly](#).

Diagnosing these errors is usually easy since you only need to examine the arguments for malformation.

## **EvalError**

An `EvalError` occurs when an error occurs with an `eval()` function call. The `eval()` function is used to execute JavaScript code stored in strings. However, since using the `eval()` function is highly discouraged due to security issues and the current ECMAScript specifications don't throw the `EvalError` class anymore, this error type exists simply to maintain backward compatibility with legacy JavaScript code.

If you're working on an older version of JavaScript, you might encounter this error. In any case, it's best to investigate the code executed in the `eval()` function call for any exceptions.

# **Creating Custom Error Types**

While JavaScript offers an adequate list of error type classes to cover for most scenarios, you can always create a new error type if the list doesn't satisfy your requirements. The foundation

of this flexibility lies in the fact that JavaScript allows you to throw anything literally with the `throw` command.

So, technically, these statements are entirely legal:

```
throw 8  
throw "An error occurred"
```

However, throwing a primitive data type doesn't provide details about the error, such as its type, name, or the accompanying stack trace. To fix this and standardize the error handling process, the `Error` class has been provided. It's also discouraged to use primitive data types while throwing exceptions.

You can extend the `Error` class to create your custom error class. Here is a basic example of how you can do this:

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

And you can use it in the following way:

```
throw ValidationError("Property not found: name")
```

And you can then identify it using the `instanceof` keyword:

```
try {  
  validateForm() // code that throws a ValidationError  
} catch (e) {  
  if (e instanceof ValidationError)
```

```
// do something  
  
else  
  
// do something else  
  
}
```

## Top 10 Most Common Errors in JavaScript

Now that you understand the common error types and how to create your custom ones, it's time to look at some of the most common errors you'll face when writing JavaScript code.

**Check Out Our Video Guide to [The Most Common JavaScript Errors](#)**

### 1. Uncaught RangeError

This error occurs in Google Chrome under a few various scenarios. First, it can happen if you call a recursive function and it doesn't terminate. You can check this out yourself in the Chrome Developer Console:

```
> var arr = new Array(1)  
    const recurfn = (arr) => {  
      arr[0] = new Array(1)  
      recurfn(arr[0])  
    }  
  
    recurfn(arr)
```

✖ ▶ Uncaught RangeError: Maximum call stack size exceeded [VM340:3](#)

- at recurfn (<anonymous>:3:14)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)
- at recurfn (<anonymous>:4:5)

— RangeError example with a recursive function call.

So to solve such an error, make sure to define the border cases of your recursive function correctly. Another reason why this error happens is if you have passed a value that is out of a function's parameter's range. Here's an example:

```
> var num = 4  
    num.toExponential(-2)
```

✖ ▶ Uncaught RangeError: toExponential() argument must be between 0 and 100 [VM395:2](#)

- at Number.toExponential (<anonymous>)
- at <anonymous>:2:5



— RangeError example with toExponential() call.

The error message will usually indicate what is wrong with your code. Once you make the changes, it will be resolved.

```
> var num = 4  
  num.toExponential(2)  
◀ '4.00e+0'
```

— Output for the toExponential() function call.

## 2. Uncaught TypeError: Cannot set property

This error occurs when you set a property on an undefined reference. You can reproduce the issue with this code:

```
var list  
list.count = 0
```

Here's the output that you'll receive:

```
> var list  
  list.count = 0  
✖ ▶ Uncaught TypeError: Cannot set properties of undefined (setting 'count') VM49:2  
  at <anonymous>:2:12
```

— TypeError example.

To fix this error, initialize the reference with a value before accessing its properties. Here's how it looks when fixed:

```
> var list = {}  
  list.count = 10  
  console.log(list.count)  
10 VM241:3
```

— How to fix TypeError.

