# Exception Handling in JavaScript

Javascript is an interpretation language. An interpreted language is a programming language where the code is translated and executed line-by-line each time it runs. The JavaScript engine reads your code and checks for syntax errors. The engine then executes the code line-by-line, translating it into machine instructions on the fly.

## Overview

### What is Exception Handling?

Exception handling is the process of identifying, catching, and managing errors in a program to prevent unexpected crashes. It ensures that the application can gracefully handle errors and continue executing without breaking functionality.

### Importance of Exception Handling in JavaScript

- Ensures errors don’t disrupt the entire program.

- Helps developers track and fix issues efficiently.
- Avoids unexpected failures and provides meaningful error messages.
- Helps maintain a smooth, predictable execution flow across different browsers.

## Types of Errors in JavaScript

- **Syntax Errors**: Occur when code is incorrectly written (e.g., missing brackets).
- **Reference Errors**: Accessing variables or functions that are not defined.
- **Type Errors**: Occurs when operations are performed on incompatible data types.
- **Range Errors**: Triggered when a number is out of an allowable range.
- **URI Errors**: Raised when using encodeURI() or decodeURI() incorrectly, such as passing a malformed URI component

## How to handle Exceptions in JavaScript?

- throw statements

- try...catch statements
- try...catch...finally statements.

Even though JavaScript isn't compiled, exception handling is still crucial for writing robust and error-resistant code. Exception handling allows you to gracefully handle unexpected errors during runtime. Without it, your code could crash, interrupting the user experience.

## What is Exception Handling?

There are several types of errors that can occur in our code, like syntax errors, reference errors, type errors, range errors, etc. In order to handle all such errors, you will need exception handling.

Exception handling helps you identify and log errors, making it easier to debug and troubleshoot issues. For example, Operations such as dividing a non-zero number by zero, calling a method with incorrect arguments, or failing to read a file could give a JavaScript exception in our code. If we have exception handling in place, this could save our application from crashing by throwing the exact error to the user.

**Read More:** Exception Handling in Selenium WebDriver

## Importance of Exception Handling in JavaScript

In general, when an application encounters an error, there is no use proceeding with it as it may cause an application to crash.

So, it is important to handle any exception that could happen in code to keep the application up and running. If an unexpected response is given and then processed by the caller, the user gets incorrect information and a bad impression of the application.

A well-written application should have a good exception-handling mechanism and approach to various kinds of errors that are likely to happen within the application during usage.

## Types of Errors in JavaScript

Below are the common types of errors that we will notice in Javascript code.

**1. Syntax error:** This error occurs when the Javascript engine encounters invalid syntax while parsing your code.

For example,

```
console.log("Hello World";
//Since the console.log is missing parentheses you will get it as syntax error
Uncaught SyntaxError: missing ) after argument list
```

**2. Reference error:** When we try to access a variable that is out of our scope or not yet defined, then this error will occur.

Take the below as an example,

```
console.log(x);
//When we try to print a variable that is never defined we get a reference error
Uncaught ReferenceError: x is not defined
```

**3. Type error**: When you perform an operation on the wrong value, this error will occur.

For example,

```
num()
//you are calling a function that is never defined and it will throw ta ype error
Uncaught TypeError: num is not a function
```

**4. URI error**: This error occurs when there is a problem in encoding or decoding the URI(Uniform Resource Identifier).

For example,

```
decodeURIComponent("%");
//You are trying to decode an invalid URI component
URIError: URI malformed
```

**5. Range error**: When the numeric value is outside of the allowed range.

For example,

```
let arr = new Array(-1);
//You are trying to create an array of negative length that is invalid
Uncaught RangeError: Invalid array length
```

**Read More:** JavaScript Unit Testing Tutorial

## What is an Error Object in JavaScript?

In JavaScript, an Error object represents an error that occurred during the execution of your code. It provides information about the error, such as its type and a descriptive message. It will have the properties below:

- **Name:** A string representing the type of error (for example, **"TypeError", "ReferenceError", "SyntaxError").**
- **Message:** A human-readable string describing the error.

- **Cause:** A property that can reference another error object, indicating the underlying cause of the current error.

