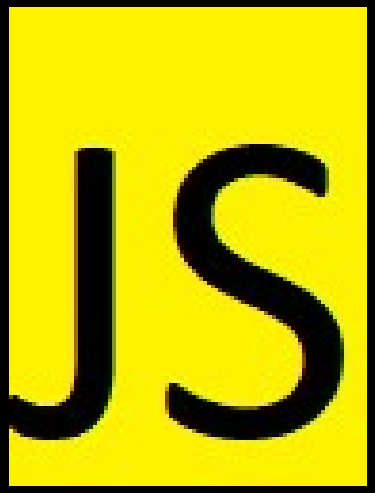


Javascript for Beginners

ESA KETONEN

PROJECT WORK

Foreword by the Author

Background:

I consulted a counselor at an unemployment office what to study to enhance my capabilities to require a work position. I commended that at least I would not study programming, but here I am.

Recommendations:

I recommend to study https://www.javascripttutorial.net and watch Frontend Masters (has a monthly fee).

Instructions:

Go through this presentation like a Javascript Engine. First parse it through and then execute it line by line. Video tips! are optional to watch.

Note!

Feel free to make me a job offer. I have my contact info at Linkedin.

The Author:

Esa Ketonen took apart to an unemployment course by Opiframe (https://opiframe.com/) called Internet of Things in Jyväskylä, Finland for seven months (14.10.2019 – 14.5.2020).

Linkedin:

https://www.linkedin.com/in/esa-ketonen-3023b7164/

Javascript for Beginners Introduction

Technical information:

This presentation was done with Microsoft Powerpoint. The structure of this presentation is mainly adapted from W3Schools Online Web Tutorials.

Code examples:

Code examples are done with Microsoft Visual Studio code editor (please read also the text with the code) . Other options were among others Atom, Notepad++, Brackets, Codepen, Sublime text and Vim.

Browser:

Google Crome web browser was used for console logging. Other options for web browsers were Mozilla Firefox, Microsoft Edge, Microsoft Internet Explorer, Opera, Apple Safari and so on.

Main sources for text and code examples were:

W3Schools Online Web Tutorials (https://www.w3schools.com/js/default.asp)

Mozilla Developer Networ (MDN) web docs which is the official source for Javascript. (https://developer.mozilla.org/en-US/docs/Web/JavaScript)

Doctor Axel Rauschmayer's Speaking Javascript online books: (https://exploringjs.com/)

https://www.javascripttutorial.net and Frontend Masters public course notes

Sources: https://www.elegantthemes.com/blog/resources/best-code-editors, https://en.wikipedia.org/wiki/List_of_web_browsers

The history of Javascript (ECMAScript)

JavaScript (JS) was invented in 1995. Javascript became an ECMA standard in 1997. ECMAScript (ES) is the official name of the language.

Version ES6 was released in 2015 and was renamed to ECMAScript 2015. Since that ECMAScript is named by year. ECMAScript 2020 is the newest version.

- HTML (Hypertext Markup Language) defines a webpage's structure and content.
- CSS (Cascading Style Sheets) sets the formatting and webpage's appearance.
- JavaScript adds interactivity to a webpage.

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview, https://en.wikipedia.org/wiki/ECMAScript and https://www.w3schools.com/js/default.asp

https://github.com/BrendanEich

Brendan Eich created JavaScript, co-founded Mozilla and Firefox.

JavaScript was developed for Netscape. After Netscape the Mozilla foundation continued to develop JavaScript for the Firefox browser.

JavaScript is commonly used for client-side scripting on the World Wide Web, and it is increasingly being used for writing server applications and services using Node.js.

The history of Javascript (ECMAScript) 2

The ECMAScript specification is a standardized specification of a scripting language. Initially it was named Mocha, later LiveScript, and finally JavaScript.

A new name became necessary because there is a trademark on JavaScript (held originally by Sun, now by Oracle). At the moment, Mozilla is one of the few companies allowed to officially use the name JavaScript because it received a license long ago.

For common usage, these rules apply:

JavaScript means the programming language.

ECMAScript is the name used by the language specification.

JavaScript's creator, Brendan Eich, had no choice but to create the Mocha very quickly (in just ten days or other, worse technologies would have been adopted by Netscape).

He borrowed from several programming languages:

Java (syntax, primitive values versus objects), Scheme and AWK (first-class functions), Self (prototypal inheritance), and Perl and Python (strings, arrays, and regular expressions).

Since that Javascript has evolved for 25 years.

Sources: https://en.wikipedia.org/wiki/ECMAScript and http://speakingjs.com/es5/ch01.html

DOM and HTML DOM

When a web page is loaded, the browser creates a Document Object Model (DOM) of the page.

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents.

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

Source: https://www.w3schools.com/js/js_htmldom.asp

With the HTML DOM, JavaScript can access and change all the elements of an HTML document.

The HTML DOM is a standard object model and programming interface for HTML. It defines:

- The HTML elements as objects
- The properties of all HTML elements
- The methods to access all HTML elements
- The events for all HTML elements

The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

HTML DOM Navigation

With the HTML DOM, you can navigate the node tree using node relationships.

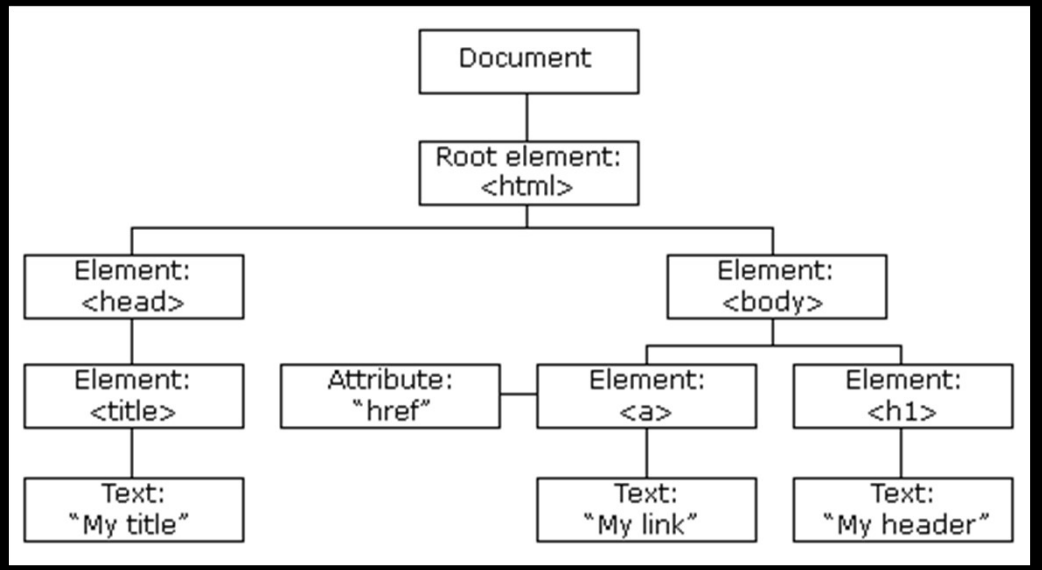
DOM Nodes: According to the W3C HTML DOM standard, everything in an HTML document is a node.

The entire document is a document node. Every HTML element is an element node. The text inside HTML elements are text nodes. Every HTML attribute is an attribute node (deprecated). All comments are comment nodes.

With the HTML DOM, all nodes in the node tree can be accessed by JavaScript.

New nodes can be created, and all nodes can be modified or deleted.

Source:
https://www.w3schools.com/js/js_htmldom.asp



The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a Document Object Model of the page.

The HTML DOM model is constructed as a tree of Objects.

HTML DOM Navigation 2

Node Relationships:

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

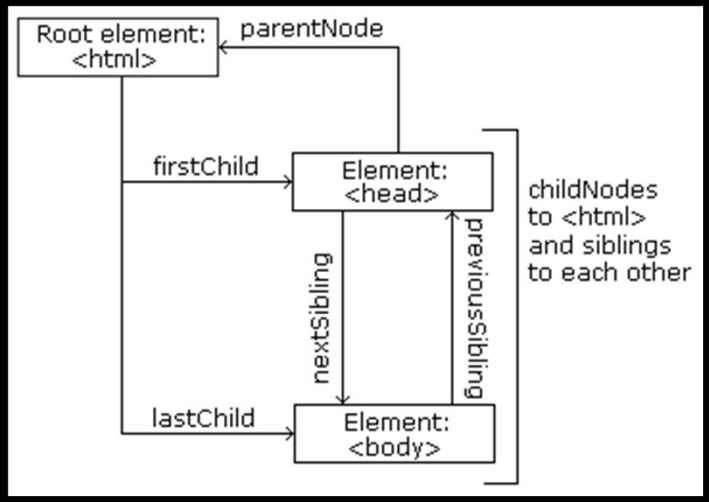
In a node tree, the top node is called the root (or root node).

Every node has exactly one parent, except the root (which has no parent).

A node can have a number of children.

Siblings (brothers or sisters) are nodes with the same parent.

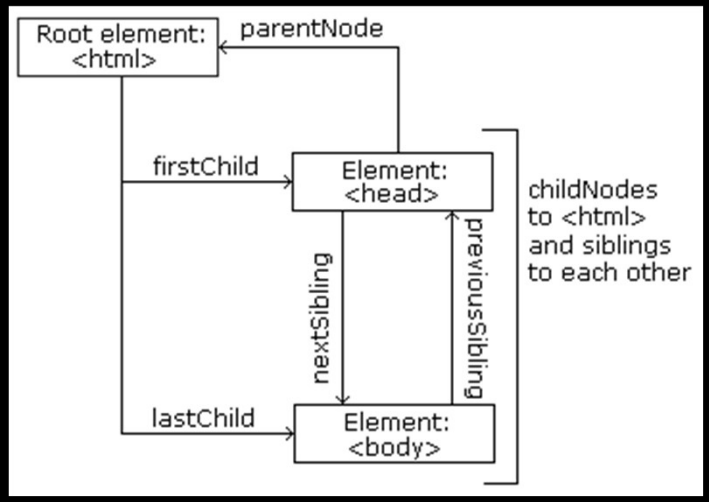
Source: https://www.w3schools.com/js/js_htmldom_navigation.asp




```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```

HTML DOM Navigation 3

```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```



From the html above you can read: <html> is the root node/<html> has no parents /<html> is the parent of <head> and <body>/<head> is the first child of <html>/<body> is the last child of <html>.

and:

<head> has one child: <title>/<title> has one child (a text node): "DOM Tutorial"/<body> has two children: <h1> and <p>/<h1> has one child: "DOM Lesson one"/<p> has one child: "Hello world!"/<h1> and <p> are siblings.

Source: https://www.w3schools.com/js/js_htmldom_navigation.asp

HTML DOM Navigation 4

Navigating Between Nodes:

You can use the following node properties to navigate between nodes with JavaScript:

parentNode

childNodes[nodenumber]

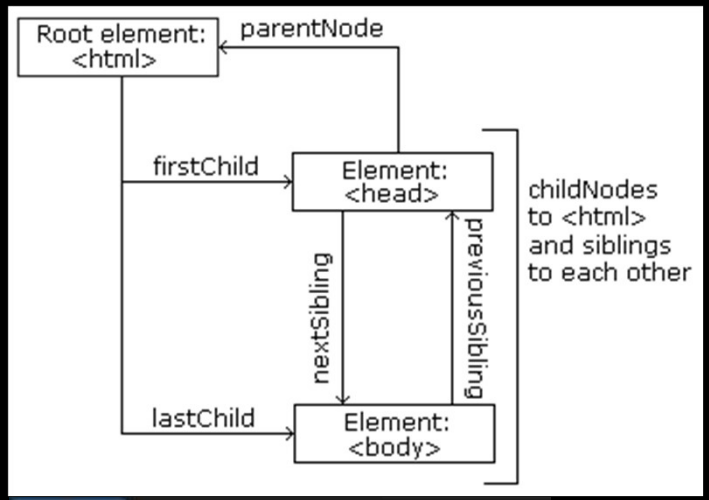
firstChild

lastChild

nextSibling

previousSibling

Source: https://www.w3schools.com/js/js_htmldom_navigation.asp




```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```

HTML DOM: Element Interface

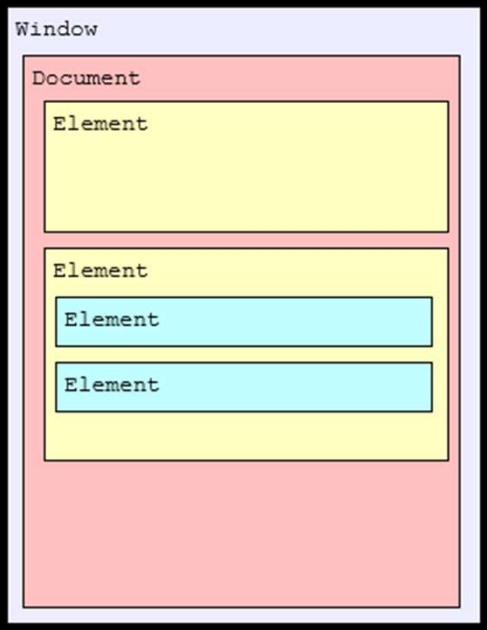
Nodes do not have any concept of including the content that is actually displayed in the document. They are empty vessels.

The fundamental notion of a node that can represent visual content is introduced by the Element interface.

An Element object instance represents a single element in a document created using either HTML or an XML vocabulary such as SVG.

For example, consider a document with two elements, one of which has two more elements nested inside it:

Source: https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API



While the Document interface is defined as part of the DOM specification, the HTML specification significantly enhances it to add information specific to using the DOM in the context of a web browser, as well as to using it to represent HTML documents specifically.

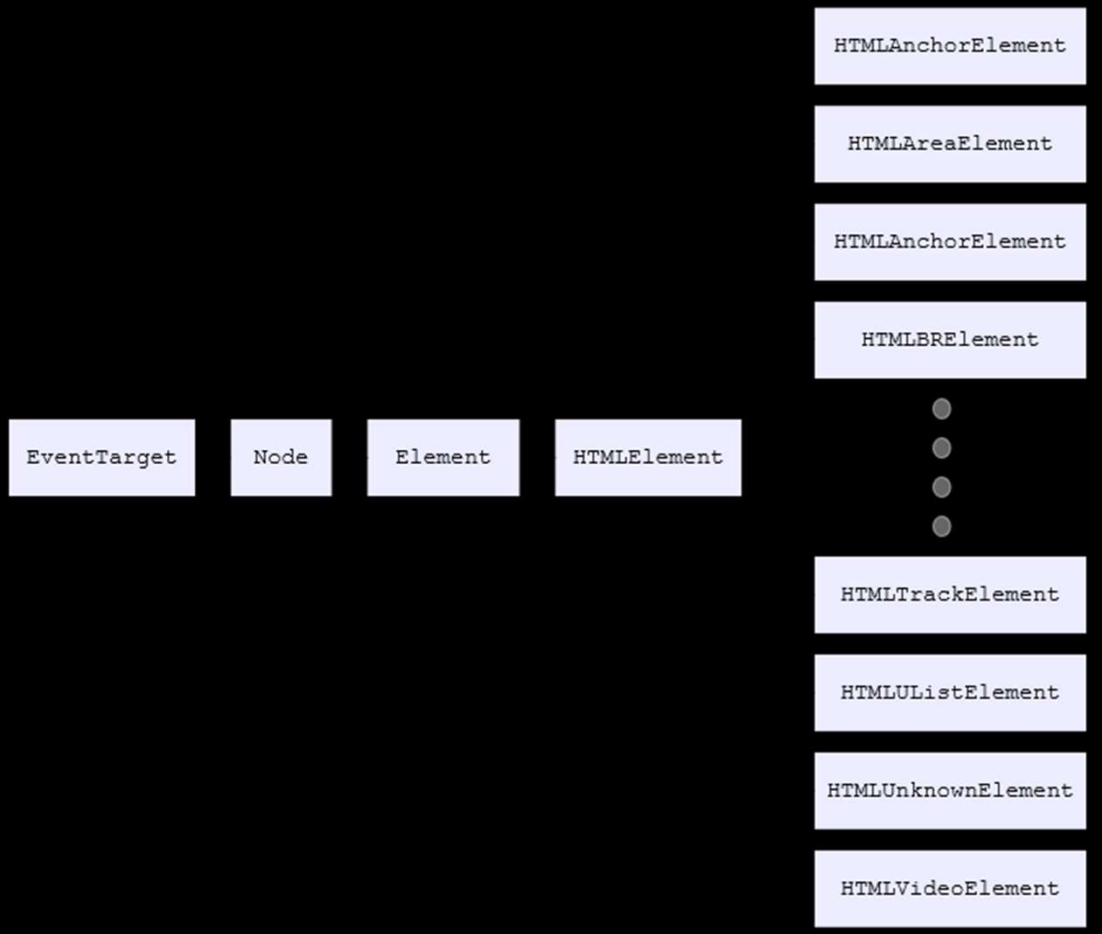
HTML DOM: HTML Element Interface

In order to expand upon the functionality of the core HTMLElement interface to provide the features needed by a specific element, the HTMLElement class is subclassed to add the needed properties and methods.

For example, the <canvas> element is represented by an object of type HTMLCanvasElement. HTMLCanvasElement augments the HTMLElement type by adding properties such as height and methods like getContext() to provide canvas-specific features.

The overall inheritance for HTML element classes looks like this:

Source: https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API



HTML DOM Methods

HTML DOM methods are actions you can perform (on HTML Elements).

HTML DOM properties are values (of HTML Elements) that you can set or change.

The HTML DOM can be accessed with JavaScript (and with other programming languages).

In the DOM, all HTML elements are defined as objects.

The programming interface is the properties and methods of each object.

Source: https://www.w3schools.com/js/js_htmldom_methods.asp

A property is a value that you can get or set (like changing the content of an HTML element).

A method is an action you can do (like add or deleting an HTML element).

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Page</h2>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>
</body>
</html>
```

The above example changes the content (the innerHTML) of the <p> element with id="demo":

In the example above, getElementById is a method, while innerHTML is a property.