## 3. Uncaught TypeError: Cannot read property

This is one of the most frequently occurring errors in JavaScript. This error occurs when you attempt to read a property or call a function on an undefined object. You can reproduce it very easily by running the following code in a Chrome Developer console:

```
var func  
func.call()
```

Here's the output:

```
> var func  
func.call()  
✖ ▶ Uncaught TypeError: Cannot read properties of undefined (reading 'call')  
   at <anonymous>:2:6 VM37:2
```

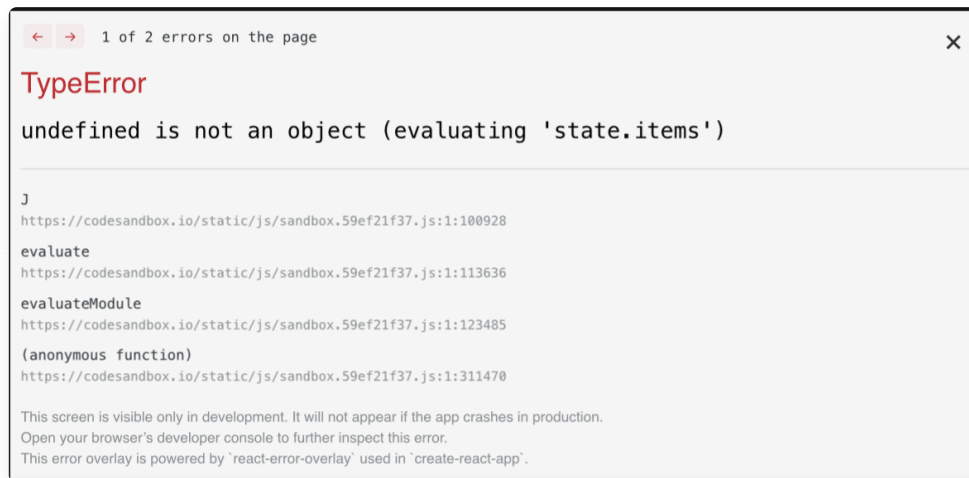
— TypeError example with undefined function.

An undefined object is one of the many possible causes of this error. Another prominent cause of this issue can be an improper initialization of the state while rendering the UI. Here's a real-world example from a React application:

```
import React, { useState, useEffect } from "react";  
  
const CardsList = () => {  
  
  const [state, setState] = useState();  
  
  useEffect(() => {  
    setTimeout(() => setState({ items: ["Card 1", "Card 2"], []});  
  }, []);  
  
  return (  
    <>  
      {state.items.map((item) => (  
        <li key={item}>{item}</li>  
      ))}  
    </>  
  );  
};  
  
export default CardsList;
```

The app starts with an empty state container and is provided with some items after a delay of

2 seconds. The delay is put in place to imitate a network call. Even if your network is super fast, you'll still face a minor delay due to which the component will render at least once. If you try to run this app, you'll receive the following error:



— TypeError stack trace in a browser.

This is because, at the time of rendering, the state container is undefined; thus, there exists no property `items` on it. Fixing this error is easy. You just need to provide an initial default value to the state container.

```
// ...  
const [state, setState] = useState({items: []});  
// ...
```

Now, after the set delay, your app will show a similar output:

- Card 1
- Card 2

— Code output.

The exact fix in your code might be different, but the essence here is to always initialize your variables properly before using them.

#### 4. TypeError: 'undefined' is not an object

This error occurs in Safari when you try to access the properties of or call a method on an undefined object. You can run the same code from above to reproduce the error yourself.

```
> var func  
  func.call()  
! ▶ TypeError: undefined is not an object (evaluating 'func.call')  
> |
```

Auto — Page ↕

— TypeError example with undefined function.

The solution to this error is also the same — make sure that you have initialized your variables correctly and they are not undefined when a property or method is accessed.

#### 5. TypeError: null is not an object

This is, again, similar to the previous error. It occurs on Safari, and the only difference between the two errors is that this one is thrown when the object whose property or method is being accessed is `null` instead of `undefined`. You can reproduce this by running the following piece of code:

```
var func = null  
  
func.call()
```

Here's the output that you'll receive:

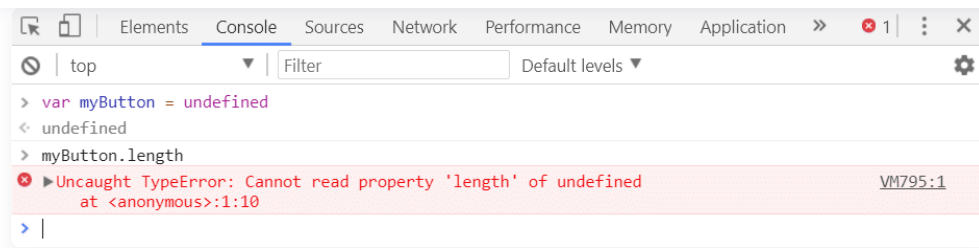
```
> var func=null  
  func.call()  
! ▶ TypeError: null is not an object (evaluating 'func.call')
```

— TypeError example with null function.

