Exception and Error Handling

**Exception types**

Just about any object can be thrown in JavaScript. Nevertheless, not all thrown objects are created equal. While it is common to throw numbers or strings as errors, it is frequently more effective to use one of the exception types specifically created for this purpose:

1. [ECMAScript exceptions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#error_types)
2. [DOMException](https://developer.mozilla.org/en-US/docs/Web/API/DOMException)

**throw statement**

Use the throw statement to throw an exception. A throw statement specifies the value to be thrown:

The throw statement throws a user-defined exception. Execution of the current function will stop (the statements after 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.

JS

**throw expression;**

You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

JS

throw "Error2"; // String type

throw 42; // Number type

throw true; // Boolean type

throw {

toString() {

return "I'm an object!";

},

};

example:

function getRectArea(width, height) {

if (isNaN(width) || isNaN(height)) {

throw new Error('Parameter is not a number!');

}

}

try {

getRectArea(3, 'A');

} catch (e) {

console.error(e);

// Expected output: Error: Parameter is not a number!

}

**Examples**

Throwing a user-defined error

This example defines a function that throws a TypeError if the input is not of the expected type.

JS

Copy to Clipboard

function isNumeric(x) {

return ["number", "bigint"].includes(typeof x);

}

function sum(...values) {

if (!values.every(isNumeric)) {

throw new TypeError("Can only add numbers");

}

return values.reduce((a, b) => a + b);

}

console.log(sum(1, 2, 3)); // 6

try {

sum("1", "2");

} catch (e) {

console.error(e); // TypeError: Can only add numbers

}

**Throwing an existing object**

This example calls a callback-based async function, and throws an error if the callback receives an error.

JS

readFile("foo.txt", (err, data) => {

if (err) {

throw err;

}

console.log(data);

});

Errors thrown this way are not catchable by the caller and will cause the program to crash unless (a) the readFile function itself catches the error, or (b) the program is running in a context that catches top-level errors. You can handle errors more naturally by using the Promise() constructor.

JS

function readFilePromise(path) {

return new Promise((resolve, reject) => {

readFile(path, (err, data) => {

if (err) {

reject(err);

}

resolve(data);

});

});

}

try {

const data = await readFilePromise("foo.txt");

console.log(data);

} catch (err) {

console.error(err);

}

**Function call or new construction**

When Error is used like a function, that is without new, it will return an Error object. Therefore, a mere call to Error will produce the same output that constructing an Error object via the new keyword would.

JS

const x = Error("I was created using a function call!");

// above has the same functionality as following

const y = new Error('I was constructed via the "new" keyword!');

example: IMP

<html>

<body>

<h2>JavaScript Error Handling</h2>

<p>How to use <b>catch</b> to display an error.</p>

<p id="demo"></p>

<script>

function error(message){

this.message = message;

this.name = "ReferenceError";

}

try {

//throw new error("Abbas"); // function call

// throw Error("error"); // call without new keyword

throw new Error("error2") // call with new keyword the result will same

altwert("Welcome guest!");

}

catch(err) {

alert(err.name);

alert(err.message);

}

</script>

</body>

</html>

Error objects are thrown when runtime errors occur. The Error object can also be used as a base object for user-defined exceptions. See below for standard built-in error types.

Error is a serializable object, so it can be cloned with structuredClone() or copied between Workers using postMessage().

**try, catch and finally block**

**try...catch statement**

The try...catch statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the try...catch statement catches it.

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.

In other words, you want the try block to succeed—but if it does not, you want control to pass to the catch block. If any statement within the try block (or in a function called from within the try block) throws an exception, control immediately shifts to the catch block. If no exception is thrown in the try block, the catch block is skipped. The finally block executes after the try and catch blocks execute but before the statements following the try...catch statement.

The following example uses a try...catch statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1 – 12), an exception is thrown with the value 'InvalidMonthNo' and the statements in the catch block set the monthName variable to 'unknown'.