HTML DOM Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- HTML web page has finished loading
- HTML input field was changed
- HTML button was clicked

Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.

Sources:
https://www.w3schools.com/js/js_events.asp and https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

HTML allows event handler attributes, with JavaScript code, to be added to HTML elements.

It is more common to see event attributes calling functions.

The Event interface represents an event which takes place in the DOM.

The Document Object Model (DOM) is a programming interface for HTML and XML documents. A Web page is a document.

This article offers a list of events:

https://developer.mozilla.org/en-US/docs/Web/Events

HTML DOM Events 2

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

Common HTML Events:

Here is a list of some common HTML events: Event handlers can be used to handle, and verify, user input, user actions, and browser actions. Many different methods can be used to let JavaScript work with events.

DOM Events are sent to notify code of interesting things that have taken place. Each event is represented by an object which is based on the Event interface and may have additional custom fields and/or functions used to get additional information about what happened.

Events can represent everything from basic user interactions to automated notifications of things happening in the rendering model. Some are standard events defined in official specifications, while others are events used internally by specific browsers.

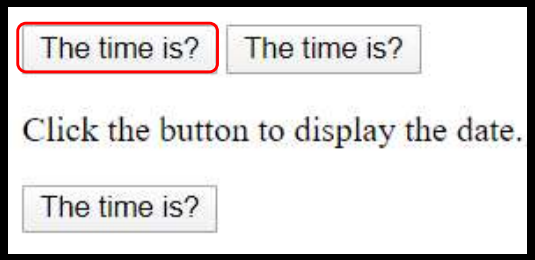
Sources: https://www.w3schools.com/js/js_events.asp and https://developer.mozilla.org/en-US/docs/Web/Events

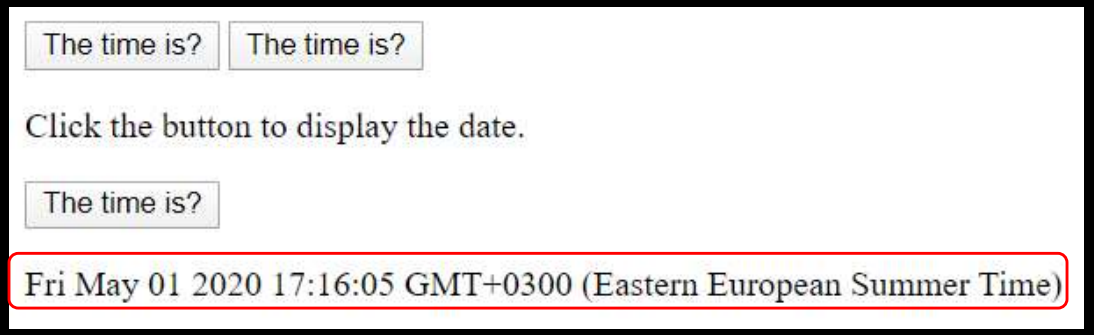
HTML DOM Events 3

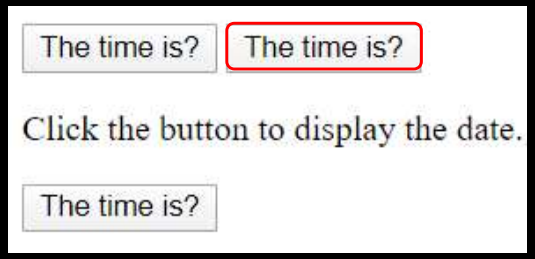
```
<!DOCTYPE html>
<html>
<body>
<!--In the following example, an onclick attribute (with code), is added to a <button> element:-->
<button onclick="document.getElementById('demo').innerHTML=Date()">The time is?</button>
<!--In the example above, the JavaScript code changes the content of the element with id="demo".
In the next example, the code changes the content of its own element (using this.innerHTML):-->
<button onclick="this.innerHTML=Date()">The time is?</button>
<!--JavaScript code is often several lines long. It is more common to see
event attributes calling functions:-->
<p>Click the button to display the date.</p>
<button onclick="displayDate()">The time is?</button>
<script>
    function displayDate() {
        document.getElementById("demo").innerHTML = Date();
    }
</script>
<p id="demo"></p>
</body>
</html>
```

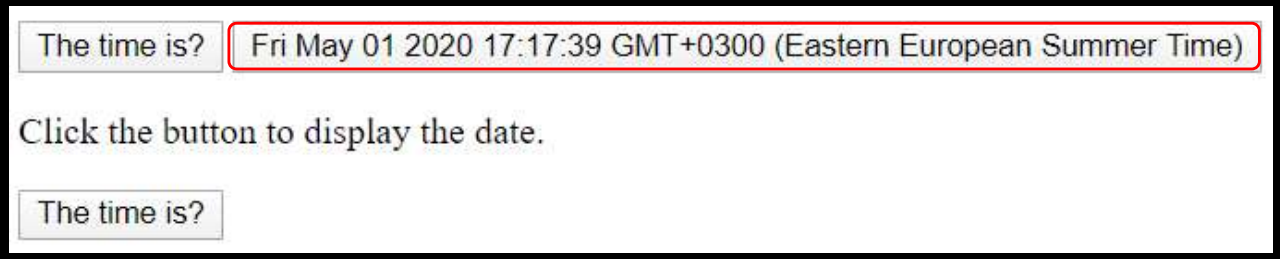
Source: https://www.w3schools.com/js/js_events.asp

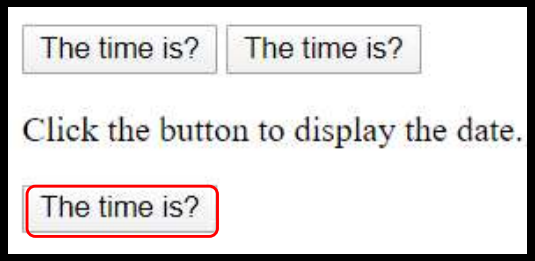
HTML DOM Events 4

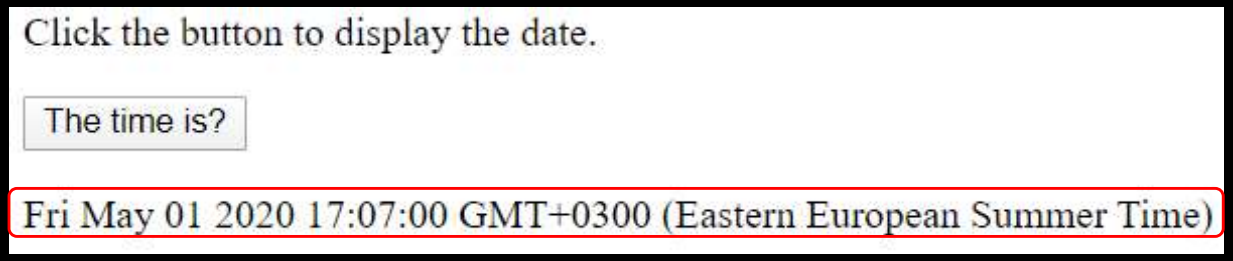












Source: https://www.w3schools.com/js/tryit.asp?filename=tryjs_event_onclick

The Browser Object Model (BOM)

There are no official standards for the Browser Object Model (BOM).

The window object is supported by all browsers. It represents the browser's window. The Document Object Model (DOM) is an architecture that describes the structure of a document.

All global JavaScript objects, functions, and variables automatically become members of the window object.

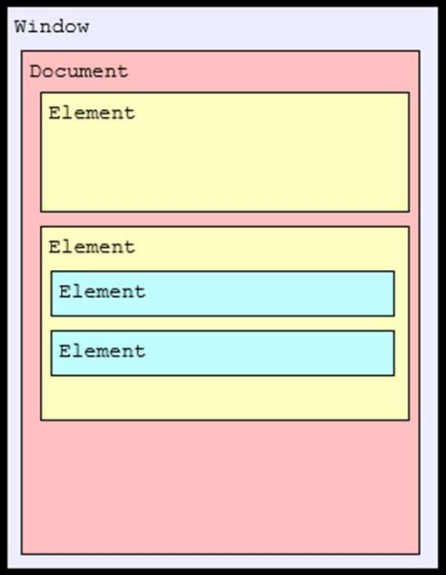
Global variables are properties of the window object.

Global functions are methods of the window object.

The document object (of the HTML DOM) is a property of the window object:

Source: https://www.w3schools.com/js/js_window.asp

On the MDN Mozilla web pages window and Web API are used instead of BOM (BOM is not mentioned at all).



Sources: https://developer.mozilla.org/en-US/docs/Web/API/Window and https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API

The Browser Object Model (BOM) 2

BOM main task is to manage browser windows and enable communication between the windows.

Each HTML page which is loaded into a browser window becomes a Document object and document object is an object in the BOM.

You can say BOM is super set of DOM. BOM has many objects, methods, and properties that are not the part of the DOM structure. The important BOM objects and methods are as:

- document
- location
- history
- navigator
- screen
- frames

```
// The document object (of the HTML DOM)
// is a property of the window object:
window.document.getElementById("header");
// is the same as:
document.getElementById("header");
// Many methods can be written
// without the window prefix
// like Popup Boxes window.alert(),
// window.confirm() and window.prompt().
alert("I am an alert box!");
```

- popup alert
- timing
- cookies

Sources: https://www.dotnettricks.com/learn/javascript/dom-bom and https://www.w3schools.com/js/js_window.asp

Where to place Javascript

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="javascript.js"></script>
</head>
<body>
    <script>console.log("Hello World!")</script>
</body>
</html>
```

- Option 1 In HTML, JavaScript code must be inserted between <script> and </script> tags. Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both. Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display.

- Option 2 Scripts can also be placed in external files. JavaScript files have the file extension .js. To use an external script, put the name of the script file in the src (source) attribute of a <script> tag.

Source: https://www.w3schools.com/js/js_whereto.asp

Modules

JavaScript programs can be in separate modules beeing imported when needed. Node.js has had this ability for a long time, and there are a number of JavaScript libraries and frameworks that enable module usage (for example, other CommonJS and AMD-based module systems like RequireJS, and more recently Webpack and Babel).

Exporting module features: The first thing you do to get access to module features is export them. This is done using the export statement.

You can export functions, var, let, const, and classes. They need to be top-level items; you can't use export inside a function, for example.

A more convenient way of exporting all the items you want to export is to use a single export statement at the end of your module file, followed by a comma-separated list of the features you want to export wrapped in curly braces. For example:

```
export { name, draw, reportArea, reportPerimeter };
```

Importing features into your script

Once you've exported some features out of your module, you need to import them into your script to be able to use them. The simplest way to do this is as follows:

```
import { name, draw, reportArea, reportPerimeter } from
'./modules/square.js';
```

You use the import statement, followed by a comma-separated list of the features you want to import wrapped in curly braces, followed by the keyword from, followed by the path to the module file.

We are using the dot (.) syntax to mean "the current location", followed by the path beyond that to the file we are trying to find. This is much better than writing out the entire relative path each time, as it is shorter, and it makes the URL portable.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Modules 2

Applying the module to your HTML

Now we just need to apply the main.js module to our HTML page. This is very similar to how we apply a regular script to a page, with a few notable differences.

First of all, you need to include type="module" in the <script> element, to declare this script as a module:

```
<script type="module" src="main.js"></script>
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

The script into which you import the module features basically acts as the top-level module. If you omit it, Firefox for example gives you an error of "SyntaxError: import declarations may only appear at top level of a module".

You can only use import and export statements inside modules; not regular scripts.

Note! You can also import modules into internal scripts, as long as you include type="module", for example <script type="module"> //include script here </script>.

Modules 3

Default exports versus named exports:

The functionality we've exported so far has been comprised of named exports — each item (be it a function, const, etc.) has been referred to by its name upon export, and that name has been used to refer to it on import as well.

There is also a type of export called the default export — this is designed to make it easy to have a default function provided by a module, and also helps JavaScript modules to interoperate with existing CommonJS and AMD module systems.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Aside: .mjs versus .js

Throughout this article, we've used .js extensions for our module files, but in other resources you may see the .mjs extension used instead. V8's documentation recommends this, for example.

The reasons given are:

It is good for clarity, i.e. it makes it clear which files are modules, and which are regular JavaScript.

It ensures that your module files are parsed as a module by runtimes such as Node.js, and build tools such as Babel.

However, we decided to keep to using .js.

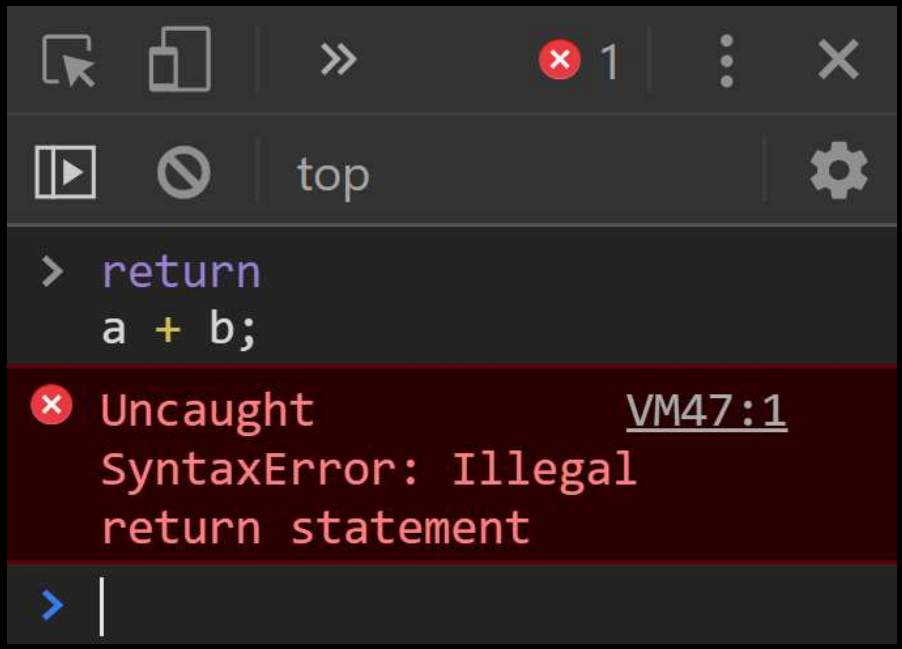
Modules 4

```
// Let's look at an example as we explain how it works.
// In our basic-modules square.js  you can find a function
// called randomSquare() that creates a square with a random color,
// size, and position. We want to export this as our default,
// so at the bottom of the file we write this:
export default randomSquare;
// Note the lack of curly braces.We could instead prepend export
// default onto the function and define it as an anonymous function:
export default function(ctx) {
  ...
}
// Over in our main.js file, we import the default function using
// this line:
import randomSquare from './modules/square.js';
// Again, note the lack of curly braces. This is because there is
// only one default export allowed per module, and we know that
// randomSquare is it. The above line is basically shorthand for:
import {default as randomSquare} from './modules/square.js';
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Automatic semicolon insertion (ASI)