Since `null` is a value explicitly set to a variable and not assigned automatically by JavaScript. This error can occur only if you're trying to access a variable you set `null` by yourself. So, you need to revisit your code and check if the logic that you wrote is correct or not.

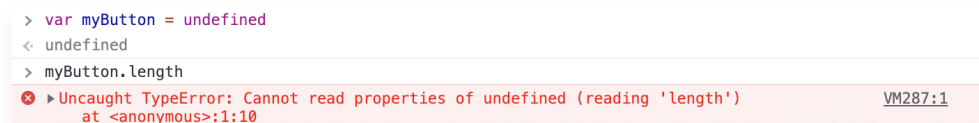
#### 6. TypeError: Cannot read property 'length'

This error occurs in Chrome when you try to read the length of a `null` or `undefined` object. The cause of this issue is similar to the previous issues, but it occurs quite frequently while handling lists; hence it deserves a special mention. Here's how you can reproduce the problem:



— TypeError example with an undefined object.

However, in the newer versions of Chrome, this error is reported as **Uncaught TypeError: Cannot read properties of undefined**. This is how it looks now:



— TypeError example with an undefined object on newer Chrome versions.

The fix, again, is to ensure that the object whose length you're trying to access exists and is not set to **null**.

## 7. TypeError: 'undefined' is not a function

This error occurs when you try to invoke a method that doesn't exist in your script, or it does but can not be referenced in the calling context. This error usually occurs in Google Chrome, and you can solve it by checking the line of code throwing the error. If you find a typo, fix it and check if it solves your issue.

If you have used the self-referencing keyword **this** in your code, this error might arise if **this** is not appropriately bound to your context. Consider the following code:

```
function showAlert() {  
    alert("message here")  
}  
  
document.addEventListener("click", () => {  
    this.showAlert();  
}))
```

If you execute the above code, it will throw the error we discussed. It happens because the anonymous function passed as the event listener is being executed in the context of the **document**.

In contrast, the function **showAlert** is defined in the context of the **window**.

To solve this, you must pass the proper reference to the function by binding it with the

`bind()` method:

## Want to know how we increased our traffic over 1000%?

Join 20,000+ others who get our weekly newsletter with insider WordPress tips!

Subscribe Now

```
document.addEventListener("click", this.showAlert.bind(this))
```

### 8. ReferenceError: event is not defined

This error occurs when you try to access a reference not defined in the calling scope. This usually happens when handling events since they often provide you with a reference called `event` in the callback function. This error can occur if you forget to define the event argument in your function's parameters or misspell it.

This error might not occur in Internet Explorer or Google Chrome (as IE offers a global event variable and Chrome attaches the event variable automatically to the handler), but it can occur in Firefox. So it's advisable to keep an eye out for such small mistakes.

### 9. TypeError: Assignment to constant variable

This is an error that arises out of carelessness. If you try to assign a new value to a constant variable, you'll be met with such a result:

```
> const func = 5
func = 6
✖ ▶ Uncaught TypeError: Assignment to constant variable.
   at <anonymous>:2:6 VM108:2
```

— TypeError example with constant object assignment.

While it seems easy to fix right now, imagine hundreds of such variable declarations and one of them mistakenly defined as `const` instead of `let`! [Unlike other scripting languages like PHP](#), there's minimal difference between the style of declaring constants and variables in JavaScript. Therefore it's advisable to check your declarations first of all when you face this error. You could also run into this error if you forget that the said reference is a constant and use it as a variable. This indicates either carelessness or a flaw in your app's logic. Make sure to check this when trying to fix this issue.

