1. **Debugging in the browser**

Before writing more complex code, let’s talk about debugging.

[Debugging](https://en.wikipedia.org/wiki/Debugging) is the process of finding and fixing errors within a script. All modern browsers and most other environments support debugging tools – a special UI in developer tools that makes debugging much easier. It also allows to trace the code step by step to see what exactly is going on.

On windows we can open the developer tools by pressing the ***F12*** button or right clicking anywhere on the screen and then select ***inspect*.**

**THE SOURCE PANEL**

To find the debugger tools, click on the “**Sources**” panel. It is composed of three main parts:

The ***file Navigator*** pane, which can be opened and closed by clicking the emboxed arrow in the high upper left corner. In the file navigator, html, css and JavaScript files can be found. Images too, that are attached to the page. Clicking in a file (for instance, JavaScript file for example), will open the code contained in the file in the second part of the source panel, the called code editor pane.

The ***code editor*** pane, which allows us to see what the written code for that page is.

The last pane is the JavaScript Debugging pane, which is used for debugging (find errors).

**BREAKPOINTS**

A breakpoint is a point of code where the debugger will automatically pause the JavaScript execution.

While the code is paused, we can **examine** current variables, **execute** commands in the console etc. In other words, we can **debug** it.

We can also pause the code by using the code, *debugger,* in the js code file, like this:

function hello(name) {

let phrase = `Hello, ${name}!`;

debugger; // <-- the debugger stops here

say(phrase);

}

Such command works only when development tools are open.

**TRACE EXECUTION**

**Play&Pause button– “Resume”: continue the execution, hotkey F8.**

Resumes the execution. If there are no additional breakpoints, then the execution continues, and the debugger loses control.

When the resume button is pressed the “Call Stack” at right panel get increased in calls (more call appear).

**– “Step”: run the next command, hotkey F9.**

Run the next statement.

Clicking this again and again will step through all script statements one by one.

**– “Step over”: run the next command, but *don’t go into a function*, hotkey F10.**

Similar to the previous “Step” command, but behaves differently if the next statement is a function call (not a built-in, like alert, but a function of our own).

The difference between the step and the step over is that, when the next line is a function, ***step over***, will execute the function and stop at next line after the function while ***step*** will go into the function and stop ate very first line within the function.

That’s good if we’re not interested to see what happens inside the function call.

**– “Step into”, hotkey F11.**

That’s similar to “Step”, but behaves differently in case of asynchronous function calls.

“Step” command ignores async actions, such as setTimeout (scheduled function call), that execute later. The “Step into” goes into their code, waiting for them if necessary.

**– “Step out”: continue the execution till the end of the current function, hotkey Shift+F11.**

Continue the execution and stop it at the very last line of the current function. That’s handy when we accidentally entered a nested call using ***step***, but it does not interest us, and we want to continue to its end as soon as possible.

**– enable/disable all breakpoints.**

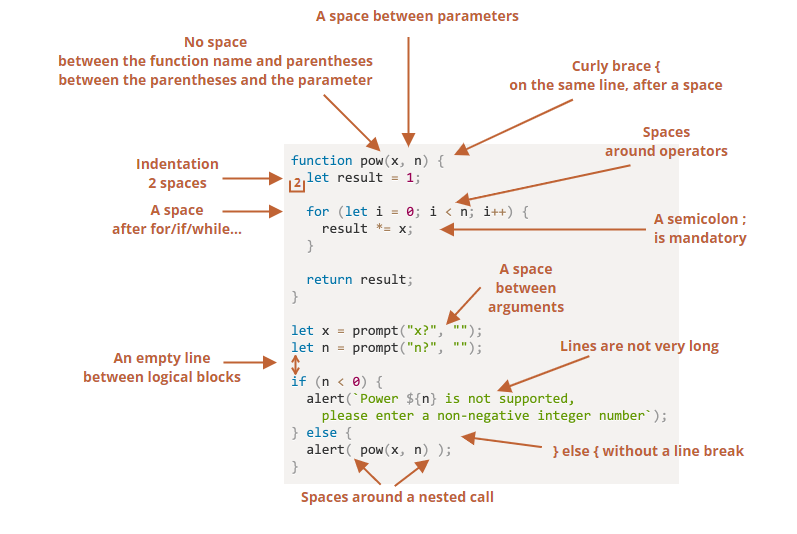
That button does not move the execution. Just a mass on/off for breakpoints.