```
1  // The return statement is affected by automatic
2  // semicolon insertion (ASI).
3  // No line terminator is allowed between
4  // the return keyword and the expression.
5  return
6  a + b;
7  // Under the hood return statement is transformed
8  // by ASI into:
9  return;
10 a + b;
11 // The console will thow an error
12 // To avoid this problem (to prevent ASI),
13 // you could use parentheses:
14 return (
15     a + b
16 );
```



On the line 9 a semi-colon is automatically inserted by ASI.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/return

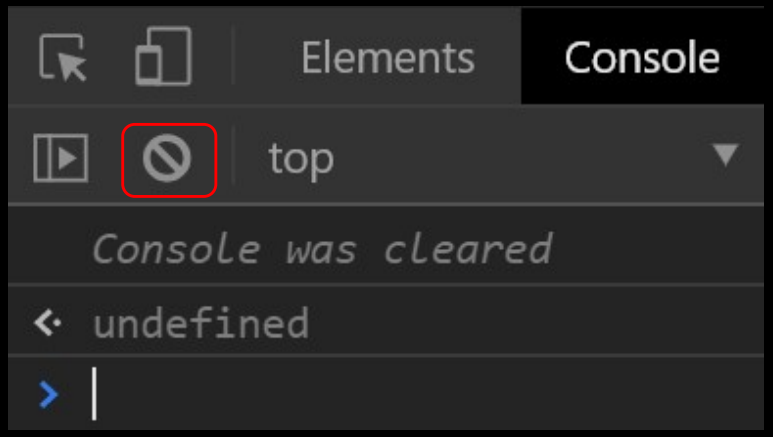
This is a Google Crome web browser screenshot of a console.

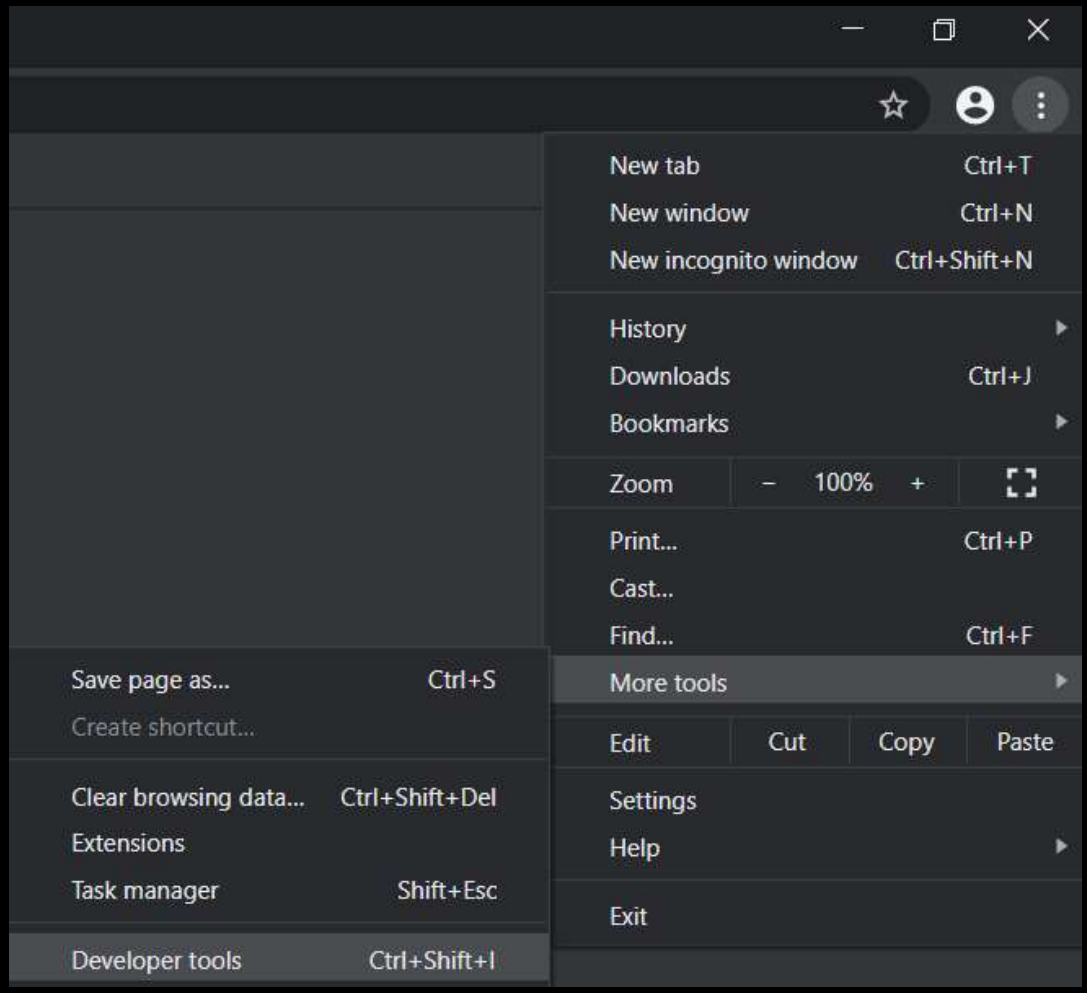
It is recommended to use Google Crome web browser over Edge or Internet Explorer browsers.

Web Browser Console

You can access Google Crome web browser console by opening the browser and selecting three dots from the upper right corner – more tools – developer tools or with a key combination ctrl +shift+i or with a keystroke of F12

You can clear the console by writing clear() to the console and press enter or by clicking the sign circled in red or with a key combination Ctlr + L.





Web Browser Console 2

```
1  // Styling console output: You can use the %c directive
2  // to apply a CSS style to console output:
3  console.log("This is %cMy log", "color: yellow;background-color: blue;padding: 2px");
4  // The text before the directive will not be affected, but the text after the directive
5  // will be styled using the CSS declarations in the parameter.
6  // Note: The console message behaves like an inline element by default. To see the effects
7  // of padding, margin, etc. you should set it to for example display: inline-block.
```

```
> console.log("This is %cMy log", "color: yellow;background-color:
  blue;padding: 2px");

  This is My log

<· undefined
```

Anything you type into the console at the prompt will be evaluated as a Javascript expression and the results of the expression will be echoed beneath the command prompt. If an error occurs, the stack trace of the error will be output instead. If the expression doesn't return a value, the console will report undefined. It's just letting you know that the previous expression didn't return a result.

Sources: https://developer.mozilla.org/en-US/docs/Web/API/Console and https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art041

Web Browser Console 3

```
> console.log("This is the outer level");
  console.group("First group");
  console.log("In the first group");
  console.group("Second group");
  console.log("In the second group");
  console.warn("Still in the second group");
  console.groupEnd();
  console.log("Back to the first group");
  console.groupEnd();
  console.debug("Back to the outer level");
This is the outer level
▼ First group
    In the first group
  ▼ Second group
      In the second group
    ⚠ ▸ Still in the second group
    Back to the first group
undefined
```

```
> for (var i=0; i<5; i++) {
    console.log("Hello, %s. You've called me %d times.", "Bob", i+1);
  }
Hello, Bob. You've called me 1 times.
Hello, Bob. You've called me 2 times.
Hello, Bob. You've called me 3 times.
Hello, Bob. You've called me 4 times.
Hello, Bob. You've called me 5 times.
undefined
```

```
> console.time("answer time");
  alert("Click to continue");
  console.timeLog("answer time");
  alert("Do a bunch of other stuff...");
  console.timeEnd("answer time");
answer time: 2912.160888671875ms
answer time: 5076.694091796875ms
undefined
```

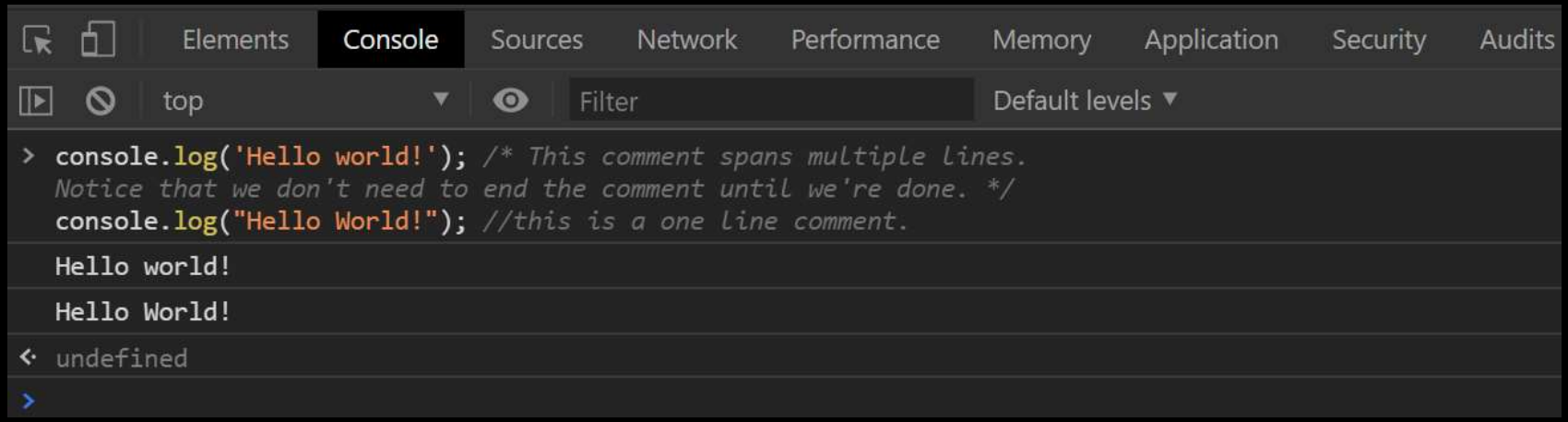
```
> function foo() {
    function bar() {
      console.trace();
    }
    bar();
  }

  foo();
▼ console.trace
    bar          @ VM41:3
    foo          @ VM41:5
    (anonymous)  @ VM41:8
undefined
```

From left to right: Using groups in the console,

using string substitutions, timers and stack traces.

Source: https://developer.mozilla.org/en-US/docs/Web/API/console

Console.log()



It is possible to test your Javasript by using console.log() in the browser console. At the end, console.log() always prints a newline. Therefore, if you call it with zero arguments, it just prints a newline.

JavaScript accepts both "double" and 'single' quotes. You can use quotes inside a string, as long as they don't match the quotes surrounding the string. "I am John 'the man' " and 'I am John "the man" '.

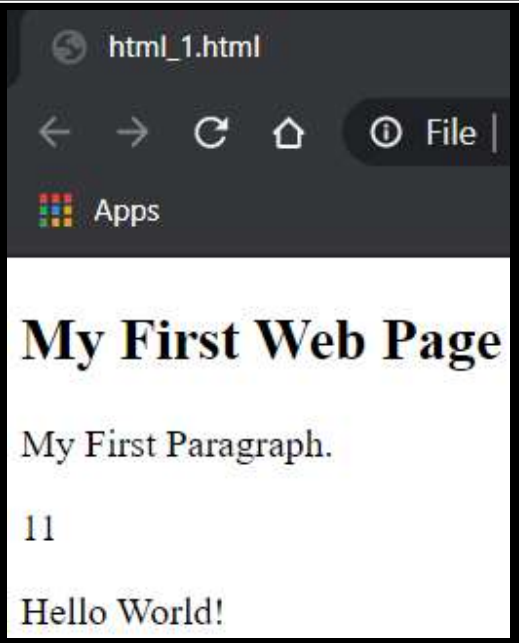
JavaScript comments: Single line comments start with //. JavaScript Block comments: Multi-line comments start with /* and end with */. Use single line comments. Block comments are often used for formal documentation.

Sources: https://www.w3schools.com/js/js_intro.asp and https://www.w3schools.com/js/js_comments.asp

Document.write() and Document.getElementById()

html_1.html

```
1   <!DOCTYPE html>
2   <html>
3   <body>
4   <h2>My First Web Page</h2>
5   <p>My First Paragraph.</p>
6   <p id="demo"></p>
7   <script>
8   document.write("Hello World!")
9   document.getElementById("demo").innerHTML = 5 + 6;
10  </script>
11  </body>
12  </html>
```



For testing purposes, it is convenient to use document.write(). WARNING ! Using document.write() after an HTML document is loaded, will delete all existing HTML.

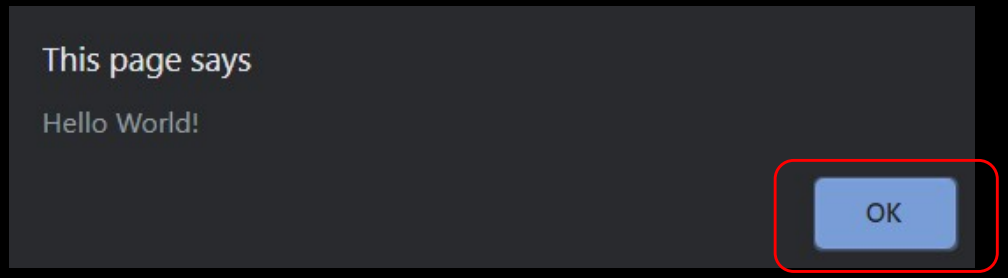
Source: https://www.w3schools.com/js/js_output.asp

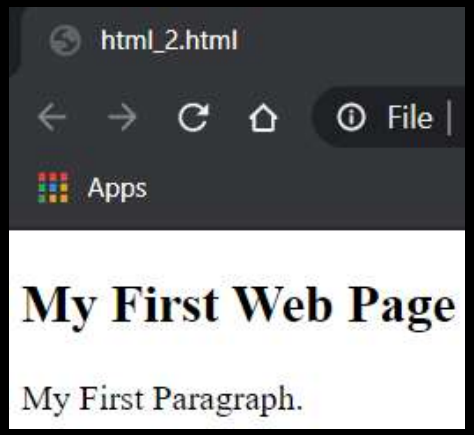
To access an HTML element, JavaScript can use the document.getElementById(id) method. The id attribute defines the HTML element. The innerHTML property defines the HTML content.

Window.alert()

html_2.html

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4  <h2>My First Web Page</h2>
5  <p>My First Paragraph.</p>
6  <script>
7  window.alert("Hello World!")
8  </script>
9  </body>
10 </html>
```





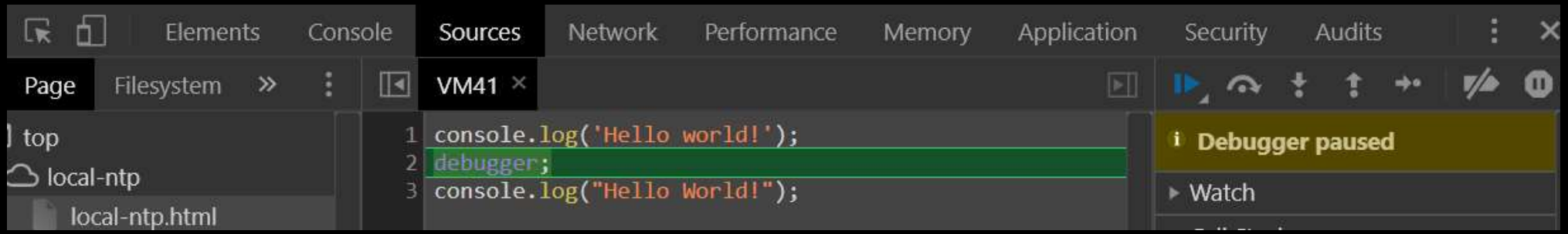
Note! This page says Hello World! alert has to be clicked ok before the actual webpage is shown.

The alert() method displays an alert box with a specified message and an OK button. An alert box is often used if you want to make sure information comes through to the user.

Source: https://www.w3schools.com/js/js_output.asp

Debugger

```
1  console.log('Hello world!');
2  debugger;
3  console.log("Hello World!");
```



Searching for (and fixing) errors in programming code is called code debugging. The first known computer bug was a real bug (an insect) stuck in the electronics. Errors are also called bugs.

The debugger keyword stops the execution of JavaScript, and calls (if available) the debugging function.

In the debugger window, you can set breakpoints in the JavaScript code. At each breakpoint, JavaScript will stop executing, and let you examine JavaScript values. After examining values, you can resume the execution of code (typically with a play button).

Source: https://www.w3schools.com/js/js_debugging.asp

V8 Javascript Engine

V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++.

It is used in Chrome and in Node.js, among others. It implements ECMAScript and WebAssembly, and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors.

V8 can run standalone or can be embedded into any C++ application.

V8 compiles and executes JavaScript source code, handles memory allocation for objects, and garbage collects objects it no longer needs. V8's stop-the-world, generational, accurate garbage collector is one of the keys to V8's performance.

JavaScript is commonly used for client-side scripting in a browser, being used to manipulate Document Object Model (DOM) objects for example. The DOM is not, however, typically provided by the JavaScript engine but instead by a browser.

The same is true of V8. Google Chrome provides the DOM. V8 does however provide all the data types, operators, objects and functions specified in the ECMA standard.

V8 enables any C++ application to expose its own objects and functions to JavaScript code. It's up to you to decide on the objects and functions you would like to expose to JavaScript.

Sources: https://v8.dev/ and https://v8.dev/docs

Browser Engine: Rendering Engines and Javascript Engines

Browser Engine comprises of two parts: Rendering Engine and Javascript Engine						
Browser	Crome	Internet Explorer	Firefox	Edge	Opera	Safari
Rendering Engine	Blink	Trident	Gecko	EdgeHTML	Blink	WebKit
Javascript Engine	V8	Chakra	SpiderMonkey	Chakra	V8	Nitro

A browser engine can be thought of as the heart of a browser, it helps to present the content of a website after you hit the URL of the web address you wish to visit. It does so by understanding the HTML, CSS, and JavaScript written on your web page.

A browser engine is responsible for everything from the text you see in the HTML code to the graphical presentation of that code on the browser with the help of Javascript engine and rendering engine.

Browser engines are uniquely designed for every browser leading to cross browser compatibility issues.

Source: https://www.lambdatest.com/blog/browser-engines-the-crux-of-cross-browser-compatibility/

Memory Management

JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection).