Consider the below code that has the name, message, and cause of the error inside it,

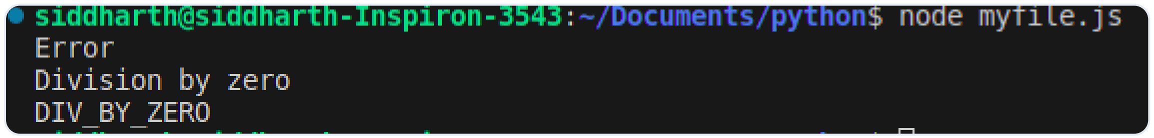
```
function divide(a, b) {
     if (b === 0) {
       throw new Error("Division by zero", {
//Adding cause property to the error function
         cause: {
           code: "DIV_BY_ZERO",
           dividend: a,
           divisor: b
         }
       });
     }
     return a / b;
   }

   try {
//Dividing the number 10 by 0
```

```
    divide(10, 0);
  } catch (error) {
    console.log(error.name)
    console.log(error.message)
    console.log(error.cause.code); // "DIV_BY_ZERO"
  }
```

Your output will look like the following,

This has the error name, error message, and error cause variable.



```
siddharth@siddharth-Inspiron-3543:~/Documents/python$ node myfile.js
Error
Division by zero
DIV_BY_ZERO
```

## How to handle Exceptions in JavaScript?

Here are some of the ways how you can handle exceptions in JavaScript.

### 1. Throw statements

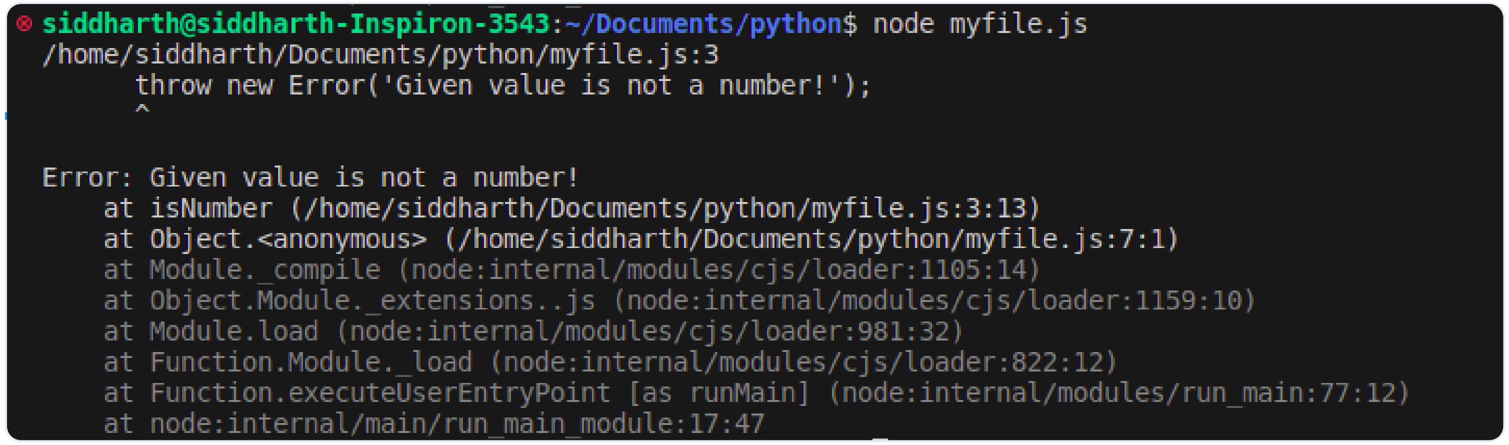
The throw statement throws a user-defined exception. It allows you to create a custom error. The exception can be a string, number, boolean, or object.

With the throw statement, the execution of the current function will stop (the statements after the throw won't be executed), and control will be passed to the first catch block in the call stack. If no catch block exists among caller functions, the program will terminate.