**– enable/disable automatic pause in case of an error.**

When enabled, if the developer tools is open, an error during the script execution automatically pauses it. Then we can analyze variables in the debugger to see what went wrong. So, if our script dies with an error, we can open debugger, enable this option, and reload the page to see where it dies and what’s the context at that moment.

# Coding Style

Take a complex task and code it in a correct and human readable way.



Source: [javascript.info](https://javascript.info/coding-style#:~:text=Here%20is%20a%20cheat%20sheet%20with%20some%20suggested%20rules%20(see%20below%20for%20more%20details)%3A)

**Function placement**

When we have a helper function, it better to write the code that uses that function first and only then, write the help function, that way we can easily understand what the code does without even needing to read through the helper function.

Automated Linters

Linters are tools that can automatically check the style of your code and make improving suggestions.

here are some well-known linting tools:

* [JSLint](https://www.jslint.com/) – one of the first linters.
* [JSHint](https://jshint.com/) – more settings than JSLint.
* [ESLint](https://eslint.org/) – probably the newest one.

For instance, for ESLint you should do the following:

1. Install [Node.js](https://nodejs.org/).
2. Install ESLint with the command npm install -g eslint (npm is a JavaScript package installer).
3. Create a config file named .eslintrc in the root of your JavaScript project (in the folder that contains all your files).
4. Install/enable the plugin for your editor that integrates with ESLint. The majority of editors have one.

# Error handling, "try...catch"

When an error occurs, the script dies, printing the error in the console. But there is a syntax construct *try…catch* that allows us to “catch” errors so the script can, instead of dying do something more reasonable.

## [The “try…catch” syntax](https://javascript.info/try-catch" \l "the-try-catch-syntax)

The try...catch construct has two main blocks: try, and then catch:

try {

// code...

} catch (err) {

// error handling

}

It works like this:

1. First, the code in **try {...} is executed.**
2. If there were **no errors, then catch (err) is ignored**: the execution reaches the end of try and goes on, skipping catch.
3. If an **error occurs**, then the **try execution is stopped**, and control flows to the beginning of **catch (err)**. The err variable (we can use any name for it) will contain an error object with details about what happened.

So, an error inside the try {...} block does not kill the script – we have a chance to handle it in catch.

NOTE:

* **try...catch only works for runtime errors** or sometimes called, “exceptions”. It means that the JavaScript code written within *try* block must be valid JavaScript. It won’t work if the code is syntactically wrong.

The JavaScript engine first reads the code, and then runs it. The errors that occur on the reading phase are called “parse-time” errors and are unrecoverable (from inside that code). That’s because the engine can’t understand the code.

* **try...catch works synchronously**

If an exception happens in “scheduled” code, like in setTimeout, then try...catch won’t catch it:

try {

setTimeout(function() {

noSuchVariable; // script will die here

}, 1000);

} catch (err) {

alert( "won't work" );

}

That’s because the function itself is executed later, when the engine has already left the try...catch construct.

To catch an exception inside a scheduled function, try...catch must be inside that function:

setTimeout(function() {

try {

noSuchVariable; // try...catch handles the error!

} catch {

alert( "error is caught here!" );

}

}, 1000);

## [Error object](https://javascript.info/try-catch" \l "error-object)

When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to catch:

For all built-in errors, the error object has two main properties:

**name**

Error name. For instance, for an undefined variable that’s "ReferenceError".

**message**

Textual message about error details.

There are other non-standard properties available in most environments. One of most widely used and supported is:

**stack**

Current call stack: a string with information about the sequence of nested calls that led to the error. Used for debugging purposes.

## [Optional “catch” binding](https://javascript.info/try-catch" \l "optional-catch-binding)

**A recent addition**

This is a recent addition to the language. Old browsers may need [polyfills](https://javascript.info/polyfills).