Memory life cycle: Regardless of the programming language, the memory life cycle is pretty much always the same.

- Allocate the memory you need.
- Use the allocated memory (read, write).
- Release the allocated memory when it is not needed anymore.

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

JavaScript, utilize a form of automatic memory management known as garbage collection (GC).

The purpose of a garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it.

This automatic process is an approximation since the general problem of determining whether or not a specific piece of memory is still needed is undecidable.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

Memory Management 2

```
1  // Value initialization: In order to not bother the programmer with allocations,
2  // JavaScript will automatically allocate memory when values are initially declared.
3  var n = 123; // allocates memory for a number
4  var s = 'azerty'; // allocates memory for a string
5  var o = {
6    a: 1,
7    b: null
8  }; // allocates memory for an object and contained values
9  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
10 // (like object) allocates memory for the array and contained values
11 var a = [1, null, 'abra'];
12 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13 function f(a) {
14   return a + 2;
15 } // allocates a function (which is a callable object)
16 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
17 // function expressions also allocate an object
18 someElement.addEventListener('click', function() {
19   someElement.style.backgroundColor = 'blue';
20 }, false);
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

Syntax: Overview

```
// An Overview of the Syntax. A few examples of syntax:
// Two slashes start single-line comments
var x; // declaring a variable
x = 3 + y; // assigning a value to the variable `x`
foo(x, y); // calling function `foo` with parameters `x` and `y`
obj.bar(3); // calling method `bar` of object `obj`
// A conditional statement
if (x === 0) { // Is `x` equal to zero?
    x = 123;
}
// Defining function `baz` with parameters `a` and `b`
function baz(a, b) {
    return a + b;
}
// Note the two different uses of the equals sign: A single equals sign (=)
// is used to assign a value to a variable. A triple equals sign (===)
// is used to compare two values
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Compound Assignment Operators:There are compound assignment operators such as +=.
// The following two assignments are equivalent:
x += 1;
x = x + 1;
```

Source: http://speakingjs.com/es5/ch01.html

Video Tip!
https://www.youtube.com/watch?v=2pL28CcEijU
What the... JavaScript? | Kyle Simpson

Video Tip!
https://www.youtube.com/watch?v=aX3ZABCdC38
dotJS 2017 A Brief History of JavaScript | Brendan Eich

Syntax: Trailing Commas

```
1  // Trailing commas in parameter definitions are now legal:
2  function foo(
3      param1,
4      param2,
5  ) {}
6  // Similarly, trailing commas in function calls are now also legal:
7  foo(
8      'abc',
9      'def',
10 );
11 // Trailing commas are ignored in object literals:
12 let obj = {
13     first: 'Jane',
14     last: 'Doe',
15 };
16 // And they are also ignored in Array literals:
17 let arr = [
18     'red',
19     'green',
20     'blue',
21 ];
22 console.log(arr.length); // 3
```

Source: https://exploringjs.com/es2016-es2017/ch_trailing-comma-parameters.html

Video Tip!
https://www.youtube.com/watch?v=fX5HhsINAbc
Promises, Generators, Async (JavaScript training) | Kyle Simpson

Video Tip!
https://www.youtube.com/watch?v=9rW0H5FvWeA
The Functional Programming Concept You Should be Using Right now | Bianca Gandolfo

Video Tip!
https://www.youtube.com/watch?v=8W6zoAmYVK8
JavaScript Fundamentals for Functional Programming (JavaScript training) | Bianca Gandolfo

Syntax: Semicolons

```
1   // Rule of thumb for semicolons
2   // Each statement is terminated by a semicolon:
3   const x = 3;
4   someFunction('abc');
5   i++;
6   // except statements ending with blocks:
7   function foo() {
8     // ...
9   }
10  if (y > 0) {
11    // ...
12  }
13  // The following case is slightly tricky:
14  const func = () => {}; // semicolon!
15  // The whole const declaration (a statement) ends with a semicolon, but inside it,
16  // there is an arrow function expression. That is, it's not the statement per se
17  // that ends with a curly brace; it's the embedded arrow function expression.
18  // That's why there is a semicolon at the end.
```

Video Tip!
https://www.youtube.com/watch?v=hQVTIJBZook
JavaScript: The Good Parts | Douglas Crockford

Video Tip!
https://www.youtube.com/watch?v=lil4YCCXRYc
Async Programming in ES7 JSConf US 2015 | Jafar Husain

Source: https://exploringjs.com/impatient-js/ch_syntax.html#semicolons

Syntax: Statements versus Expressions

```
// To understand JavaScript's syntax, you should know that it has two major syntactic categories:
// statements and expressions: Statements "do things." A program is a sequence of statements.
// Here is an example of a statement, which declares (creates) a variable foo:
var foo;
// Expressions produce values. They are function arguments, the right side of an assignment, etc.
// Here's an example of an expression:
3 * 7
// The distinction between statements and expressions is best illustrated by the fact that
// JavaScript has two different ways to do if-then-else—either as a statement:
var x;
if (y >= 0) {
    x = y;
} else {
    x = -y;
}
// or as an expression:
var x = y >= 0 ? y : -y;
// You can use the latter as a function argument (but not the former):
myFunction(y >= 0 ? y : -y)
// Finally, wherever JavaScript expects a statement, you can also use an expression; for example:
foo(7, 1);
// The whole line is a statement (a so-called expression statement),
// but the function call foo(7, 1) is an expression.
```

Video Tip!
https://www.youtube.com/watch?v=qouPzSryggk
Netflix JavaScript Talks - Human Performance | Jem Young

Source: http://speakingjs.com/es5/ch01.html

Syntax: Statements versus expressions 2

```
// Statements: A statement is a piece of code that can be executed and
// performs some kind of action. For example, if is a statement:
let myStr;
if (myBool) {
  myStr = 'Yes';
} else {
  myStr = 'No';
}
// One more example of a statement: a function declaration.
function twice(x) {
  return x + x;
}
// An expression is a piece of code that can be evaluated to produce a value.
// For example, the code between the parentheses is an expression:
let myStr = (myBool ? 'Yes' : 'No');
// The operator _?_:_ used between the parentheses is called the ternary operator.
// It is the expression version of the if statement.
// More examples of expressions. We enter expressions and the REPL evaluates them for us:
'ab' + 'cd' // 'abcd'
Number('123') // 123
true || false // true
```

Video Tip!
https://www.youtube.com/watch?v=8aGhZQkoFbQ
What the heck is the event loop anyway? JSConf EU | Philip Roberts

Source: https://exploringjs.com/impatient-js/ch_syntax.html#an-overview-of-javascripts-syntax

Syntax: Statements versus expressions 3

```
// The current location within JavaScript source code determines which
// kind of syntactic constructs you are allowed to use:
// The body of a function must be a sequence of statements:
function max(x, y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
// The arguments of a function call or a method call must be expressions:
console.log('ab' + 'cd', Number('123'));
// However, expressions can be used as statements. Then they are called expression statements.
// The opposite is not true: when the context requires an expression, you can't use a statement.
// The following code demonstrates that any expression bar() can be either
// expression or statement – it depends on the context:
function f() {
  console.log(bar()); // bar() is expression
  bar(); // bar(); is (expression) statement
}
```

Video Tip!
https://www.youtube.com/watch?v=exrc_rLj5iw
An Introduction to Functions, Execution Context and the Call Stack (FULL VIDEO - Parts 1-3) | Will Sentence

Source: https://exploringjs.com/impatient-js/ch_syntax.html#an-overview-of-javascripts-syntax

Ambiguous Syntax

```
1  // JavaScript has several programming constructs that are syntactically ambiguous:
2  // the same syntax is interpreted differently, depending on whether it is used
3  // in statement context or in expression context.
4  // Same syntax: function declaration and function expression
5  A function declaration is a statement:
6  function id(x) {
7    return x;
8  }
9  // A function expression is an expression (right-hand side of =):
10 const id = function me(x) {
11   return x;
12 };
13 // Same syntax: object literal and block
14 // In the following code, {} is an object literal:
15 // an expression that creates an empty object.
16 const obj = {};
17 // This is an empty code block (a statement):
18 {
19 }
```

Video Tip!
https://www.youtube.com/watch?v=Bv_5Zv5c-Ts
JavaScript: Understanding the Weird Parts - The First 3.5 Hours | Tony Alicea

Source: https://exploringjs.com/impatient-js/ch_syntax.html#an-overview-of-javascripts-syntax

Syntax: The Temporal Dead Zone

A variable declared by let or const has a so-called temporal dead zone (TDZ): When entering its scope, it can't be accessed (got or set) until execution reaches the declaration. Let's compare the life cycles of var-declared variables (which don't have TDZs) and let-declared variables (which have TDZs).

The life cycle of var-declared variables: var variables don't have temporal dead zones. Their life cycle comprises the following steps:

When the scope (its surrounding function) of a var variable is entered, storage space (a binding) is created for it. The variable is immediately initialized, by setting it to undefined.

When the execution within the scope reaches the declaration, the variable is set to the value specified by the initializer (an assignment) – if there is one. If there isn't, the value of the variable remains undefined.

The life cycle of let-declared variables: Variables declared via let have temporal dead zones and their life cycle looks like this:

When the scope (its surrounding block) of a let variable is entered, storage space (a binding) is created for it. The variable remains uninitialized. Getting or setting an uninitialized variable causes a ReferenceError.

When the execution within the scope reaches the declaration, the variable is set to the value specified by the initializer (an assignment) if there is one. If there isn't then the value of the variable is set to undefined. const variables work similarly to let variables, but they must have an initializer (i.e., be set to a value immediately) and can't be changed.

Source: https://exploringjs.com/es6/ch_variables.html#sec_temporal-dead-zone

Syntax: The Temporal Dead Zone 2

```
// Examples: Within a TDZ, an exception is thrown if a variable is got or set:
let tmp = true;
if (true) { // enter new scope, TDZ starts. Uninitialized binding for `tmp` is created
    console.log(tmp); // ReferenceError
    let tmp; // TDZ ends, `tmp` is initialized with `undefined`
    console.log(tmp); // undefined
    tmp = 123;
    console.log(tmp); // 123
}
console.log(tmp); // true
// If there is an initializer then the TDZ ends after the initializer was evaluated
// and the result was assigned to the variable:
let foo = console.log(foo); // ReferenceError
// The following code demonstrates that the dead zone is really temporal (based on time)
// and not spatial (based on location):
if (true) { // enter new scope, TDZ starts
    const func = function () {
        console.log(myVar); // OK!
    }; // Here we are within the TDZ and accessing `myVar` would cause a `ReferenceError`
    let myVar = 3; // TDZ ends
    func(); // called outside TDZ
}
```

Source: https://exploringjs.com/es6/ch_variables.html#sec_temporal-dead-zone

Syntax: Static versus Dynamic

```
1  // There are two angles from which you can examine the workings of a program:
2  // Statically (or lexically): You examine the program as it exists in source code,
3  // without running it. Given the following code, we can make the static assertion
4  // that function g is nested inside function f:
5  function f() {
6      function g() {
7      }
8  }
9  // The adjective lexical is used synonymously with static, because both pertain to the lexicon
10 // (the words, the source) of the program.
11 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
12 // Dynamically: You examine what happens while executing the program ("at runtime").
13 // Given the following code:
14 function g() {
15 }
16 function f() {
17     g();
18 }
19 // when we call f(), it calls g(). During runtime, g being called by f
20 // represents a dynamic relationship.
```

Source: http://speakingjs.com/es5/ch16.html#_declaring_a_variable

Escape Characters

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

```
1  // The sequence \" inserts a double quote in a string:
2  var x = "We are the so-called \"Vikings\" from the north.";
3  // The sequence \' inserts a single quote in a string:
4  var x = 'It\'s alright.';
5  // The sequence \\ inserts a backslash in a string:
6  var x = "The character \\ is called backslash.";
7  // The string will be chopped to "We are the so-called ".
8  var x = "We are the so-called "Vikings" from the north.";
```

The escape characters above on the right were originally designed to control typewriters, teletypes, and fax machines. They do not make any sense in HTML.

Because strings must be written within quotes the text on the line 8 will be chopped to "We are the so-called".

Source: https://www.w3schools.com/js/js_strings.asp

Reserved Keywords as of ECMAScript 2015

Reserved keywords			
abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

In JavaScript you cannot use reserved keywords as variables, labels, or function names.

Words marked with* are new in ECMAScript 5 and ECMAScript 6.

Additionally, the literals null, true, and false cannot be used as identifiers in ECMAScript.

In the year 2015 ECMA International published ECMAScript 6 or ES6 which was later renamed (ECMAScript 2015).

Since 2015, ECMAScript-standard has been updated and named by year. ECMAScript 2020 is the newest version (https://tc39.es/ecma262/).

Source:
https://www.w3schools.com/js/js_reserved.asp

Operators

JavaScript has the following types of operators:

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- String operators
- Conditional (ternary) operator
- Comma operator
- Unary operators
- Relational operators

Note! (=>) is not an operator, but the notation for Arrow functions. Arrow functions can have either a "concise body" or the usual "block body".

In a concise body, only an expression is specified, which becomes the implicit return value. In a block body, you must use an explicit return statement.

```
var func = x => x * x;
// concise body syntax,
// implied "return"
var func = (x, y) => { return x + y; };
// with block body, explicit
// "return" needed
```

Source: https:/developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

Arithmetic and Assigment Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

Arithmetic operators on the left are used to perform arithmetic on numbers. Assignment operators on the right assign values to JavaScript variables.

Source: https://www.w3schools.com/js/js_operators.asp

Arithmetic operators: Exponentiation operator (**)

```
6 ** 2 // 36
// An infix operator for exponentiation
// ** is an infix operator for exponentiation:
x ** y
// produces the same result as
Math.pow(x, y)
// Examples normal use:
const squared = 3 ** 2; // 9
// Exponentiation assignment operator:
let num = 3;
num **= 2;
console.log(num); // 9
// Using exponentiation in a function (Pythagorean theorem):
function dist(x, y) {
  return Math.sqrt(x**2 + y**2);
}
// Precedence: The exponentiation operator binds very strongly,
// more strongly than * (which, in turn, binds more strongly than +):
2**2 * 2 // 8
2 ** (2*2) // 16
```

Source: https://exploringjs.com/es2016-es2017/ch_exponentiation-operator.html

Arithmetic Operators and Operands

Operand	Operator	Operand
100	+	50
a	*	b

```
var a = 4;
var b = 2;
console.log (a*b)
// logs 8
```

Arithmetic operators perform arithmetic on numbers (literals or variables). The numbers (in an arithmetic operation) are called operands. Literal number operands are 100 and 50 (valuates to 150). Variable operands are a and b (valuates to 8). The operation (to be performed between the two operands) is defined by an operator. Operators are (+) and (*).

```
// When used on strings, the + operator is called the concatenation operator.
fn = "Esa", ln= "Ketonen",
fullName = fn + " " + ln; // "Esa Ketonen"
// The += assignment operator can also be used to add (concatenate) strings:
var txt1 = "What a very ";
txt1 += "nice day"; // "What a very nice day"
// If you add a number and a string, the result will be a string!
var y = "5" + 5;
var z = "Hello" + 5;
console.log(y," ",z); // 55  Hello5
```

Source: https://www.w3schools.com/js/js_arithmetic.asp

Arithmetic Operators

```
1  // Remainder:The modulus operator (%) returns the division remainder.
2  var x = 5;
3  var y = 2;
4  var z = x % y;
5  console.log(z); // Result is 1
6  // Incrementing: The increment operator (++) increments numbers.
7  var x = 5;
8  x++;
9  var z = x;
10 console.log(z); // Result is 6
11 // Decrementing: The decrement operator (--) decrements numbers.
12 var x = 5;
13 x--;
14 var z = x;
15 console.log(z); // Result is 4
16 // Exponentiation x ** y produces the same result as Math.pow(x,y):
17 var x = 5;
18 var z = Math.pow(x,2);
19 console.log(z); // result is 25
```

Source: https://www.w3schools.com/js/js_arithmetic.asp

Arithmetic Operator Precedence

Precedence	Operator type	Associativity	Individual operators
21	Grouping	n/a	(…)
20	Member Access	left-to-right	… . …
	Computed Member Access	left-to-right	… […]
	new (with argument list)	n/a	new … (…)
	Function Call	left-to-right	… (…)

Operator precedence describes the order in which operations are performed in an arithmetic expression. Expressions in parentheses are computed before the value is used in the rest of the expression (value 21)

The precedence can be changed by using parentheses. When many operations have the same precedence (like addition and subtraction value 14), they are computed from left to right. (The highest precedence value is 21)

Multiplication (*) and division (/) have higher precedence value 15 than addition (+) and subtraction (-) value 14.

Sources: https://www.w3schools.com/js/js_arithmetic.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#Table

Comparison, Logical, Typeof and Instanceof Operators

Operator	Description
&&	logical and
||	logical or
!	logical not
Operator	**Description**
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

Comparison and logical operators are used to test for true or false. You can use the typeof operator to find the data type of a JavaScript variable. The instanceof operator returns true if the specified object is an instance of the specified object.

Source: https://www.w3schools.com/js/js_operators.asp

Logical and Comparison Operators

```
// Logical Operators: Logical operators are used to determine the logic between
// variables or values.
x = 6;
y = 3;
(x < 10 && y > 1); // true
(x == 5 || y == 5); // false
!(x == y); // true
// Comparing Different Types:Comparing data of different types may give unexpected
// results.When comparing a string with a number, JavaScript will convert the string
// to a number when doing the comparison. An empty string converts to 0.
// A non-numeric string converts to NaN which is always false.
2 < 12 //true
2 < "12" //true
2 < "John" //false
2 > "John" // false
2 == "John" //false
"2" < "12" //false
"2" > "12" //true
"2" == "12" //false
// When comparing two strings, "2" will be greater than "12", because (alphabetically)
// 1 is less than 2. To secure a proper result, variables should be converted to the
// proper type before comparison.
```

Source: https://www.w3schools.com/js/js_comparisons.asp

Bitwise Operations

JavaScript Uses 32 bits Bitwise Operands

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.

Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers.

After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.