### 10. (unknown): Script error

A script error occurs when a third-party script sends an error to your browser. This error is followed by (unknown) because the third-party script belongs to a different domain than your app. The browser hides other details to prevent leaking sensitive information from the third-

party script.

You can not resolve this error without knowing the complete details. Here's what you can do to get more information about the error:

1. Add the `crossorigin` attribute in the script tag.
2. Set the correct `Access-Control-Allow-Origin` header on the server hosting the script.
3. [Optional] If you don't have access to the server hosting the script, you can consider using a proxy to relay your request to the server and back to the client with the correct headers.

Once you can access the details of the error, you can then set down to fix the issue, which will probably be with either the third-party library or the network.

## How to Identify and Prevent Errors in JavaScript

While the errors discussed above are the most common and frequent in JavaScript, you'll come across, relying on a few examples can never be enough. It's vital to understand how to detect and prevent any type of error in a JavaScript application while developing it. Here is how you can handle errors in JavaScript.

### Manually Throw and Catch Errors

The most fundamental way of handling errors that have been thrown either manually or by the runtime is to catch them. Like most other languages, JavaScript offers a set of keywords to handle errors. It's essential to know each of them in-depth before you set down to handle errors in your JavaScript app.

#### throw

The first and most basic keyword of the set is `throw`. As evident, the throw keyword is used to throw errors to create exceptions in the JavaScript runtime manually. We have already discussed this earlier in the piece, and here's the gist of this keyword's significance:

- You can `throw` anything, including numbers, strings, and `Error` objects.
- However, it's not advisable to throw primitive data types such as strings and numbers since they don't carry debug information about the errors.
- Example: `throw TypeError("Please provide a string")`

#### try

The `try` keyword is used to indicate that a block of code might throw an exception. Its syntax is:

```
try {  
    // error-prone code here  
}
```

It's important to note that a `catch` block must always follow the `try` block to handle errors effectively.

#### catch

The `catch` keyword is used to create a catch block. This block of code is responsible for handling the errors that the trailing `try` block catches. Here is its syntax:

```
catch (exception) {  
    // code to handle the exception here  
}
```

And this is how you implement the `try` and the `catch` blocks together:

```
try {  
    // business logic code  
} catch (exception) {  
    // error handling code  
}
```

Unlike C++ or Java, you can not append multiple `catch` blocks to a `try` block in JavaScript. This means that you can not do this:

```
try {  
    // business logic code  
} catch (exception) {  
    if (exception instanceof TypeError) {  
        // do something  
    }  
} catch (exception) {  
    if (exception instanceof RangeError) {  
        // do something  
    }  
}
```

Instead, you can use an `if...else` statement or a switch case statement inside the single catch block to handle all possible error cases. It would look like this:



```
try {  
    // business logic code  
} catch (exception) {  
    if (exception instanceof TypeError) {  
        // do something  
    } else if (exception instanceof RangeError) {  
        // do something else  
    }  
}
```

### finally

The **finally** keyword is used to define a code block that is run after an error has been handled. This block is executed after the try and the catch blocks.

Also, the finally block will be executed regardless of the result of the other two blocks. This means that even if the catch block cannot handle the error entirely or an error is thrown in the catch block, the interpreter will execute the code in the finally block before the program crashes.

To be considered valid, the try block in JavaScript needs to be followed by either a catch or a finally block. Without any of those, the interpreter will raise a SyntaxError. Therefore, make sure to follow your try blocks with at least either of them when handling errors.

## Handle Errors Globally With the onerror() Method

The **onerror()** method is available to all HTML elements for handling any errors that may occur with them. For instance, if an **img** tag cannot find the image whose URL is specified, it fires its onerror method to allow the user to handle the error.

Typically, you would provide another image URL in the onerror call for the **img** tag to fall back to. This is how you can do that via JavaScript:

```
const image = document.querySelector("img")  
  
image.onerror = (event) => {  
    console.log("Error occurred: " + event)  
}
```

However, you can use this feature to create a global error handling mechanism for your app. Here's how you can do it:

```
window.onerror = (event) => {  
  console.log("Error occurred: " + event)  
}
```

With this event handler, you can get rid of the multiple `try...catch` blocks lying around in your code and centralize your app's error handling similar to event handling. You can attach multiple error handlers to the window to maintain the Single Responsibility Principle from the SOLID design principles. The interpreter will cycle through all handlers until it reaches the appropriate one.