JS

function getMonthName(mo) {

mo--; // Adjust month number for array index (so that 0 = Jan, 11 = Dec)

const months = [

"Jan", "Feb", "Mar", "Apr", "May", "Jun",

"Jul", "Aug", "Sep", "Oct", "Nov", "Dec",

];

if (months[mo]) {

return months[mo]

} else {

throw new Error("InvalidMonthNo"); // throw keyword is used here

}

}

try {

// statements to try

monthName = getMonthName(13); // function could throw exception

console.log(monthName);

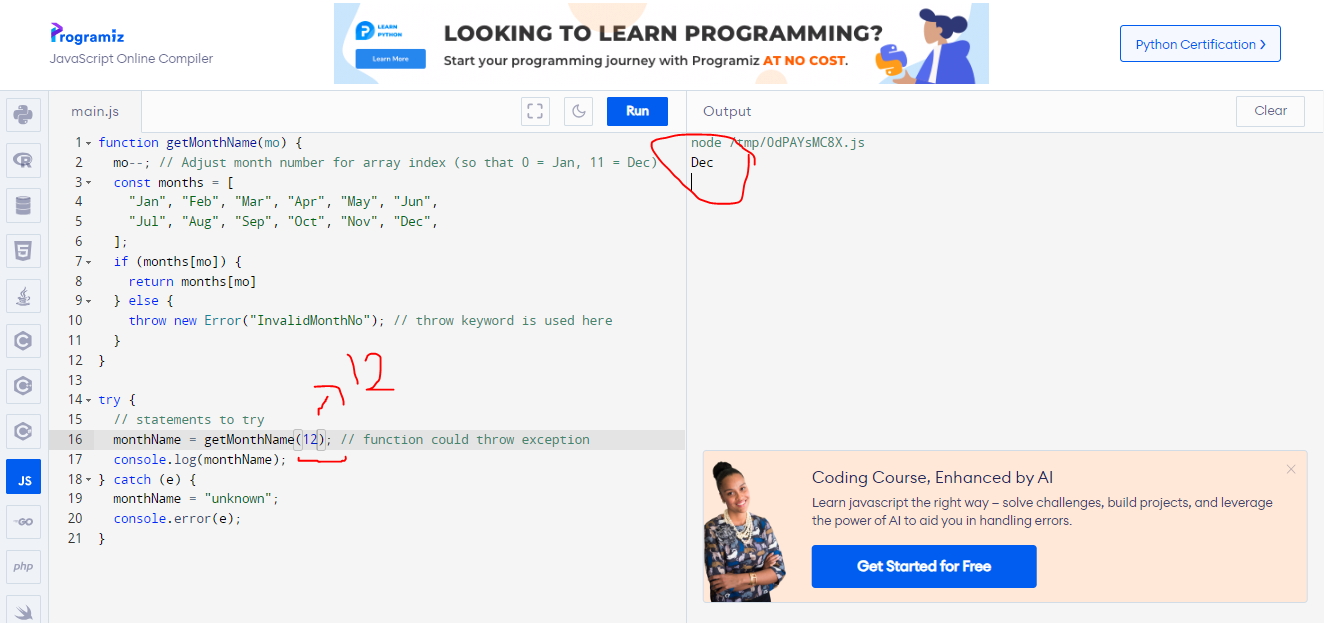
} catch (e) {

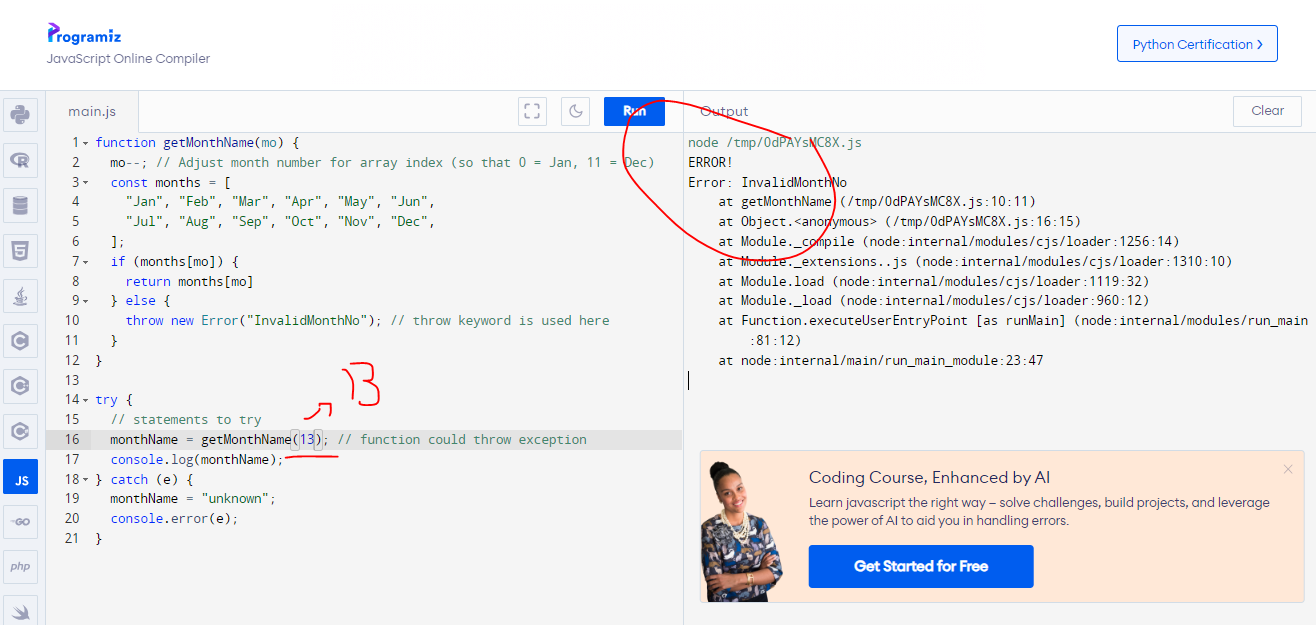
monthName = "unknown";

