Debugging in the browser

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.

The “sources” panel

Your Chrome version may look a little bit different, but it still should be obvious what’s there.
Open the page in Chrome.
Turn on developer tools with `F12` (Mac: `Cmd+Opt+I`).
Select the `Sources` panel.

click on the toggler button and select `the file`.
The Sources panel has 3 parts:
The **File Navigator pane** lists HTML, JavaScript, CSS and other files, including images that are attached to the page. Chrome extensions may appear here too.
The **Code Editor pane** shows the source code.
The **JavaScript Debugging pane** is for debugging.

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 always find a list of breakpoints in the right panel. That’s useful when we have many breakpoints in various files. It allows us to :
Quickly jump to the breakpoint in the code (by clicking on it in the right panel).
Temporarily disable the breakpoint by unchecking it.
Remove the breakpoint by right-clicking and selecting Remove.
…And so on.

Hint : Conditional breakpoints

Right click on the line number allows to create a *conditional* breakpoint. It only triggers when the given expression, that you should provide when you create it, is truthy.
That’s handy when we need to stop only for a certain variable value or for certain function parameters.

The command debugger

We can also pause the code by using the `debugger` command in it, like this:

Example

```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger;  // <-- the debugger stops here
  say(phrase);
}
```

Hint: Such command works only when the development tools are open, otherwise the browser ignores it.

Pause and look around

The easiest way to activate the debugger (after we've set the breakpoints) is to reload the page.
These stools allow us to examine the current code state:
`Watch` – shows current values for any expressions.
You can click the plus + and input an expression. The debugger will show its value, automatically recalculating it in the process of execution.
`Call Stack` – shows the nested calls chain.
At the current moment the debugger is inside `hello()` call, called by a script in `index.html` (no function there, so it's called "anonymous").
If you click on a stack item (e.g., "anonymous"), the debugger jumps to the corresponding code, and all its variables can be examined as well.
`Scope` – current variables.
`Local` show's local function variables. You can also see their values highlighted right over the source.
`Global` has global variables (out of any functions).
There's also `this` keyword there that we didn't study yet, but we'll do that soon.

Tracing the execution

There are buttons for it at the top of the right panel. Let's engage them.
– **"Resume":** continue the execution, hotkey `F8`.
Resumes the execution. If there are no additional breakpoints, then the execution just continues and the debugger loses control.
Here's what we can see after a click on it:
The execution has resumed, reached another breakpoint inside `say()` and paused there. Take a look at the "Call Stack" at the right. It has increased by one more call. We're inside `say()` now.
– **"Step":** run the next command, hotkey `F9`.
Run the next statement. If we click it now, `alert` will be shown.
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).
If we compare them, the **"Step"** command goes into a nested function call and pauses the execution at its first line, while **"Step over"** executes the nested function call invisibly to us, skipping the function internals.
The execution is then paused immediately after that function call.
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. If you're only starting to learn JavaScript, then you can ignore the difference, as we don't have asynchronous calls yet.
For the future, just note that **"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 , 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.

Continue to here

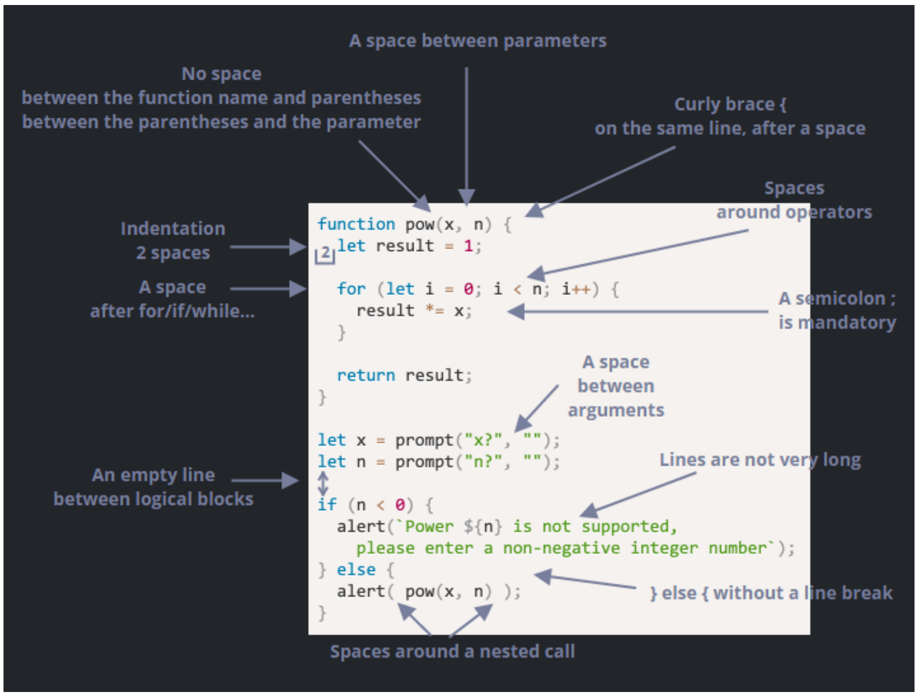
Right click on a line of code opens the context menu with a great option called **"Continue to here".** That's handy when we want to move multiple steps forward to the line, but we're too lazy to set a breakpoint.