## Pass Errors via Callbacks

While simple and linear functions allow error handling to remain simple, callbacks can complicate the affair.

**Need a hosting solution that gives you a competitive edge? Kinsta's got you covered with incredible speed, state-of-the-art security, and auto-scaling. [Check out our plans](#)**

Consider the following piece of code:

```
const calculateCube = (number, callback) => {  
  setTimeout(() => {  
    const cube = number * number * number  
    callback(cube)  
  }, 1000)  
}  
  
const callback = result => console.log(result)  
  
calculateCube(4, callback)
```

The above function demonstrates an asynchronous condition in which a function takes some time to process operations and returns the result later with the help of a callback.

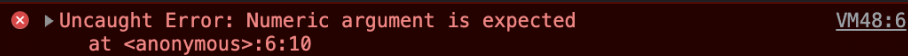
If you try to enter a string instead of 4 in the function call, you'll get `NaN` as a result.

This needs to be handled properly. Here's how:

```
const calculateCube = (number, callback) => {
```

```
    setTimeout(() => {  
      if (typeof number !== "number")  
        throw new Error("Numeric argument is expected")  
  
      const cube = number * number * number  
      callback(cube)  
    }, 1000)  
}  
  
const callback = result => console.log(result)  
  
try {  
  calculateCube(4, callback)  
} catch (e) { console.log(e) }
```

This should solve the problem ideally. However, if you try passing a string to the function call, you'll receive this:



```
✖ ▶ Uncaught Error: Numeric argument is expected  
   at <anonymous>:6:10 VM48:6
```

— Error example with the wrong argument.

Even though you have implemented a try-catch block while calling the function, it still says the error is uncaught. The error is thrown after the catch block has been executed due to the timeout delay.

This can occur quickly in network calls, where unexpected delays creep in. You need to cover such cases while developing your app.

Here's how you can handle errors properly in callbacks:

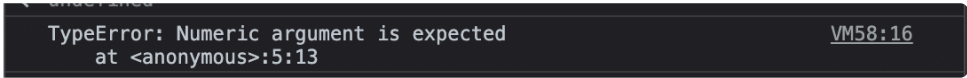
```
const calculateCube = (number, callback) => {  
  
  setTimeout(() => {  
    if (typeof number !== "number") {  
      callback(new TypeError("Numeric argument is expected"))  
      return  
    }  
  
    const cube = number * number * number  
    callback(null, cube)  
  })  
}
```

```
    }, 2000)
  }

  const callback = (error, result) => {
    if (error !== null) {
      console.log(error)
      return
    }
    console.log(result)
  }

  try {
    calculateCube('hey', callback)
  } catch (e) {
    console.log(e)
  }
}
```

Now, the output at the console will be:



```
TypeError: Numeric argument is expected
    at <anonymous>:5:13 VM58:16
```

— TypeError example with illegal argument.

This indicates that the error has been appropriately handled.

## Handle Errors in Promises

Most people tend to prefer promises for handling asynchronous activities. Promises have another advantage — a rejected promise doesn't terminate your script. However, you still need to implement a catch block to handle errors in promises. To understand this better, let's rewrite the `calculateCube()` function using Promises:

```
const delay = ms => new Promise(res => setTimeout(res, ms))

const calculateCube = async (number) => {
  if (typeof number !== "number")
    throw Error("Numeric argument is expected")
  await delay(5000)
  const cube = number * number * number
  return cube
}
```

```
}

try {
    calculateCube(4).then(r => console.log(r))
} catch (e) { console.log(e) }
```

The timeout from the previous code has been isolated into the `delay` function for understanding. If you try to enter a string instead of 4, the output that you get will be similar to this:

```
✖ ▶ Uncaught (in promise) Error: Numeric argument is expected VM445:6
    at calculateCube (<anonymous>:6:10)
    at <anonymous>:15:1
```

— TypeError example with an illegal argument in Promise.