```
1  // Converting Decimal to Binary:
2  dec2bin(-5);
3  function dec2bin(dec){
4    return (dec >>> 0).toString(2);
5  }
6  // returns "11111111111111111111111111111011"
7  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8  // Converting Binary to Decimal:
9  bin2dec(101);
10 function bin2dec(bin){
11   return parseInt(bin, 2).toString(10);
12 }
13 // returns "5"
```

Source:
https://www.w3schools.com/js/js_bitwise.asp

Bitwise Operations 2

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

JavaScript bitwise operators and examples.

The examples above uses 4 bits unsigned binary numbers. Because of this ~ 5 returns 10. Since JavaScript uses 32 bits signed integers, it will not return 10. It will return -6.

Source:
https://www.w3schools.com/js/js_bitwise.asp

Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
5 | 1	5	0101 | 0001	0101
~ 5	10	~0101	1010
5 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

String Operators

```
1  // The + operator can also be used to add (concatenate) strings.
2  var txt1 = "John";
3  var txt2 = "Doe";
4  var txt3 = txt1 + " " + txt2;
5  console.log(txt3); // John Doe
6  // The += assignment operator can also be used to add (concatenate) strings:
7  var txt1 = "What a very ";
8  txt1 += "nice day"; // "What a very nice day"
9  // When used on strings, the + operator is called the concatenation operator.
10 // Adding two numbers, will return the sum, but adding a number and a string
11 // will return a string:
12 var x = 5 + 5;
13 var y = "5" + 5;
14 var z = "Hello" + 5;
15 console.log(x,y,z); // 10 "55" "Hello5"
16 // If you add a number and a string, the result will be a string!
```

Source: https://www.w3schools.com/js/js_operators.asp

Unary, Binary and Ternary Operators

```
1   // JavaScript has both binary and unary operators,
2   // and one special ternary operator, the conditional operator.
3   // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
4   // A binary operator requires two operands,
5   // one before the operator and one after the operator:
6   3+4;
7   x*y;
8   // A unary operator requires a single operand,
9   // either before or after the operator:
10  x++;
11  ++x;
12  // The conditional operator is the only JavaScript operator
13  // that takes three operands.
14  // The operator can have one of two values based on a condition:
15  var age = 49
16  var status = (age >= 18) ? 'adult' : 'minor';
17  console.log(status) // returns adult
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

Unary Operators: Void Operator

```
// The void Operator. Here are some examples:
// The syntax for the void operator is: void «expr»
// which evaluates expr and returns undefined.
void 0 // undefined
void (0) // undefined
void 4+7 // same as (void 4)+7 // NaN
void (4+7) // undefined
var x;
x = 3 // 3
void (x = 5)
undefined
x // 5
// Thus, if you implement void as a function, it looks as follows:
function myVoid(expr) {
    return undefined;
}
// The void operator is associated closely with its operand,
// so use parentheses as necessary. For example, void 4+7 binds as (void 4)+7.
// if you want to open a new window without changing
// the currently displayed content, you can do the following:
void window.open("http://example.com/")
```

Source: speakingjs.com/es5/ch09.html#_special_operators

Unary Operators: Typeof Operator

Typeof operator always returns a string (containing the type of the operand).

The data type of NaN is number

The data type of an array is object

The data type of a date is object

The data type of null is object

The data type of an undefined variable is undefined and a variable that has not been assigned a value is also undefined.

You cannot use typeof to determine if a JavaScript object is an array (or a date).

Source: https://www.w3schools.com/js/js_type_conversion.asp

```
// The typeof operator returns the type of a
// variable, object, function or expression.
typeof "john" + "\n" +
typeof 3.14 + "\n" +
typeof NaN + "\n" +
typeof false + "\n" +
typeof [1,2,3,4] + "\n" +
typeof {name:'john', age:34} + "\n" +
typeof new Date() + "\n" +
typeof function () {} + "\n" +
typeof myCar + "\n" +
typeof null;
// "string
// number
// number
// boolean
// object
// object
// object
// function
// undefined
// object"
```

Comma Operator and Conditional (Ternary) Operator (? :)

```
// The Comma Operator «left» , «right»
// The comma operator evaluates both operands and returns the result of right.
// Roughly, it does for expressions what the semicolon does for statements.
// This example demonstrates that the second operand becomes the result of the operator:
123, 'abc' // 'abc'
// This example demonstrates that both operands are evaluated:
var x = 0;
var y = (x++, 10);
x // 1
y // 10
// The comma operator is confusing. It's better to not be clever and
// to write two separate statements whenever you can.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The Conditional Operator ( ? : )
// The conditional operator is an expression:
// «condition» ? «if_true» : «if_false»
// If the condition is true, the result is if_true; otherwise, the result is if_false.
// For example:
var x = (obj ? obj.prop : null);
// The parentheses around the operator are not needed, but they make it easier to read.
```

Source: http://speakingjs.com/es5/ch09.html#_special_operators

Relational Operators: Instanceof

```
// The instanceof operator tests whether the prototype property of a constructor
// appears anywhere in the prototype chain of an object. The following code creates
// an object type Car and an instance of that object type, mycar. The instanceof
// operator demonstrates that the mycar object is of type Car and of type Object.
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
let mycar = new Car('Honda', 'Accord', 1998)
let a = mycar instanceof Car      // returns true
let b = mycar instanceof Object   // returns true
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Not an instanceof: To test if an object is not an instanceof a specific constructor, you can do
if (!(mycar instanceof Car)) {
  // Do something, like:
  // mycar = new Car(mycar)
}
// This is really different from:
if (!mycar instanceof Car) // This will always be false. (!mycar will be evaluated
// before instanceof, so you always try to know if a boolean is an instance of Car).
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof

Throw Errors

When an error occurs, JavaScript will normally stop and generate an error message. The technical term for this is: JavaScript will throw an exception (throw an error).

JavaScript will actually create an Error object with two properties: name and message.

The throw Statement:

The throw statement allows you to create a custom error.

Technically you can throw an exception (throw an error).

The exception can be a JavaScript String, a Number, a Boolean or an Object:

The console.error() method writes an error message to the console. The console is useful for testing purposes.

Sources: https://www.w3schools.com/js/js_errors.asp and https://www.w3schools.com/Jsref/met_console_error.asp

Errors

Property	Description
name	Sets or returns an error name
message	Sets or returns an error message (a string)
Error Name	**Description**
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

JavaScript has a built in error object that provides error information when an error occurs. The error object provides two useful properties: name and message. Six different values can be returned by the error name property.

Source: https://www.w3schools.com/js/js_errors.asp

Errors: Try Catch statement

The try statement consists of a try-block, which contains one or more statements. {} must always be used, even for single statements. At least one catch-block, or a finally-block, must be present. This gives us three forms for the try statement:

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

A catch-block contains statements that specify what to do if an exception is thrown in the try-block. If any statement within the try-block (or in a function called from within the try-block) throws an exception, control is immediately shifted to the catch-block.

If no exception is thrown in the try-block, the catch-block is skipped.

The finally-block will always execute after the try-block and catch-block(s) have finished executing. It always executes, regardless of whether an exception was thrown or caught.

You can nest one or more try statements. If an inner try statement does not have a catch-block, the enclosing try statement's catch-block is used instead.

You can also use the try statement to handle JavaScript exception.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

Errors: Try Catch statement

Nested try-blocks

Any given exception will be caught only once by the nearest enclosing catch-block unless it is rethrown.

Of course, any new exceptions raised in the "inner" block (because the code in catch-block may do something that throws), will be caught by the "outer" block.

Note! A constructor Error with new keyword is used on the line 3.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

```
> try {
    try {
      throw new Error('oops');
    }
    catch (ex) {
      console.error('inner', ex.message);
      throw ex;
    }
    finally {
      console.log('finally');
    }
  }
  catch (ex) {
    console.error('outer', ex.message);
  }
▶ inner oops
finally
▶ outer oops
```

Range and Reference Errors

```
> var num = 1;
try {
  num.toPrecision(500);   // A number cannot have 500 significant digits
}
catch(err) {
  console.error(err);
}
```

```
RangeError: toPrecision() argument must be between 1 and 100
    at Number.toPrecision (<anonymous>)
    at <anonymous>:3:7
```

```
undefined
```

```
> var x;
try {
  x = y + 1;   // y cannot be referenced (used)
}
catch(err) {
  console.error(err);
}
```

```
ReferenceError: y is not defined
    at <anonymous>:3:3
```

Source: https://www.w3schools.com/js/js_errors.asp

Syntax and Type errors

```
> try {
    eval("alert('Hello)");   // Missing ' will produce an error
  }
  catch(err) {
    console.error(err);
  }
▶ SyntaxError: Invalid or unexpected token
    at <anonymous>:2:5
undefined
> var num = 1;
try {
  num.toUpperCase();   // You cannot convert a number to upper case
}
catch(err) {
    console.error(err);
}
▶ TypeError: num.toUpperCase is not a function
    at <anonymous>:3:7
```

Source: https://www.w3schools.com/js/js_errors.asp

URI and Eval Errors

```
> try {
    decodeURI("%%%");   // You cannot URI decode percent signs
  }
  catch(err) {
    console.error(err);
  }
▶URIError: URI malformed
    at decodeURI (<anonymous>)
    at <anonymous>:2:5
```

Warning ! eval() is a dangerous function, which executes the code it's passed with the privileges of the caller. If you run eval() with a string that could be affected by a malicious party, you may end up running malicious code on the user's machine with the permissions of your webpage / extension.

The eval() function evaluates JavaScript code represented as a string. Executing JavaScript from a string is an enormous security risk. It is far too easy for a bad actor to run arbitrary code when you use eval(). More importantly, a third-party code can see the scope in which eval() was invoked, which can lead to possible attacks in ways to which the similar Function is not susceptible.

Sources: https://www.w3schools.com/js/js_errors.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

URI(Identifier): URL(Location) URN(Name)

URL - Uniform Resource Locator

Web browsers request pages from web servers by using a URL.

The URL is the address of a web page, like: https://www.w3schools.com.

URL Encoding (Percent Encoding)

URLs can only be sent over the Internet using the ASCII character-set.

Since URLs often contain characters outside the ASCII set, the URL has to be converted into a valid ASCII format.

URL encoding replaces unsafe ASCII characters with a "%" followed by two hexadecimal digits.

URLs cannot contain spaces. URL encoding normally replaces a space with a plus (+) sign or with %20.

Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is a string of characters which identifies an Internet Resource.

The most common URI is the Uniform Resource Locator (URL) which identifies an Internet domain address. Another, not so common type of URI is the Uniform Resource Name (URN).

URI	https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#examples
URL	https://en.wikipedia.org
URN	/wiki/Uniform_Resource_Identifier#examples

Sources:
https://www.w3schools.com/tags/ref_urlencode.ASP,
https://www.w3schools.com/xml/xml_namespaces.asp and
https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#examples

EncodeURIComponent() and DecodeURIComponent()

```javascript
// The encodeURIComponent() function encodes a URI component.
// This function encodes special characters. In addition,
// it encodes the following characters: , / ? : @ & = + $ #
var uri = "https://w3schools.com/my test.asp?name=ståle&car=saab";
var uri_enc = encodeURIComponent(uri);
console.log(uri_enc);
// logs https%3A%2F%2Fw3schools.com%2Fmy%20test.asp%3Fname%3Dst%C3%A5le%26car%3Dsaab
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The decodeURIComponent() function decodes a URI component.
var uri_dec = decodeURIComponent(uri_enc);
console.log(uri_dec);
// logs https://w3schools.com/my test.asp?name=ståle&car=saab
```

ASCII Encoding Reference : Your browser will encode input, according to the character-set used in your page. The default character-set in HTML5 is UTF-8. The table below includes three first encodings.

Character	From Windows-1252	From UTF-8
space	%20	%20
!	%21	%21
"	%22	%22

Sources: https://www.w3schools.com/jsref/jsref_encodeuricomponent.asp, https://www.w3schools.com/jsref/jsref_decodeuricomponent.asp and https://www.w3schools.com/tags/ref_urlencode.ASP

Javacript Engine

Engine: responsible for start-to-finish compilation and execution of our JavaScript program.

Compiler: handles all the dirty work of parsing and code-generation.

Scope Manager: collects and maintains a lookup list of all the declared variables/identifiers and enforces a set of rules as to how these are accessible to currently executing code.

var students = [..] has a declaration and initialization-assignment parts.

We typically think of that as a single statement, but that's not how Engine sees it. In fact, JS treats these as two distinct operations, one which Compiler will handle during compilation, and the other which Engine will handle during execution.