Coding Style

Our code must be as clean and easy to read as possible.
That is actually the art of programming – to take a complex task and code it in a way that is both correct and human-readable. A good code style greatly assists in that.

Syntax

Here is a cheat sheet with some suggested rules (see below for more details):



```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

let x = prompt("x?", "");
let n = prompt("n?", "");

if (n < 0) {
  alert(`Power ${n} is not supported,
    please enter a non-negative integer number`);
} else {
  alert( pow(x, n) );
}
```

Curly Braces

In most JavaScript projects curly braces are written in “Egyptian” style with the opening brace on the same line as the corresponding keyword – not on a new line. There should also be a space before the opening bracket, like this:

```
if (n < 0) {
  alert(`Power ${n} is not supported`);
}
```

line-length

No one likes to read a long horizontal line of code. It’s best practice to split them.

For example:

```
// backtick quotes ` allow to split the string into multiple lines
let str = `
  ECMA International's TC39 is a group of JavaScript developers,
  implementers, academics, and more, collaborating with the community
  to maintain and evolve the definition of JavaScript.
`;
```

And, for if statements:

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

Hint: The maximum line length should be agreed upon at the team-level. It’s usually 80 or 120 characters.

Indents

There are two types of indents:

Horizontal indents: 2 or 4 spaces.

A horizontal indentation is made using either 2 or 4 spaces or the horizontal tab symbol (key Tab). Which one to choose is an old holy war. Spaces are more common nowadays.

One advantage of spaces over tabs is that spaces allow more flexible configurations of indents than the tab symbol.

For instance, we can align the parameters with the opening bracket, like this:

```
show(parameters,
     aligned, // 5 spaces padding at the left
     one,
     after,
     another
  ) {
  // ...
}
```

Vertical indents: empty lines for splitting code into logical blocks.

Even a single function can often be divided into logical blocks. In the example below, the initialization of variables, the main loop and returning the result are split vertically:

```
function pow(x, n) {
  let result = 1;
  //            <--
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //            <--
  return result;
}
```

Hint: Insert an extra newline where it helps to make the code more readable. There should not be more than nine lines of code without a vertical indentation.

Semicolons

A **semicolon** should be present after each statement, even if it could possibly be skipped.
There are languages where a semicolon is truly optional and it is rarely used. In JavaScript, though, there are cases where a line break is not interpreted as a semicolon, leaving the code vulnerable to errors.

Nesting levels

Try to avoid nesting code too many levels deep.
For example, in the loop, it's sometimes a good idea to use the continue directive to avoid extra nesting.
For example, instead of adding a nested if conditional like this:

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- one more nesting level
  }
}
```

We can write:

```
for (let i = 0; i < 10; i++) {
  if (!cond) continue;
  ...  // <- no extra nesting level
}
```

Function placement

If you are writing several **"helper"** functions and the code that uses them, there are three ways to organize the functions.

Declare the functions above the code that uses them:

```
// function declarations
function createElement() {
  ...
}
```

```
function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}

// the code which uses them
let elem = createElement();
setHandler(elem);
walkAround();
```

Code first, then functions

```
// the code which uses the functions
let elem = createElement();
setHandler(elem);
walkAround();

// --- helper functions ---
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}
```

Mixed: a function is declared where it's first used.

Hint: Most of time, the second variant is preferred.
That's because when reading code, we first want to know what it does. If the code goes first, then it becomes clear from the start. Then, maybe we won't need to read the functions at all, especially if their names are descriptive of what they actually do.

Style guides

A style guide contains general rules about "how to write" code, e.g. which quotes to use, how many spaces to indent, the maximal line length, etc. A lot of minor things.
When all members of a team use the same style guide, the code looks uniform, regardless of which team member wrote it.
Of course, a team can always write their own style guide, but usually there's no need to. There are many existing guides to choose from.

Some popular choices:
Google JavaScript Style Guide
Airbnb JavaScript Style Guide
Idiomatic.JS

StandardJS
(plus many more)
If you're a novice developer, start with the cheat sheet at the beginning of this chapter. Then you can browse other style guides to pick up more ideas and decide which one you like best.

Automated Linters

Linters are tools that can automatically check the style of your code and make improving suggestions. The great thing about them is that style-checking can also find some bugs, like typos in variable or function names. Because of this feature, using a linter is recommended even if you don't want to stick to one particular "code style".

Here are some well-known linting tools:
JSLint – one of the first linters.
JSHint – more settings than **JSLint**.
ESLint – probably the newest one.

All of them can do the job. The author uses ESLint.
Most linters are integrated with many popular editors: just enable the plugin in the editor and configure the style.

Here's an example of an **.eslintrc** file:

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "no-console": 0,
    "indent": 2
  }
}
```

Here the directive "extends" denotes that the configuration is based on the **"eslint:recommended"** set of settings. After that, we specify our own.