Again, this is due to the `Promise` throwing the error after everything else has completed execution. The solution to this issue is simple. Simply add a `catch()` call to the promise chain like this:

```
calculateCube("hey")
    .then(r => console.log(r))
    .catch(e => console.log(e))
```

Now the output will be:

```
Error: Numeric argument is expected VM495:15
    at calculateCube (<anonymous>:6:10)
    at <anonymous>:15:1
```

— Handled TypeError example with illegal argument.

You can observe how easy it is to handle errors with promises. Additionally, you can chain a `finally()` block and the promise call to add code that will run after error handling has been completed.

Alternatively, you can also handle errors in promises using the traditional try-catch-finally technique. Here's how your promise call would look like in that case:

```
try {
```

```
    let result = await calculateCube("hey")
    console.log(result)
  } catch (e) {
    console.log(e)
  } finally {
    console.log('Finally executed')
  }
```

However, this works inside an asynchronous function only. Therefore the most preferred way to handle errors in promises is to chain `catch` and `finally` to the promise call.

## throw/catch vs onerror() vs Callbacks vs Promises: Which is the Best?

With four methods at your disposal, you must know how to choose the most appropriate in any given use case. Here's how you can decide for yourselves:

### throw/catch

You will be using this method most of the time. Make sure to implement conditions for all possible errors inside your catch block, and remember to include a finally block if you need to run some memory clean-up routines after the try block.

However, too many try/catch blocks can make your code difficult to maintain. If you find yourself in such a situation, you might want to handle errors via the global handler or the promise method.

When deciding between asynchronous try/catch blocks and promise's `catch()`, it's advisable to go with the async try/catch blocks since they will make your code linear and easy to debug.

### onerror()

It's best to use the `onerror()` method when you know that your app has to handle many errors, and they can be well-scattered throughout the codebase. The `onerror` method enables you to handle errors as if they were just another event handled by your application. You can define multiple error handlers and attach them to your app's window on the initial rendering.

However, you must also remember that the `onerror()` method can be unnecessarily challenging to set up in smaller projects with a lesser scope of error. If you're sure that your app will not throw too many errors, the traditional throw/catch method will work best for you.

### Callbacks and Promises

Error handling in callbacks and promises differs due to their code design and structure. However, if you choose between these two before you have written your code, it would be best to go with promises.

This is because promises have an inbuilt construct for chaining a `catch()` and a `finally()` block to handle errors easily. This method is easier and cleaner than defining additional arguments/reusing existing arguments to handle errors.

## Keep Track of Changes With Git Repositories

Many errors often arise due to manual mistakes in the codebase. While developing or debugging your code, you might end up making unnecessary changes that may cause new errors to appear in your codebase. [Automated testing](#) is a great way to keep your code in check after every change. However, it can only tell you if something's wrong. If you don't take

frequent backups of your code, you'll end up wasting time trying to fix a function or a script that was working just fine before.

This is where git plays its role. With a proper commit strategy, you can use your git history as a backup system to view your code as it evolved through the development. You can easily browse through your older commits and find out the version of the function working fine before but throwing errors after an unrelated change.

You can then restore the old code or compare the two versions to determine what went wrong. Modern [web development tools](#) like [GitHub Desktop](#) or [GitKraken](#) help you to visualize these changes side by side and figure out the mistakes quickly.

A habit that can help you make fewer errors is running [code reviews](#) whenever you make a significant change to your code. If you're working in a team, you can create a pull request and have a team member review it thoroughly. This will help you use a second pair of eyes to spot out any errors that might have slipped by you.

## Best Practices for Handling Errors in JavaScript

The above-mentioned methods are adequate to help you design a robust error handling approach for your next JavaScript application. However, it would be best to keep a few things in mind while implementing them to get the best out of your error-proofing. Here are some tips to help you.

### 1. Use Custom Errors When Handling Operational Exceptions

We introduced custom errors early in this guide to give you an idea of how to customize the error handling to your application's unique case. It's advisable to use custom errors wherever possible instead of the generic `Error` class as it provides more contextual information to the calling environment about the error.

On top of that, custom errors allow you to moderate how an error is displayed to the calling environment. This means that you can choose to hide specific details or display additional information about the error as and when you wish.

You can go so far as to format the error contents according to your needs. This gives you better control over how the error is interpreted and handled.