```
1  var students = [
2      { id: 14, name: "Kyle" },
3      { id: 73, name: "Suzy" },
4      { id: 112, name: "Frank" },
5      { id: 6, name: "Sarah" }
6  ];
7
8  function getStudentName(studentID) {
9      for (let student of students) {
10         if (student.id == studentID) {
11             return student.name;
12         }
13     }
14 }
15
16 var nextStudent = getStudentName(73);
17
18 console.log(nextStudent);
19 // "Suzy"
```

1 2 3

Source: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch2.md

Javascript Compiler

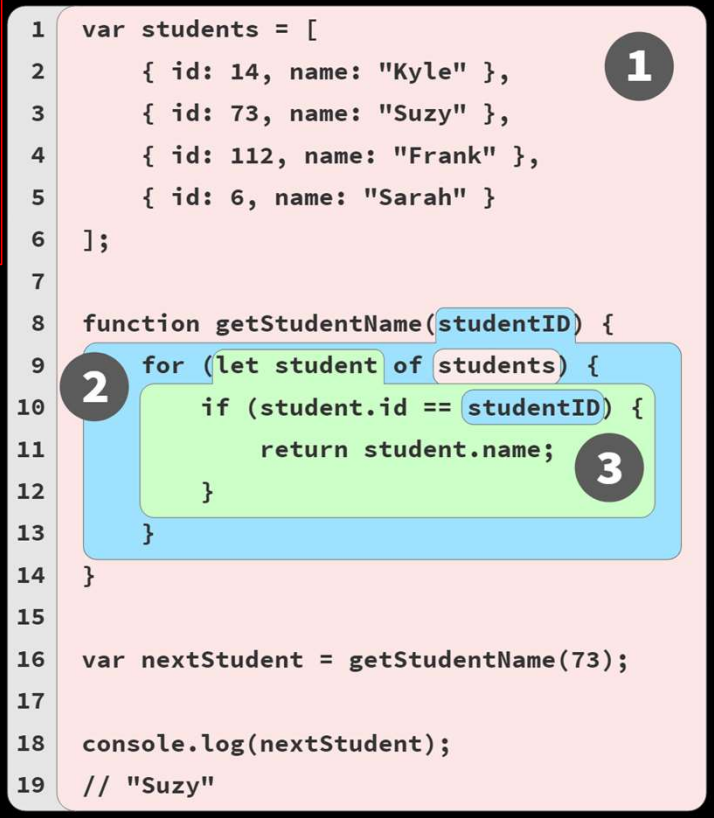
Processing of JS programs occurs in two phases: parsing/compilation first, then execution. (Code is first parsed [possibly to global memory/scope] and then executed [accessing global memory/scope if needed]).

Note! Hoisting is done during parsing phase.

Other than declarations, all occurrences of variables/identifiers in a program serve in one of two "roles": either they're the target of an assignment or they're the source of a value.

The terms "LHS" (aka, target) and "RHS" (aka, source) are used for these roles, respectively. As you might guess from the "L" and the "R", the acronyms mean "Left-Hand Side" and "Right-Hand Side", as in left and right sides of an = assignment operator.

However, assignment targets and sources don't always literally appear on the left or right of an = , so it's probably clearer to think in terms of target / source rather than left / right.



Source: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch1.md

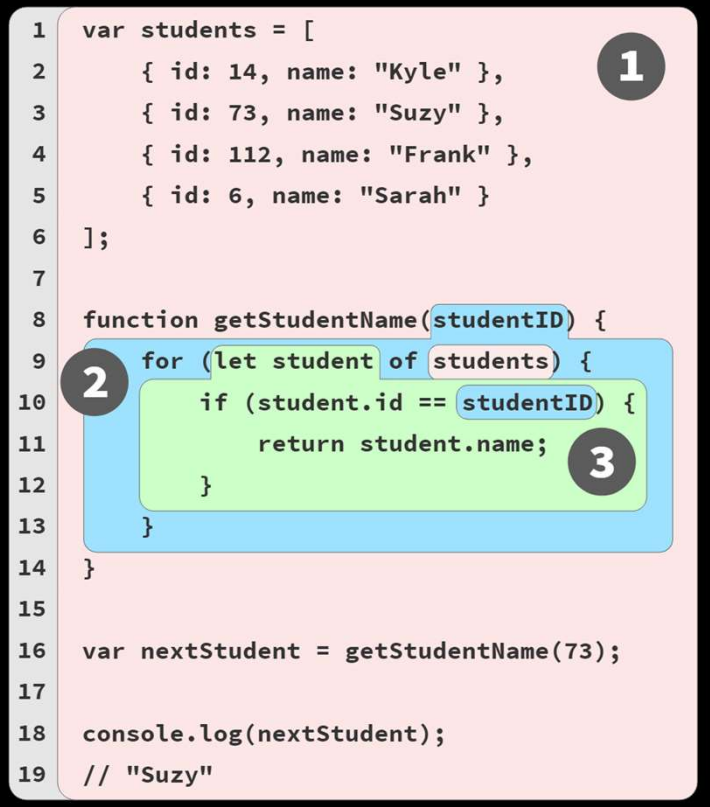
Javacript Scope Manager

Bubble 1 (RED) encompasses the global scope, which holds three identifiers/variables: students (line 1), getStudentName (line 8), and nextStudent (line 16).

Bubble 2 (BLUE) encompasses the scope of the function getStudentName(..) (line 8), which holds just one identifier/variable: the parameter studentID (line 8).

Bubble 3 (GREEN) encompasses the scope of the for-loop (line 9), which holds just one identifier/variable: student (line 9).

Scope bubbles are determined during compilation based on where the functions/blocks of scope are written, the nesting inside each other, and so on. Each scope bubble is entirely contained within its parent scope bubble. A scope is never partially in two different outer scopes.



Source: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch2.md

Lexical Scope

We've demonstrated that JS's scope is determined at compile time; the term for this kind of scope is "lexical scope". "Lexical" is associated with the "lexing" stage of compilation.

The key idea of "lexical scope" is that it's controlled entirely by the placement of functions, blocks, and variable declarations, in relation to one another.

If you place a variable declaration inside a function, the compiler handles this declaration as it's parsing the function and associates that declaration with the function's scope. If a variable is block-scope declared (let / const), then it's associated with the nearest enclosing { .. } block, rather than its enclosing function (as with var).

A reference (target or source role) for a variable must be resolved as coming from one of the scopes that are lexically available to it; otherwise the variable is said to be "undeclared".

If the variable is not declared in the current scope, the next outer/enclosing scope will be consulted.

This process of stepping out one level of scope nesting continues until either a matching variable declaration can be found, or the global scope is reached.

Compilation doesn't do anything in terms of reserving memory for scopes and variables. None of the program has been executed yet.

Compilation creates a map of all the lexical scopes that lays out what the program will need while it executes.

While scopes are identified during compilation, they're not actually created until runtime, each time a scope needs to run.

Source: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch1.md

Global Scope

The global scope is where JS exposes its built-ins:

primitives: undefined, null, Infinity, NaN

natives: Date(), Object(), String(), etc.

global functions: eval(), parseInt(), etc.

namespaces: Math, Atomics, JSON

friends of JS: Intl, WebAssembly

The environment hosting the JS engine exposes its own built-ins:

console (and its methods)

the DOM (window, document, etc)

timers (setTimeout(..), etc)

web platform APIs: navigator, history, geolocation, WebRTC, etc.

The global scope is the "outermost" scope – it has no outer scope. Its environment is the global environment. Every environment is connected with the global environment via a chain of environments that are linked by outer environment references.

Different JS environments handle the scopes of your programs, especially the global scope, differently.

In a programming environment, the global scope is the scope that contains, and is visible in, all other scopes.

In client-side JavaScript, the global scope is generally the web page inside which all the code is being executed.

Sources: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch4.md, https://exploringjs.com/deep-js/ch_global-scope.html and https://developer.mozilla.org/en-US/docs/Glossary/Global_scope

Variables

The names of variables, called identifiers, conform to certain rules.

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($). Subsequent characters can also be digits (0–9).

Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) as well as "a" through "z" (lowercase).

Some examples of legal names are Number_hits, temp99, $credit, and _name.

A variable declared using the var or let statement with no assigned value specified has the value of undefined.

An attempt to access an undeclared variable results in a ReferenceError exception being thrown.

```
var a;
console.log('The value of a is ' + a); // The value of a is undefined

console.log('The value of b is ' + b); // The value of b is undefined
var b;
// This one may puzzle you until you read 'Variable hoisting'
```

Source:
https://developer.mozilla.org/enUS/docs/Web/JavaScript/Guide/Grammar_and_Types#Variables

Initializing a Variable

```
1  var a;
2  console.log(a) //returns undefined
3  a = "foo";
4  console.log(a) //returns foo
```

Once you've declared a variable, you can initialize it with a value. You do this by typing the variable name, followed by an equals sign (=), followed by the value you want to give it.

In JavaScript, all code instructions should end with a semi-colon (;) . If you forget a semi-colon the ASI will add it.

A variable is a container for a value. But one special thing about variables is that their contained values can change.

Variables can also contain strings, numbers, complex data and even entire functions.

Variables aren't the values themselves; they are containers for values. Don't confuse a variable that exists but has not defined value with a variable that doesn't exist at all.

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Variables

JavaScript Primitive versus Reference Values

In JavaScript, a variable may store two types of values: primitive and reference.

JavaScript provides six primitive types as undefined, null, boolean, number, string, and symbol , and a reference type object.

The size of a primitive value is fixed, therefore, JavaScript stores the primitive value on the stack.

On the other hand, the size of a reference value is dynamic so JavaScript stores the reference value on the heap.

When you assign a value to a variable, the JavaScript engine will determine whether the value is a primitive or reference value.

Source: https://www.javascripttutorial.net/javascript-primitive-vs-reference-values/

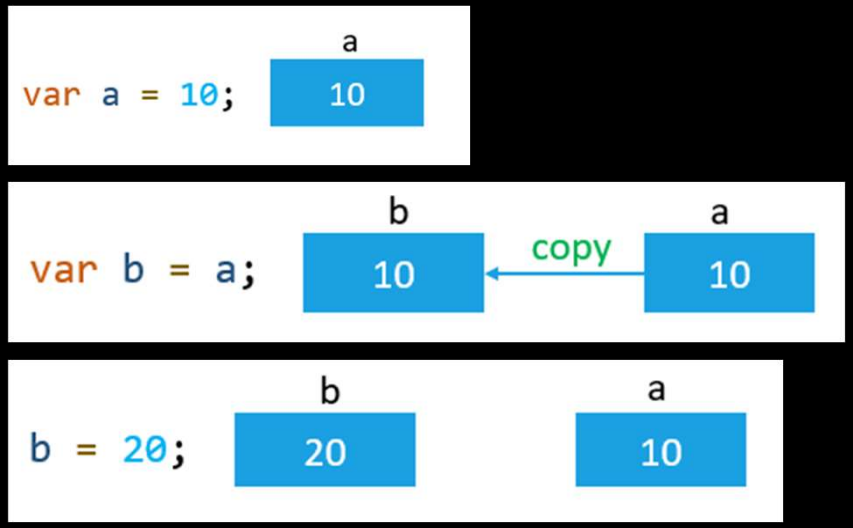
If the value is a primitive value, when you access the variable, you manipulate the actual value stored in that variable. In other words, the variable that stores a primitive value is accessed by value.

Unlike a primitive value, when you manipulate an object, you work on the reference of that object, rather than the actual object. It means a variable that stores an object is accessed by reference.

```
1  // To determine the type of a primitive value
2  // you use the typeof operator.
3  let x = 10;
4  console.log(typeof(x)); // number
5  // To find the type of a reference value,
6  // you use the instanceof operator:
7  let rgb = ['red','green','blue'];
8  console.log(rgb instanceof Array); // true
```

JavaScript Primitive versus Reference Values

Copying primitive values: When you assign a variable that stores a primitive value to another, the value stored in the variable is created and copied into the new variable.



Source: https://www.javascripttutorial.net/javascript-primitive-vs-reference-values/

```
// First, declare a variable a and
// initialize its value to 10.
var a = 10;
// Second, declare another variable b
// and assign it a value of the variable a.
// Internally, JavaScript engine copies the
// value stored in a into the location of b.
var b = a;
// Third, assign the variable b
// new value 20.
b = 20;
// Since a and b have no relationship,
// when you change the value stored
// in the b variable, the value of
// the a variable doesn't change.
console.log(a); // 10;
console.log(b); // 20;
```

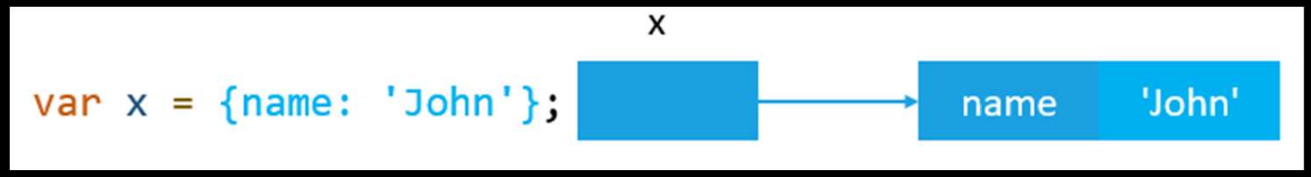
JavaScript Primitive versus Reference Values

Copying reference values:

When you assign a reference value from one variable to another, the value stored in the variable is also copied into the location of the new variable.

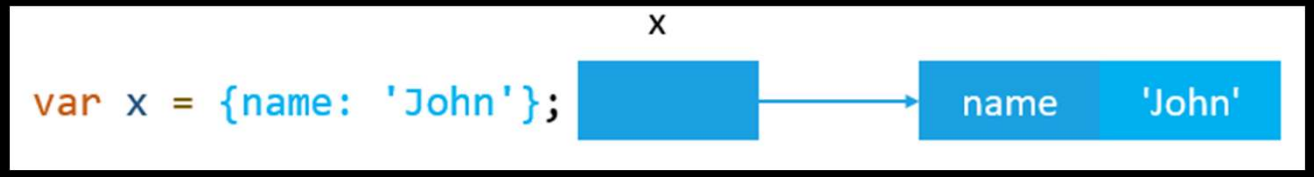
The difference is that the values stored in both variables now are the address of the actual object stored on the heap. As a result, both variables are referencing the same object.

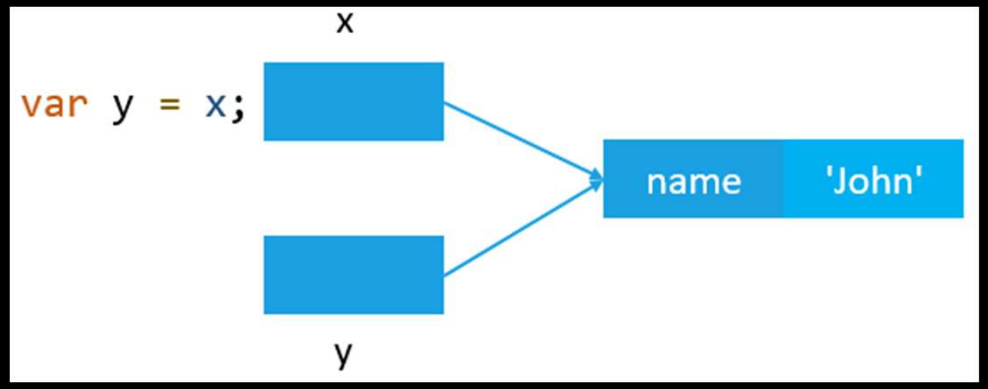
```
1   // First, declare a variable x that holds an
2   // object whose name property is 'John'.
3   var x = {name: 'John'};
4   // Second, declare another variable y and assign
5   // the x variable to y. Both x and y are now
6   // referencing the same object on the heap.
7   var y = x;
8   // Third, modify the value stored in the name
9   // property of the object using the y variable.
10  y.name = 'David';
11  // Since both x and y are referencing the same
12  // object, the change is also reflected if you
13  // access the object using the x variable.
14  console.log(x.name); // 'David'
```

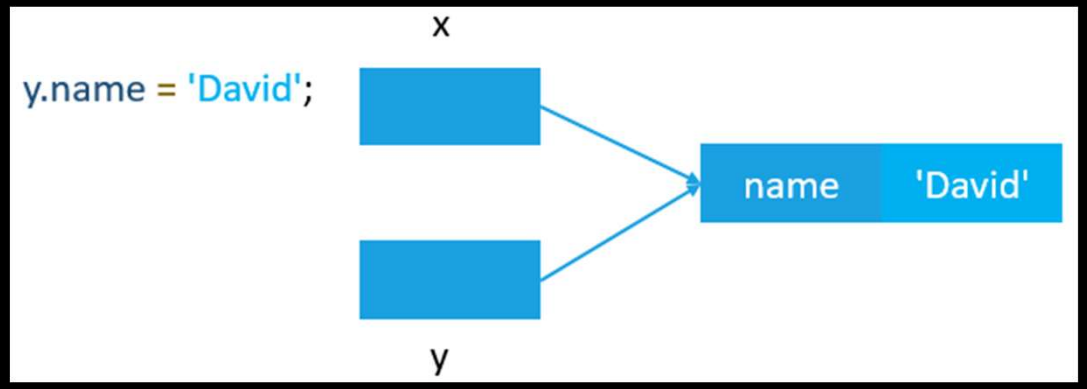


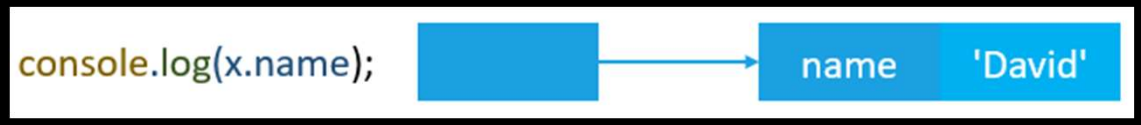
Source: https://www.javascripttutorial.net/javascript-primitive-vs-reference-values/

JavaScript Primitive versus Reference Values









Sources: https://www.javascripttutorial.net/javascript-primitive-vs-reference-values/ and https://stackoverflow.com/questions/6605640/javascript-by-reference-vs-by-value

Javascript is always pass by value, but when a variable refers to an object (including arrays), the "value" is a reference to the object. Changing the value of a variable never changes the underlying primitive or object, it just points the variable to a new primitive or object. However, changing a property of an object referenced by a variable does change the underlying object.

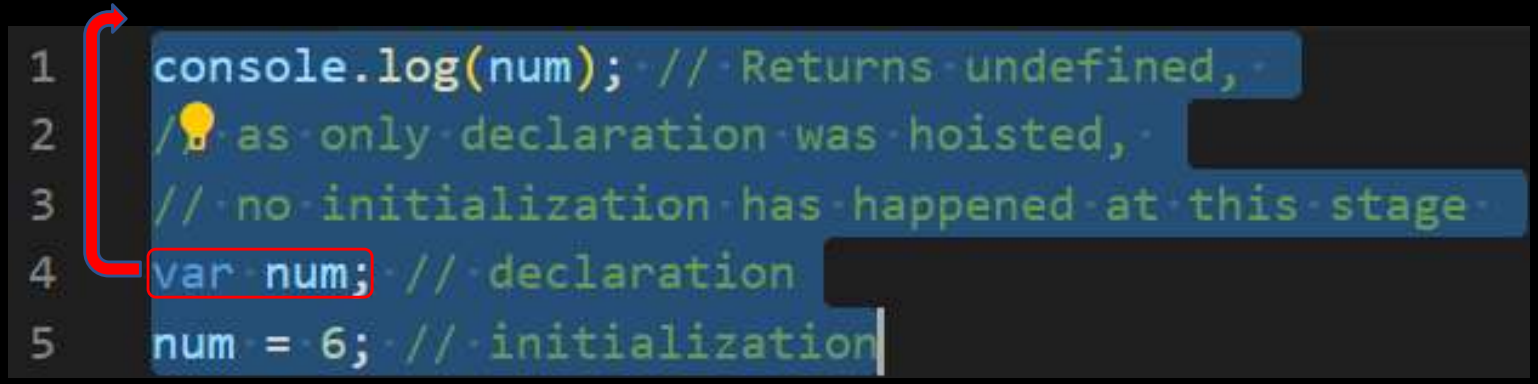
Hoisting

```
1   // Hoisting is JavaScript's default behavior of moving declarations to the top.
2   // In JavaScript, a variable can be declared after it has been used.
3   // In other words; a variable can be used before it has been declared.
4   x = 5; // Assign 5 to x
5   console.log(x); // logs 5
6   var x; // Declare x
7   // The first example gives the same result as the second example:
8   var x; // Declare x
9   console.log(x); // logs 5
10  x = 5; // Assign 5 to x
11  console.log(x);
12  // Hoisting is JavaScript's default behavior of moving all declarations to the top
13  // of the current scope (to the top of the current script or the current function).
14  // Variables and constants declared with let or const are not hoisted!
15  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
16  // JavaScript only hoists declarations, not initializations.
17  var x = 5; // Initialize x
18  var foo = x + " " + y; // Displays x and y
19  console.log(foo); // logs 5 undefined
20  var y = 7; // Initialize y
```

Source: https://www.w3schools.com/js/js_hoisting.asp

Variable Hoisting

The variable and function declarations are put into memory during the compile phase, but stay exactly where you typed them in your code. JavaScript only hoists declarations, not initializations.