Error handling, "try...catch"

Usually, a script "dies" (immediately stops) in case of an error, printing it to console.
But there's a syntax construct **try...catch** that allows us to **"catch"** errors so the script can, instead of dying, do something more reasonable.

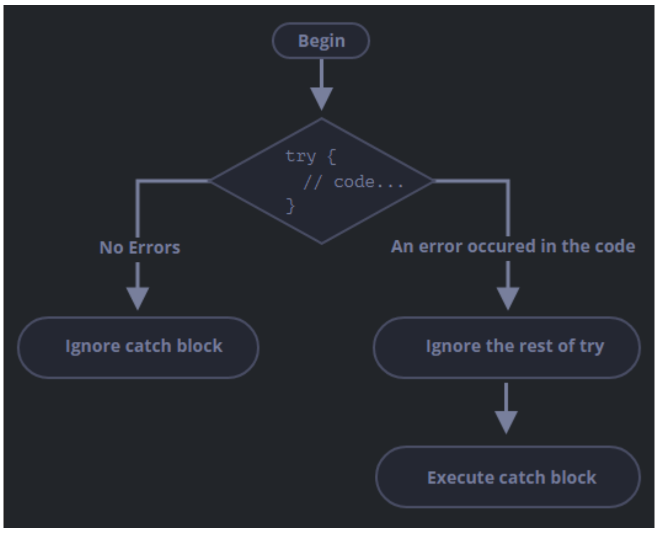
The try catch syntax

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

Example

```
try {

  // code...

} catch (err) {

  // error handling

}
```

It works like this:
First, the code in **try {...}** is executed.
If there were no errors, then **catch (err)** is ignored: the execution reaches the end of **try** and goes on, skipping **catch**.
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.



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

HINT : try...catch only works for runtime errors

For **try...catch** to work, the code must be runnable. In other words, it should be valid JavaScript.

It won't work if the code is syntactically wrong, for instance it has unmatched curly braces:

```
try {
  {{{{{{{{{{{{
} catch (err) {
  alert("The engine can't understand this code, it's invalid");
}
```

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.
So, **try...catch** can only handle errors that occur in valid code. Such errors are called “runtime errors” or, sometimes, “exceptions”.

HINT : 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

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

```
try {
  // ...
} catch (err) { // <-- the "error object", could use another word instead of err
  // ...
}
```

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.

For instance :

```
try {
  lalala ; // error, variable is not defined!
} catch (err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined at (...call stack)

  // Can also show an error as a whole
  // The error is converted to string as "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
```

Optional catch binding

A recent addition
This is a recent addition to the language. Old browsers may need polyfills.

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

```
try {
  // ...
} catch { // <-- without (err)
  // ...
}
```

Usin -try catch

Let's explore a real-life use case of **try...catch**.

As we already know, JavaScript supports the JSON.parse(str) method to read JSON-encoded values. Usually it's used to decode data received over the network, from the server or another source.
We receive it and call **JSON.parse** like this:

```
let json = '{"name":"John", "age": 30}'; // data from the server

let user = JSON.parse(json); // convert the text representation to JS object

// now user is an object with properties from the string
alert( user.name ); // John
alert( user.age );  // 30
```

Throwing our own errors

What if `json` is syntactically correct, but doesn't have a required `name` property?
Like this:

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

try {

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

} catch (err) {
  alert( "doesn't execute" );
}
```

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);
// ...
```

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.

For example:

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

try {
  user = JSON.parse(json); // <-- forgot to put "let" before user

  // ...
} catch (err) {
  alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined
  // (no JSON Error actually)
}
```

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 ...
}
```

Global catch

Environment-specific
The information from this section is not a part of the core JavaScript.

Let's imagine we've got a fatal error outside of **try...catch**, and the script died. Like a programming error or some other terrible thing.

Is there a way to react on such occurrences? We may want to log the error, show something to the user (normally they don't see error messages), etc.

The syntax:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

message
Error message.

url
URL of the script where error happened.

line, col
Line and column numbers where error happened.

error
Error object.

For instance :

```
<script>
  window.onerror = function(message, url, line, col, error) {
    alert(`${message}\n At ${line} :${col} of ${url}`) ;
  } ;

  function readData() {
    badFunc() ; // Whoops, something went wrong !
  }

  readData() ;
</script>
```

The role of the global handler **window.onerror** is usually not to recover the script execution – that's probably impossible in case of programming errors, but to send the error message to developers.

They work like this:

- We register at the service and get a piece of JS (or a script URL) from them to insert on pages.
- That JS script sets a custom window.onerror function.
- When an error occurs, it sends a network request about it to the service.
- We can log in to the service web interface and see errors.