### 2. Do Not Swallow Any Exceptions

Even the most senior developers often make a rookie mistake — consuming exceptions levels deep down in their code.

You might come across situations where you have a piece of code that is optional to run. If it works, great; if it doesn't, you don't need to do anything about it.

In these cases, it's often tempting to put this code in a try block and attach an empty catch block to it. However, by doing this, you'll leave that piece of code open to causing any kind of error and getting away with it. This can become dangerous if you have a large codebase and many instances of such poor error management constructs.

The best way to handle exceptions is to determine a level on which all of them will be dealt and raise them until there. This level can be a controller (in an MVC architecture app) or a middleware (in a traditional server-oriented app).

This way, you'll get to know where you can find all the errors occurring in your app and choose how to resolve them, even if it means not doing anything about them.

### 3. Use a Centralized Strategy for Logs and Error Alerts

Logging an error is often an integral part of handling it. Those who fail to develop a centralized strategy for logging errors may miss out on valuable information about their app's usage.

An app's event logs can help you figure out crucial data about errors and help to debug them quickly. If you have proper alerting mechanisms set up in your app, you can know when an error occurs in your app before it reaches a large section of your user base.

It's advisable to use a pre-built logger or create one to suit your needs. You can configure this logger to handle errors based on their levels (warning, debug, info, etc.), and some loggers even go so far as to send logs to remote logging servers immediately. This way, you can watch how your application's logic performs with active users.

#### 4. Notify Users About Errors Appropriately

Another good point to keep in mind while defining your error handling strategy is to keep the user in mind.

All errors that interfere with the normal functioning of your app must present a visible alert to the user to notify them that something went wrong so the user can try to work out a solution. If you know a quick fix for the error, such as retrying an operation or logging out and logging back in, make sure to mention it in the alert to help fix the user experience in real-time.

In the case of errors that don't cause any interference with the everyday user experience, you can consider suppressing the alert and logging the error to a remote server for resolving later.

#### 5. Implement a Middleware (Node.js)

The [Node.js environment](#) supports middlewares to add functionalities to server applications. You can use this feature to create an error-handling middleware for your server.

The most significant benefit of using middleware is that all of your errors are handled centrally in one place. You can choose to enable/disable this setup for testing purposes easily.

Here's how you can create a basic middleware:

```
const logError = err => {
  console.log("ERROR: " + String(err))
}

const errorLoggerMiddleware = (err, req, res, next) => {
  logError(err)
  next(err)
}

const returnErrorMiddleware = (err, req, res, next) => {
  res.status(err.statusCode || 500)
    .send(err.message)
}

module.exports = {
  logError,
  errorLoggerMiddleware,
  returnErrorMiddleware
}
```



You can then use this middleware in your app like this:

```
const { errorLoggerMiddleware, returnErrorMiddleware } = require('./middleware')

app.use(errorLoggerMiddleware)

app.use(returnErrorMiddleware)
```

You can now define custom logic inside the middleware to handle errors appropriately. You don't need to worry about implementing individual error handling constructs throughout your codebase anymore.

## 6. Restart Your App To Handle Programmer Errors (Node.js)

When Node.js apps encounter programmer errors, they might not necessarily throw an exception and try to close the app. Such errors can include issues arising from programmer mistakes, like high CPU consumption, memory bloating, or memory leaks. The best way to handle these is to gracefully restart the app by crashing it via the Node.js cluster mode or a unique tool like PM2. This can ensure that the app doesn't crash upon user action, presenting a terrible user experience.

## 7. Catch All Uncaught Exceptions (Node.js)

You can never be sure that you have covered every possible error that can occur in your app. Therefore, it's essential to implement a fallback strategy to catch all uncaught exceptions from your app.

Here's how you can do that:

```
process.on('uncaughtException', error => {
  console.log("ERROR: " + String(error))
  // other handling mechanisms
})
```

You can also identify if the error that occurred is a standard exception or a custom operational error. Based on the result, you can exit the process and restart it to avoid unexpected behavior.

## 8. Catch All Unhandled Promise Rejections (Node.js)


Similar to how you can never cover for all possible exceptions, there's a high chance that you might miss out on handling all possible promise rejections. However, unlike exceptions, promise rejections don't throw errors.