If we don’t need error details, catch may omit it:

try {

  // ...

} catch { // <-- without (err)

  // ...

}

## [Throwing our own errors](https://javascript.info/try-catch" \l "throwing-our-own-errors)

Sometimes, things go wrong and are not noticed immediately. We can encode exceptions handle to catch such exceptions. We do that using the “Throw” operator.

### [“Throw” operator](https://javascript.info/try-catch" \l "throw-operator)

The throw operator generates an error.

The syntax is:

throw <error object>

Technically, we can use anything as an error object. That may be even a primitive, like a number or a string, but it’s better to use objects, preferably with name and message properties (to stay somewhat compatible with built-in errors).

JavaScript has many built-in constructors for standard errors: Error, SyntaxError, ReferenceError, TypeError and others. We can use them to create error objects as well.

Their syntax is:

let error = new Error(message);

// or

let error = new SyntaxError(message);

let error = new ReferenceError(message);

// ...

For built-in errors (not for any objects, just for errors), the name property is exactly the name of the constructor. And message is taken from the argument.

For instance:

let error = new SyntaxError("A SyntaxError occured");

alert(error.name); // SyntaxError

alert(error.message); // A syntax error occured

Now let’s see how we can use the throw operator:

let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors

  if (!user.name) {

    throw new SyntaxError("Incomplete data: no name"); // (\*)

  }

  alert( user.name );

} catch (err) {

  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name

}

In the line (\*), the throw operator generates a SyntaxError with the given message, the same way as JavaScript would generate it itself. The execution of try immediately stops and the control flow jumps into catch.

Now catch became a single place for all error handling: both for JSON.parse and other cases.

## [Rethrowing](https://javascript.info/try-catch" \l "rethrowing)

In the example above we use try...catch to handle incorrect data. But is it possible that another unexpected error occurs within the try {...} block? Like a programming error (variable is not defined) or something else, not just this “incorrect data” thing.

catch gets *all* errors from try. Here it gets an unexpected error, but still shows the same "JSON Error" message. That’s wrong and also makes the code more difficult to debug.

To avoid such problems, we can employ the “rethrowing” technique. The rule is simple:

**Catch should only process errors that it knows and “rethrow” all others.**

The “rethrowing” technique can be explained in more detail as:

1. Catch gets all errors.
2. In the catch (err) {...} block we analyze the error object err.
3. If we don’t know how to handle it, we do throw err.

Usually, we can check the error type using the instanceof operator:

try {

  user = { /\*...\*/ };

} catch (err) {

  if (err instanceof ReferenceError) {

    alert('ReferenceError'); // "ReferenceError" for accessing an undefined variable

  }

}

In the code below, we use rethrowing so that catch only handles SyntaxError:

let json = '{ "age": 30 }'; // incomplete data

try {

// do something in here

} catch (err) {

  if (err instanceof SyntaxError) {

    alert( "JSON Error: " + err.message );

  } else {

    throw err; // rethrow (\*)

  }

}

The error throwing on line (\*) from inside catch block “falls out” of try...catch and can be either caught by an outer try...catch construct (if it exists), or it kills the script.

So the catch block actually handles only errors that it knows how to deal with and “skips” all others.

The example below demonstrates how such errors can be caught by one more level of try...catch:

function readData() {

  let json = '{ "age": 30 }';

  try {

    // ...

    blabla(); // error!

  } catch (err) {

    // ...

    if (!(err instanceof SyntaxError)) {

      throw err; // rethrow (don't know how to deal with it)

    }

  }

}

try {

  readData();

} catch (err) {

  alert( "External catch got: " + err ); // caught it!

}

Here readData only knows how to handle SyntaxError, while the outer try...catch knows how to handle everything.

## [try…catch…finally](https://javascript.info/try-catch" \l "try-catch-finally)

Wait, that’s not all.

The try...catch construct may have one more code clause: finally.

If it exists, it runs in all cases:

* after try, if there were no errors,
* after catch, if there were errors.

The extended syntax looks like this:

try {

... try to execute the code ...

} catch (err) {

... handle errors ...

} finally {

... execute always ...

}