**For example**, you may want to throw an error if the value which we pass is not a number. You can define like below to throw an exception if the variable we pass isn't a number.

```
function isNumber(num) {
if (isNaN(num)) {
   throw new Error('Given value is not a number!');
}
  }


isNumber('A')
```



```
siddharth@siddharth-Inspiron-3543:~/Documents/python$ node myfile.js
/home/siddharth/Documents/python/myfile.js:3
      throw new Error('Given value is not a number!');
      ^

Error: Given value is not a number!
    at isNumber (/home/siddharth/Documents/python/myfile.js:3:13)
    at Object.<anonymous> (/home/siddharth/Documents/python/myfile.js:7:1)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47
```

Talk to an Expert

## 2. try…catch statements

The try…catch statements have a try block and then a catch block. The code written inside the try block will get executed first and if it throws any error then the catch block will throw the error mentioned inside it.

You can understand it better with the below **example.** Call a function that is never defined inside a try block, and inside the catch block, print the error saying it is an invalid function,

```
try {
```

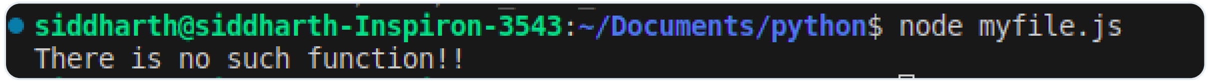
```
noSuchFunction()

}

catch{

console.log("There is no such function!!")

}
```



```
siddharth@siddharth-Inspiron-3543:~/Documents/python$ node myfile.js
There is no such function!!
```

With the above code you might have noticed that even though there is no such function present, the catch block handled the error and printed the error message.

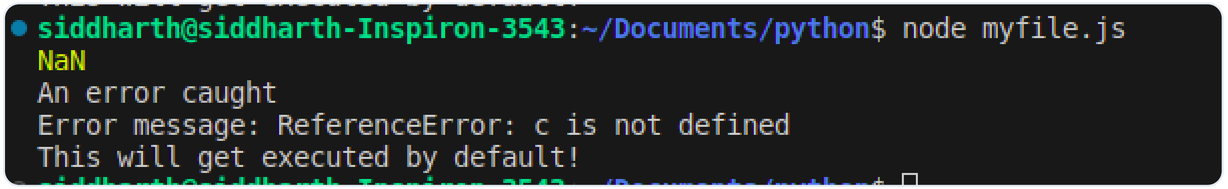
**Read More:** [Common JavaScript Issues and its Solutions]

## 3. try…catch…finally statements

You can also add a final statement to your try…catch block and make your code look better. With the final block added, the piece of code inside will get executed regardless of whether the try block throws an error or not.

**For example**, let **a,b** & **c** be the three variables. Define a as **a** number, **b** as a string, and **c** as undefined.

```
const a= 10, b = 'string';
try {
  console.log(a/b);
  console.log(c); //Since the variable is undefined, it will be passed on to catch
}
catch(error) {
console.log('An error caught');
console.log('Error message: ' + error);
}
finally {
  console.log('This will get executed by default!');
}
```



```
siddharth@siddharth-Inspiron-3543:~/Documents/python$ node myfile.js
NaN
An error caught
Error message: ReferenceError: c is not defined
This will get executed by default!
```

From the above screenshot, you might have noticed that even though the try block has any error or not, the final block will get executed and throw the specified error message.

## Best Practices for Exception Handling in JavaScript

Here are some of the best practices for using exception handling in Javascript:

- Always have only the piece of code which will throw an exception inside the try block. Avoid having irrelevant code inside the try block.
- Don't catch generic errors unless absolutely necessary. Try to catch specific exceptions (for example, network errors and validation errors) and handle them accordingly.
- Create custom error classes to represent specific error conditions. This makes debugging easier by providing clear error types.
- Use logging for exceptions to ensure that you capture sufficient details to debug.
- Avoid catching errors if you don't plan to handle or log them. Let the error propagate up the call stack unless it's critical to handle it at the current level.

- The finally block is executed after the try or catch block, regardless of whether an error was thrown or not. Use it for resource cleanup (for example, closing files, and clearing timeouts).

**Read More:** Javascript Unit Testing Best Practices to Follow

## Why choose BrowserStack to run JavaScript Tests?

BrowserStack Automate is a powerful cloud-based testing platform designed to streamline and enhance your web application testing processes. Here is why you must choose BrowserStack to run JavaScript Tests:

**Test Automation on Real Devices**

Check out BrowserStack Automate Tool to run your Test Automation Suites on a real device cloud and get comprehensive test results with artifacts.

Contact Sales