So, an important promise that was rejected might slip by as a warning and leave your app open to the possibility of running into unexpected behavior. Therefore, it's crucial to implement

a fallback mechanism for handling promise rejection.

Here's how you can do that:

```
const promiseRejectionCallback = error => {  
  console.log("PROMISE REJECTED: " + String(error))  
}  
  
process.on('unhandledRejection', callback)
```

“ If you create an application, there are chances that you'll create bugs and other issues in it as well. 😄  
Learn how to handle them with help from this guide 

CLICK TO TWEET

## Summary

Like any other programming language, errors are quite frequent and natural in JavaScript. In some cases, you might even need to throw errors intentionally to indicate the correct response to your users. Hence, understanding their anatomy and types is very crucial.

Moreover, you need to be equipped with the right tools and techniques to identify and prevent errors from taking down your application.

In most cases, a solid strategy to handle errors with careful execution is enough for all types of JavaScript applications.

*Are there any other JavaScript errors that you still haven't been able to resolve? Any techniques for handling JS errors constructively? Let us know in the comments below!*

Save time, costs and maximize site performance with:

- Instant help from WordPress hosting experts, 24/7.
- Cloudflare Enterprise integration.
- Global audience reach with 35 data centers worldwide.
- Optimization with our built-in Application Performance Monitoring.

All of that and much more, in one plan with no long-term contracts, assisted migrations, and a 30-day-money-back-guarantee. [Check out our plans](#) or [talk to sales](#) to find the plan that's right for you.

Hand-picked related articles

BLOG

### The 40 Best JavaScript Libraries and Frameworks for 2022

Explore our hand-picked list of the best JavaScript libraries and frameworks. You'll also learn their features, benefits, and top use cases.

34 MIN READ   MARCH 15, 2021   JAVASCRIPT FRAMEWORKS  
JAVASCRIPT TUTORIALS



BLOG

### What Is the Best Programming Language to Learn in 2022?

With so many available, it can be hard to know which is the best programming language to learn right now. We break down your options here.

18 MIN READ   MARCH 5, 2021   LEARN PHP  
WEB DEVELOPMENT LANGUAGES



BLOG

### The Ultimate Guide to Fixing and Troubleshooting the Most Common WordPress Errors (70+ Issues)

The longest list of the most common WordPress errors and how to quickly fix/troubleshoot them (continuously updated).

43 MIN READ   APRIL 1, 2020   WEBSITE ERRORS  
WORDPRESS ISSUES



## Comments

Leave A Comment

### Leave a Reply

**Comment policy:** We love comments and appreciate the time that readers spend to share ideas and give feedback. However, all comments are manually moderated and those deemed to be spam or solely promotional will be deleted.

#### Comment

#### Name

#### Email

By submitting this form: You agree to the processing of the submitted personal data in accordance with Kinsta's Privacy Policy, including the transfer of data to the United States.

☐ You also agree to receive information from Kinsta related to our services, events, and promotions. You may unsubscribe at any time by following the instructions in the communications received.

Post Comment



### Subscribe to our Newsletter

Get premium content from an [award-winning](#) WordPress hosting platform

Subscribe



[Log in](#) to MyKinsta. Or, [create an account](#) for **\$20 off your first month** of Application Hosting and Database Hosting.

English

Kinsta Hosting	Resources	Company
Pricing	All Resources	About Us
WordPress Hosting	Blog	Careers
Application Hosting	Knowledge Base	Clients & Case Studies
Database Hosting	Help Center	Contact Us
Agency Hosting	Feature Updates	Kinsta Reviews
Single Site Hosting	Kinsta Academy	Partners
WooCommerce Hosting	Ebooks	Press
Enterprise Hosting	Development Tools	Why Us
Kinsta Support	Affiliate Academy	Affiliate Program
Free Migration	Agency Directory	Affiliate Dashboard
Platform		Compare Kinsta
Add-Ons		Kinsta vs WP Engine
APM Tool		Kinsta vs SiteGround
DevKinsta		Kinsta vs Flywheel
Cloudflare Integration		More Comparisons
System Status		