```
1  console.log(num); // Returns undefined,
2  // as only declaration was hoisted,
3  // no initialization has happened at this stage
4  var num; // declaration
5  num = 6; // initialization
```

```
1  // Example below only has initialization.
2  // No hoisting happens so trying to read the variable
3  // results in ReferenceError: Num is not defined
4  console.log(num); // throws ReferenceError exception
5  num = 6; // initialization
```

Source: https://developer.mozilla.org/en-US/docs/Glossary/Hoisting

Using Variables : Var, Let and Const

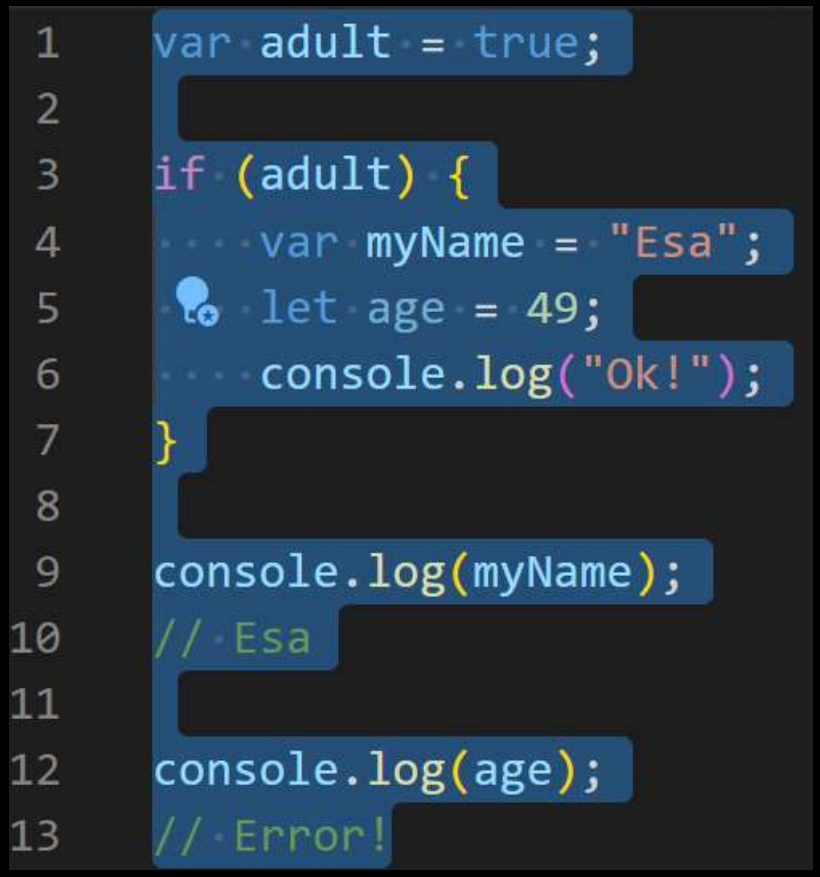
Values can be held in variables; think of variables as just containers for values. Variables have to be declared (created) to be used.

The let keyword has some differences to var, with the most obvious being that let allows a more limited access to the variable than var. This is called "block scoping" as opposed to regular or function scoping.

The attempt to access age outside of the if statement results in an error, because age was “block-scoped” to the if, whereas myName was not.

A third declaration form is const. It's like let but has an additional limitation that it must be given a value at the moment it's declared, and const cannot be re-assigned a different value later.

Source: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch2.md



```
var adult = true;

if (adult) {
    var myName = "Esa";
    let age = 49;
    console.log("Ok!");
}

console.log(myName);
// Esa

console.log(age);
// Error!
```

The difference between Var and Let

Back when JavaScript was first created, there was only var.

Let and const were created in modern versions of JavaScript.

If you write a multiline JavaScript program that declares and initializes a variable, you can actually declare a variable with var after you initialize it and it will still work.

This works because of hoisting.

Hoisting no longer works with let. If we changed var to let in the example, it would fail with an error. This is a good thing.

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Variables

The difference between Var, Let and Const

When you use var, you can declare the same variable as many times as you like (lines 1 and 2)

With let you can not declare the value again, but you can give it a new value (lines 9 and 10)

Const works in exactly the same way as let, except that you can not give a const a new value (lines 13 and 14)

For these reasons and more, it is recommended that you use let and const as much as possible in your code, rather than var.

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Variables

Statements

A computer program is a list of "instructions" to be "executed" by a computer.

In a programming language, these programming instructions are called statements.

A JavaScript program is a list of programming statements.

The statements are executed, one by one (synchronously), in the same order as they are written.

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

Source:
https://www.w3schools.com/js/js_statements.asp

Semicolons separate JavaScript statements.

Add a semicolon at the end of each executable statement (not required, but highly recommended).

JavaScript ignores multiple spaces. You can add white space to your script to make it more readable.

A good practice is to put spaces around operators (= + - * /).

JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.

The purpose of code blocks is to define statements to be executed together.

JavaScript keywords are reserved words. Reserved words cannot be used as names for variables or functions.

Variables, Literals, Expressions, Identifiers and Keywords

The JavaScript syntax defines two types of values: fixed values and variable values.

Fixed values are called **literals**.

Variable values are called **variables**.

An expression is a combination of values, variables, and operators, which computes to a value.

The computation is called an evaluation.

The values can be of various types, such as numbers and strings.

Source:
https://www.w3schools.com/js/js_syntax.asp

Identifiers are names. Identifiers are used to name variables (keywords, functions, and labels).

Any JavaScript identifier that is not a reserved keyword is a label.

JavaScript keywords are used to identify actions to be performed. (var, let and const are among reserved keywords)

Lower Camel Case:

JavaScript programmers tend to use camel case that starts with a lowercase letter:

firstName, lastName, masterCard, interCity.

Variables, Literals, Expressions, Identifiers and Keywords

```
var x, y, z;       // How to declare variables
x = 5; y = 6;      // How to assign values
z = x + y;         // How to compute values
// Numbers are written with or without decimals
10.50, 1001
// Strings are text, written within double or single quotes
"Esa", 'Esa'
// JavaScript uses arithmetic operators ( + - * / )
// to compute values
(5 + 6) * 10
// 5 * 10 evaluates to 50
5 * 10
// JavaScript uses an assignment operator ( = )
// to assign values to variables
// The var keyword tells the browser to create variables
var x, y;
x = 5 + 6;
y = x * 10;
```

Source: https://www.w3schools.com/js/js_syntax.asp

Data Types

In JavaScript there are 5 different data types that can contain values:

- string
- number
- boolean
- object
- function

There are 2 data types that cannot contain values:

- null
- undefined

There are 6 types of objects:

- Object
- Date
- Array
- String
- Number
- Boolean

Source: https://www.w3schools.com/js/js_type_conversion.asp

Value Type Determination

Unit of information in a program is a value. Values are data. They're how the program maintains state.

Values come in two forms in Javascript:

primitive or object

Functions, like arrays, are a special kind (aka, sub-type) of object.

Primitive values and object values behave differently when they're assigned or passed around.

For distinguishing values, the typeof operator tells you its built-in type, if primitive, or "object".

Source: https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch2.md

```
typeof 42; // "number"
typeof "abc"; // "string"
typeof true; // "boolean"
typeof undefined; // "undefined"
typeof null; // "object" -- oops, bug!
typeof { "a": 1 }; // "object"
typeof [1,2,3]; // "object"
typeof function hello(){}; // "function"
```

WARNING !

Typeof null unfortunately returns "object" instead of the expected "null".

Typeof returns the specific "function" for functions, but not the expected "array" for arrays.

Primitive Data Types

A primitive (primitive value, primitive data type) is data that is not an object and has no methods.

There are 6 primitive data types:

- string
- number
- bigint
- boolean
- undefined
- symbol (new in ECMAScript 2015)

There also is null, which is seemingly primitive, but indeed is a special case for every Object.

JavaScript has dynamic types. This means that the same variable can be used to hold different data types.

It is important not to confuse a primitive itself with a variable assigned a primitive value.

All primitives are immutable and cannot be directly altered. Instead of working primitives directly, we're working on a replaced copy, without affecting the original.

The variable may be reassigned a new value, but the existing value can not be changed in the ways that objects, arrays, and functions can be altered.

Source: https://developer.mozilla.org/en-US/docs/Glossary/Primitive

Eight Data Types

There is seven data types that are primitives and an Object:

1) Boolean True and false.
2) Undefined A top-level property whose value is not defined.
3) Number An integer or floating point number. For example: 42 (integer) or 3.14159 (floating point).
4) BigInt An integer with arbitrary precision. For example: 9007199254740992n.
5) String A sequence of characters that represent a text value. For example: "Howdy"
6) Symbol (new in ECMAScript 2015) A data type whose instances are unique and immutable.
7) Null A special keyword denoting a null value.
8) Object including:

Object
Date
Array
String
Number
Boolean

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types

Undefined and Null

```
// Most programming languages have values denoting missing information.
// JavaScript has two such "nonvalues," undefined and null:
// undefined means "no value." Uninitialized variables are undefined:
var foo;
foo
undefined
// Missing parameters are undefined:
function f(x) { return x }
f()
undefined
// If you read a nonexistent property, you get undefined:
var obj = {}; // empty object
obj.foo
undefined
// null means "no object." It is used as a nonvalue whenever an object
// is expected (parameters, last in a chain of objects, etc.).
// Warning: undefined and null have no properties,
// not even standard methods such as toString().
```

Source: http://speakingjs.com/es5/ch01.html

Strings

```
// Normally, JavaScript strings are primitive values,
// created from literals:
var x = "John";
typeof(x) // typeof x will return string
// But strings can also be defined as objects
// with the keyword new:
var y = new String("John");
typeof(y) // typeof y will return object
```

Strings are useful for holding data that can be represented in text form. A JavaScript string is zero or more characters written inside quotes. String Methods and String Properties: Primitive values, like "John Doe", cannot have properties or methods (because they are not objects).

But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties. Strings can also be defined as objects with the keyword new. WARNING ! Don't create strings as objects. It slows down execution speed.

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String and https://www.w3schools.com/js/js_strings.asp

String Methods and Properties

```
1  // A string is converted to upper case with toUpperCase():
2  var text1 = "Hello World!";       // String
3  var text2 = text1.toUpperCase();  // text2 is text1 converted to upper
4  // A string is converted to lower case with toLowerCase():
5  var text1 = "Hello World!";       // String
6  var text2 = text1.toLowerCase();  // text2 is text1 converted to lower
7  // concat() joins two or more strings:
8  var text = "Hello" + " " + "World!";
9  var text = "Hello".concat(" ", "World!");
10 // The trim() method removes whitespace from both sides of a string:
11 var str = "      Hello World!      ";
12 alert(str.trim());
13 // The length property returns the length of a string:
14 var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
15 var sln = txt.length;
```

All string methods return a new string. They don't modify the original string. Formally said: strings are immutable: strings cannot be changed, only replaced.

Source: https://www.w3schools.com/js/js_string_methods.asp

Converting a String to an Array

```
// A string can be converted to an array with the split() method:
var greetings = "Hello, Hi, Hola"; // String
greetings.split(","); // Split on commas // logs ["Hello, Hi, Hola"]
greetings.split(" "); // Split on spaces // logs ["Hello,", "Hi,", "Hola"]
greetings.split("|"); // Split on pipe // logs ["Hello, Hi, Hola"] (Oops!)
// If the separator is omitted, the returned array
// will contain the whole string in index [0].
var y = greetings.split()
console.log(y)
// logs ["Hello, Hi, Hola"]
console.log(y[0]); // logs Hello, Hi, Hola
console.log(y[1]); // logs undefined
// If the separator is "", the returned array
// will be an array of single characters:
var x = greetings.split("")
console.log(x)
// logs ["H", "e", "l", "l", "o", ",", " ", "H", "i", ",", " ", "H", "o", "l", "a"]
```

Source: https://www.w3schools.com/js/js_string_methods.asp

Template Literals (Template Strings)

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

They were called "template strings" in prior editions of the ES2015 specification.

Template literals are enclosed by the backtick (` `) (grave accent) character instead of double or single quotes.

Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}). The expressions in the placeholders and the text between the backticks (` `) get passed to a function.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

The default function just concatenates the parts into a single string. If there is an expression preceding the template literal (tag here), this is called a tagged template. In that case, the tag expression (usually a function) gets called with the template literal, which you can then manipulate before outputting.

```
1  // Syntax:
2  // `string text`
3  // `string text line 1
4  //  string text line 2`
5  // `string text ${expression} string text`
6  // tag`string text ${expression} string text`
7  // To escape a backtick in a template literal,
8  // put a backslash (\) before the backtick.
9  `\`` === '`' // --> true
```

Template Literals (Template Strings) 2

```
1  // Multi-line strings: Any newline characters
2  // inserted in the source are part of the
3  // template literal.
4  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
5  // Using normal strings, you would have to use
6  // the following syntax in order to get
7  // multi-line strings:
8  console.log('string text line 1\n' +
9  'string text line 2');
10 // "string text line 1
11 // string text line 2"
12 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13 // Using template literals, you can do
14 // the same like this:
15 console.log(`string text line 1
16 string text line 2`);
17 // "string text line 1
18 // string text line 2"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Template Literals (Template Strings) 3

```
1   // Expression interpolation: In order to embed expressions within normal
2   // strings, you would use the following syntax:
3   let a = 5;
4   let b = 10;
5   console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
6   // "Fifteen is 15 and
7   // not 20."
8   // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
9   // Now, with template literals, you are able to make use of the syntactic
10  // sugar, making substitutions like this more readable:
11  let a = 5;
12  let b = 10;
13  console.log(`Fifteen is ${a + b} and
14  not ${2 * a + b}.`);
15  // "Fifteen is 15 and
16  // not 20."
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Template Literals (Template Strings) 4

```
// Tagged templates: A more advanced form of template literals are tagged templates. Tags allow you
// to parse template literals with a function. The first argument of a tag function contains an
// array of string values. The remaining arguments are related to the expressions.
let person = 'Mike';
let age = 28;
function myTag(strings, personExp, ageExp) {
  let str0 = strings[0]; // "That "
  let str1 = strings[1]; // " is a "
// There is technically a string after the final expression (in our example),
// but it is empty (""), so disregard. let str2 = strings[2];
  let ageStr;
  if (ageExp > 99){
    ageStr = 'centenarian';
  } else {
    ageStr = 'youngster';
  }
// We can even return a string built using a template literal
  return `${str0}${personExp}${str1}${ageStr}`;
}
let output = myTag`That ${ person } is a ${ age }`;
console.log(output); // That Mike is a youngster
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Template Literals (Template Strings) 5

```
1  // Raw strings: The special raw property, available on the first argument
2  // to the tag function, allows you to access the raw strings as they were
3  // entered, without processing escape sequences.
4  function tag(strings) {
5    console.log(strings.raw[0]);
6  }
7  tag`string text line 1 \n string text line 2`;
8  // logs "string text line 1 \n string text line 2",
9  // including the two characters '\' and 'n'
10 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11 // In addition, the String.raw() method exists to create raw strings—
12 // just like the default template
13 // function and string concatenation would create.
14 let str = String.raw`Hi\n${2+3}!`;
15 // "Hi\n5!"
16 str.length;
17 // 6
18 str.split('').join(',');
19 // "H,i,\,n,5,!"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Numbers

```
// JavaScript has only one type of number.
// Numbers can be written with or without decimals.
var x = 3.14;    // A number with decimals
var y = 3;       // A number without decimals
// Extra large or extra small numbers can be written
// with scientific (exponent) notation:
var x = 123e5;    // 12300000
var y = 123e-5;   // 0.00123
// Integers (numbers without a period or exponent notation)
// are accurate up to 15 digits.
var x = 999999999999999;   // x will be 999999999999999
var y = 9999999999999999;  // y will be 10000000000000000
// The maximum number of decimals is 17,
// but floating point arithmetic is not always 100% accurate:
var x = 0.2 + 0.1;         // x will be 0.30000000000000004
```

JavaScript numbers are always 64-bit floating point. JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

Source: https://www.w3schools.com/js/js_numbers.asp

Base Systems of Numbers