console.error(e);

}

Output:





**The catch block**

You can use a catch block to handle all exceptions that may be generated in the try block.

JS

Copy to Clipboard

catch (catchID) {

statements

}

The catch block specifies an identifier (catchID in the preceding syntax) that holds the value specified by the throw statement. You can use this identifier to get information about the exception that was thrown.

JavaScript creates this identifier when the catch block is entered. The identifier lasts only for the duration of the catch block. Once the catch block finishes executing, the identifier no longer exists.

For example, the following code throws an exception. When the exception occurs, control transfers to the catch block.

JS

try {

throw "myException"; // generates an exception

} catch (err) {

// statements to handle any exceptions

logMyErrors(err); // pass exception object to error handler

}

**The finally block**

The finally block contains statements to be executed after the try and catch blocks execute. Additionally, the finally block executes before the code that follows the try…catch…finally statement.

It is also important to note that the finally block will execute whether or not an exception is thrown. If an exception is thrown, however, the statements in the finally block execute even if no catch block handles the exception that was thrown.

You can use the finally block to make your script fail gracefully when an exception occurs. For example, you may need to release a resource that your script has tied up.

The following example opens a file and then executes statements that use the file. (Server-side JavaScript allows you to access files.) If an exception is thrown while the file is open, the finally block closes the file before the script fails. Using finally here ensures that the file is never left open, even if an error occurs.

JS

openMyFile();

try {

writeMyFile(theData); // This may throw an error

} catch (e) {

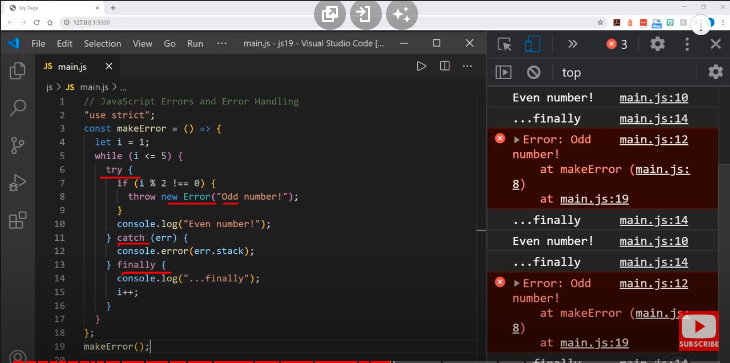
handleError(e); // If an error occurred, handle it

} finally {

closeMyFile(); // Always close the resource

}

Example of generic Error Constructor utilizing try, catch, and finally block



**Utilizing Error objects**

Depending on the type of error, you may be able to use the name and message properties to get a more refined message.

The name property provides the general class of Error (such as DOMException or Error), while message generally provides a more succinct message than one would get by converting the error object to a string.

If you are throwing your own exceptions, in order to take advantage of these properties (such as if your catch block doesn't discriminate between your own exceptions and system ones), you can use the Error constructor.

For example:

JS

function doSomethingErrorProne() {

if (ourCodeMakesAMistake()) {

throw new Error("The message");

} else {

doSomethingToGetAJavaScriptError();

}

}

try {

doSomethingErrorProne();

} catch (e) {

// Now, we actually use `console.error()`

console.error(e.name); // 'Error'

console.error(e.message); // 'The message', or a JavaScript error message

}