```
// By default, JavaScript displays numbers as base 10 decimals.
// But you can use the toString() method to output numbers from base 2 to base 36.
// Hexadecimal is base 16. Decimal is base 10. Octal is base 8. Binary is base 2.
// There is base 36 as well.
var myNumber = 32;
" Decimal " + myNumber.toString(10); // "Decimal 32"
" Hexadecimal " + myNumber.toString(16); // "Hexadecimal 20"
" Octal " + myNumber.toString(8); // "Octal 40"
" Binary " + myNumber.toString(2); // "Binary 100000"
// JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x
var x = 0xFF;        // x will be 255
// NaN is a JavaScript reserved word indicating that a number is not a legal number.
// Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):
var x = 100 / "Apple";  // x will be NaN (Not a Number)
```

WARNING ! Never write a number with a leading zero (like 07). Some JavaScript versions interpret numbers as octal if they are written with a leading zero. Octal number syntax uses a leading zero.

Source: https://www.w3schools.com/js/js_numbers.asp

Numbers: NaN

The global NaN property is a value representing Not-A-Number. NaN is a property of the global object.

The initial value of NaN is Not-A-Number — the same as the value of Number.NaN. In modern browsers, NaN is a non-configurable, non-writable property. Even when this is not the case, avoid overriding it.

It is rather rare to use NaN in a program. It is the returned value when Math functions fail (Math.sqrt(-1)) or when a function trying to parse a number fails (parseInt("blabla")).

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN

```
1  // Standard built-in objects - NaN
2  function sanitise(x) {
3      if (isNaN(x)) {
4          return NaN;
5      }
6      return x;
7  }
8
9  console.log(sanitise('1'));
10 // expected output: "1"
11
12 console.log(sanitise('NotANumber'));
13 // expected output: NaN
```

Numbers: NaN

```
// NaN compares unequal (via ==, !=, ===, and !==)
// to any other value -- including to another NaN value.
// Use Number.isNaN() or isNaN() to most clearly determine
// whether a value is NaN. Or perform a self-comparison:
// NaN, and only NaN, will compare unequal to itself.
NaN === NaN;        // false
Number.NaN === NaN; // false
isNaN(NaN);         // true
isNaN(Number.NaN);  // true
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function valueIsNaN(v) { return v !== v; }
valueIsNaN(1);         // false
valueIsNaN(NaN);       // true
valueIsNaN(Number.NaN); // true
// However, do note the difference between isNaN() and Number.isNaN():
// the former will return true if the value is currently NaN, or if
// it is going to be NaN after it is coerced to a number,
// while the latter will return true only if the value is currently NaN:
isNaN('hello world');       // true
Number.isNaN('hello world'); // false
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN

Regular Expressions

A regular expression is a sequence of characters that forms a search pattern. The search pattern can be used for text search and text replace operations. Syntax: /pattern/modifiers;

```
1  var patt = /w3schools/i;
2  // /w3schools/i is a regular expression.
3  // w3schools is a pattern (to be used in a search).
4  // i is a modifier (modifies the search to be case-insensitive).
```

In JavaScript, regular expressions are also objects.

These patterns are used with the exec() and test() methods of RegExp, and with the match(), matchAll(), replace(), search(), and split() methods of String.

Parentheses around any part of the regular expression pattern "/d(b+)d/g "causes that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use.

Sources: https://www.w3schools.com/js/js_regexp.asp and https://developer.mozilla.org/enUS/docs/Web/JavaScript/Guide/Regular_Expressions

Regular Expressions 2

Methods that use regular expressions:

Method	Description
exec()	Executes a search for a match in a string. It returns an array of information or null on a mismatch.
test()	Tests for a match in a string. It returns true or false.
match()	Returns an array containing all of the matches, including capturing groups, or null if no match is found.
matchAll()	Returns an iterator containing all of the matches, including capturing groups.
search()	Tests for a match in a string. It returns the index of the match, or -1 if the search fails.
replace()	Executes a search for a match in a string and replaces the matched substring with a replacement substring.
split()	Uses a regular expression or a fixed string to break a string into an array of substrings.

Source: https://www.w3schools.com/js/js_regexp.asp

Regular Expressions 3

Special characters in regular expressions and regular expression flags:

Characters / constructs	Corresponding article
\, ., \cX, \d, \D, \f, \n, \r, \s, \S, \t, \v, \w, \W, \o, \xhh, \uhhhh, \uhhhhh, [\b]	*Character classes*
^, $, x(?=y), x(?!y), (?<=y)x, (?<!y)x, \b, \B	*Assertions*
(x), (?:x), x|y, [xyz], [^xyz], *Number*	*Groups and ranges*
*, +, ?, x{*n*}, x{*n*,}, x{*n*,*m*}	*Quantifiers*
\p{*UnicodeProperty*}, \P{*UnicodeProperty*}	*Unicode property escapes*

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, you can include special characters in the pattern. For example, to match a single "a" followed by zero or more "b"s followed by "c", you'd use the pattern /ab*c/: the * after "b" means "0 or more occurrences of the preceding item." In the string "cbbabbbbcdebc", this pattern will match the substring "abbbbc". If you need to use any of the special characters literally (actually searching for a "*", for instance), you must escape it by putting a backslash in front of it.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Regular Expressions 4

Flag	Description	Corresponding property
g	Global search.	RegExp.prototype.global
i	Case-insensitive search.	RegExp.prototype.ignoreCase
m	Multi-line search.	RegExp.prototype.multiline
s	Allows . to match newline characters. (Added in ES2018, not yet supported in Firefox).	RegExp.prototype.dotAll
u	"unicode"; treat a pattern as a sequence of unicode code points.	RegExp.prototype.unicode
y	Perform a "sticky" search that matches starting at the current position in the target string. See sticky.	RegExp.prototype.sticky

Regular expressions have six optional flags that allow for functionality like global and case insensitive searching. These flags can be used separately or together in any order and are included as part of the regular expression. To include a flag with the regular expression, use this syntax: `var re = /pattern/flags;` or `var re = new RegExp('pattern', 'flags');` Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Booleans

A JavaScript Boolean represents one of two values: true or false.

Very often, in programming, you will need a data type that can only have one of two values, like

- YES / NO
- ON / OFF
- TRUE / FALSE

Boolean conditionals true and false are often used to decide which sections of code to execute (such as in if statements) or repeat (such as in for loops).

The Boolean value is named after English mathematician George Boole, who pioneered the field of mathematical logic.

```
1  // You can use the Boolean() function
2  // to find out
3  // if an expression (or a variable)
4  // is true or false:
5  Boolean(10 > 9) // returns true
6  (10 > 9)        // also returns true
7  10 > 9          // also returns true
8  9 > 10          // returns false
```

The Boolean value of an expression is the basis for all JavaScript comparisons and conditions.

Sources:
https://www.w3schools.com/js/js_booleans.asp and
https://developer.mozilla.org/en-US/docs/Glossary/Boolean

Booleans

```
1   // Everything With a "Value" is True
2   var b1 = Boolean(100);
3   var b2 = Boolean(3.14);
4   var b3 = Boolean(-15);
5   var b4 = Boolean("Hello");
6   var b5 = Boolean('false');
7   var b6 = Boolean(1 + 7 + 3.14);
8   // Everything Without a "Value" is False
9   var b7 = Boolean(0); // The Boolean value of 0 (zero) is false
10  var b8 = Boolean(-0); // The Boolean value of -0 (minus zero) is false
11  var b9 = Boolean(""); // The Boolean value of "" (empty string) is false
12  var b10 = Boolean(x); // The Boolean value of undefined is false
13  var b11 = Boolean(null); // The Boolean value of null is false
14  var b12 = Boolean(false); // The Boolean value of false is (you guessed it) false
15  var x = 10 / "H";
16  var b13 = Boolean(x); // The Boolean value of NaN is false
```

Source: https://www.w3schools.com/js/js_booleans.asp

Booleans: Truthy and Falsy

Whenever JavaScript expects a boolean value (e.g., for the condition of an if statement), any value can be used. It will be interpreted as either true or false. The following values are interpreted as false:

- undefined, null
- Boolean: false
- Number: 0, NaN
- String: ''

All other values (including all objects!) are considered true. Values interpreted as false are called falsy, and values interpreted as true are called truthy. Boolean(), called as a function, converts its parameter to a boolean.

The primitive boolean type comprises the values true and false. The following operators produce booleans:

- Binary logical operators: && (And), || (Or)
- Prefix logical operator: ! (Not)
- Comparison operators:
- Equality operators: ===, !==, ==, !=
- Ordering operators (for strings and numbers): >, >=, <, <=

Sources: http://speakingjs.com/es5/ch01.html#basic_truthy_falsy and http://speakingjs.com/es5/ch01.html

Symbol

Symbols are guaranteed to be unique. Even if we create many symbols with the same description, they are diffe rent values. A symbol may be used as an object property.

```
// Here are two symbols with the same description:
let Sym1 = Symbol("Sym")
let Sym2 = Symbol("Sym")
console.log(Sym1 === Sym2) // returns "false"
// Symbols don't "Auto-Convert" to strings
let Sym = Symbol("Sym")
alert(Sym) // TypeError: Cannot convert a Symbol value to a string
// If you really want to show a symbol, we need to call .toString() on it.
let Sym = Symbol("Sym")
alert(Sym.toString()) // Symbol(Sym), now it works
// Or you can use the symbol.description property to get its description:
let _Sym = Symbol("Sym");
alert(_Sym.description); // Sym
```

Source: https://developer.mozilla.org/en-US/docs/Glossary/Symbol

Symbol 2

```
// Symbols are tokens that serve as unique IDs. You create symbols via the factory function
// Symbol() (which is loosely similar to String returning strings if called as a function):
const symbol1 = Symbol();
console.log(symbol1); // Symbol()
// Symbol() has an optional string-valued parameter that lets you give the newly created
// Symbol a description. That description is used when the symbol is converted to
// a string (via toString() or String()):
const symbol2 = Symbol('symbol2');
String(symbol2); // 'Symbol(symbol2)'
// Every symbol returned by Symbol() is unique, every symbol has its own identity:
Symbol() === Symbol(); // false
// You can see that symbols are primitive if you apply the typeof operator to one of them –
// it will return a new symbol-specific result:
typeof Symbol(); // 'symbol'
// The function Symbol: Symbol(description?) : symbol
// Creates a new symbol. The optional parameter description allows you to give the symbol
// a description. The only way to access the description is to convert the symbol to a string
// (via toString() or String()). The result of such a conversion is 'Symbol('+description+')':
const sym = Symbol('hello');
String(sym); //'Symbol(hello)'
// Symbol is can't be used as a constructor – an exception is thrown if you invoke it via new.
```

Source: https://exploringjs.com/es6/ch_symbols.html#sec_overview-symbols

The Typeof Operator

```
typeof "John"                 // Returns "string"
typeof 3.14                   // Returns "number"
typeof NaN                    // Returns "number"
typeof false                  // Returns "boolean"
typeof [1,2,3,4]              // Returns "object"
typeof {name:'John', age:34}  // Returns "object"
typeof new Date()             // Returns "object"
typeof function () {}         // Returns "function"
typeof myCar                  // Returns "undefined" *
typeof null                   // Returns "object"
```

You can use the typeof operator to find the data type of a JavaScript variable. You cannot use typeof to determine if a JavaScript object is an array (or a date). Typeof operator always returns a string (containing the type of the operand).

The data type of an undefined variable is undefined. The data type of a variable that has not been assigned a value is also undefined.

Source: https://www.w3schools.com/js/js_type_conversion.asp

Type Conversion

```
5 + null    // returns 5         because null is converted to 0
"5" + null  // returns "5null"   because null is converted to "null"
"5" + 2     // returns "52"      because 2 is converted to "2"
"5" - 2     // returns 3         because "5" is converted to 5
"5" * "2"   // returns 10        because "5" and "2" are converted to 5 and 2
```

JavaScript variables can be converted to a new variable and another data type.

By the use of a JavaScript function. (Implicit or explicit conversion). Number() converts to a Number, String() converts to a String, Boolean() converts to a Boolean.

Automatically by JavaScript itself. (Implicit coercion)

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" data type.

The result is not always what you expect:

Source: https://www.w3schools.com/js/js_type_conversion.asp

Type Conversion versus Type Coersion

```
1  "5"*"5" // returns 25 because "5"
2  // is automatically or implicitly coersed to 5
3  // due to the use of arithmetic operator (*)
4  Boolean(5) // returns explicitly converted true
5  // due to the use of Boolean() function
```

Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers).

Type conversion is similar to type coercion because they both convert values from one data type to another with one key difference: type coercion is implicit whereas type conversion can be either implicit or explicit.

Sources: https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion and https://www.w3schools.com/js/js_type_conversion.asp

Type Conversion

Original Value	Converted to Number	Converted to String	Converted to Boolean
false	0	"false"	false
true	1	"true"	true
0	0	"0"	false
1	1	"1"	true
"0"	0	"0"	**true**
"000"	0	"000"	**true**
"1"	1	"1"	true
NaN	NaN	"NaN"	false
Infinity	Infinity	"Infinity"	true
-Infinity	-Infinity	"-Infinity"	true
""	**0**	""	**false**

Original Value	Converted to Number	Converted to String	Converted to Boolean
"20"	20	"20"	true
"twenty"	NaN	"twenty"	true
[ ]	**0**	""	true
[20]	**20**	"20"	true
[10,20]	NaN	"10,20"	true
["twenty"]	NaN	"twenty"	true
["ten","twenty"]	NaN	"ten,twenty"	true
function(){}	NaN	"function(){}"	true
{ }	NaN	"[object Object]"	true
null	**0**	"null"	false
undefined	NaN	"undefined"	false

Note! Red values indicate values with unexpected results.

Source: https://www.w3schools.com/js/js_type_conversion.asp

Automatic String Conversion

```
1  // Automatic String Conversion:JavaScript automatically calls the variable's
2  // toString() function when you try to "output" an object or a variable:
3  myVar = {name:"Fjohn"}  // toString converts to "[object Object]"
4  myVar1 = [1,2,3,4]      // toString converts to "1,2,3,4"
5  myVar2 = new Date()     // toString converts to "Sun Apr 26 2020 11:54:59 GMT+0300
6  // (Eastern European Summer Time)"
7  // Numbers and booleans are also converted, but this is not very visible:
8  myVar3 = 123            // toString converts to "123"
9  myVar4 = true           // toString converts to "true"
10 myVar5 = false          // toString converts to "false"
11 document.write(myVar + " ",myVar1 + " ",myVar2 + " ",myVar3 + " ",myVar4 + " ",myVar5 + " ");
```

[object Object] 1,2,3,4 Sun Apr 26 2020 11:54:59 GMT+0300 (Eastern European Summer Time) 123 true false

JavaScript type conversion: JavaScript variables can be converted to a new variable and another data type: By the use of a JavaScript function or automatically by JavaScript itself.

Source: https://www.w3schools.com/js/js_type_conversion.asp

Strict Mode

JavaScript was designed to be easy for novice developers and that is why default mode, non-strict mode is referred to as "sloppy mode". This isn't an official term.

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode).

Declared inside a function, it has local scope (only the code inside the function is in strict mode).

Strict mode makes several changes to normal JavaScript semantics.

Sources: https://www.w3schools.com/js/js_strict.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Eliminates some JavaScript silent errors by changing them to throw errors.

Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.

Prohibits some syntax likely to be defined in future versions of ECMAScript.

Strict mode code and non-strict mode code can coexist, so scripts can opt into strict mode incrementally.

Strict mode helps you to write cleaner code.

Strict mode changes previously accepted "bad syntax" into real errors.

With strict mode it is impossible to accidentally create a global variable.

Strict Mode 2

```
1   // Declared at the beginning of a script, it has global scope
2   // (all code in the script will execute in strict mode):
3   "use strict";
4   myFunction();
5   function myFunction() {
6     y = 3.14; // This will cause an error because y is not declared
7   }
8   // Declared inside a function, it has local scope
9   // (only the code inside the function is in strict mode):
10  x = 3.14;  // This will not cause an error.
11  myFunction();
12  function myFunction() {
13    "use strict";
14    y = 3.14; // This will cause an error
15  }
```

Source: https://www.w3schools.com/js/js_strict.asp

Strict Mode 3

```
"use strict";
// Using a variable, without declaring it, is not allowed:
x = 3.14;                // This will cause an error
// Using an object, without declaring it, is not allowed:
x = {p1:10, p2:20};      // This will cause an error
// Deleting a variable (or object) is not allowed.
var x = 3.14;
delete x;                // This will cause an error
// Deleting a function is not allowed.
function x(p1, p2) {};
delete x;                // This will cause an error
// Duplicating a parameter name is not allowed:
function x(p1, p1) {};   // This will cause an error
// Octal numeric literals are not allowed:
var x = 010;             // This will cause an error
// Octal escape characters are not allowed:
var x = "\010";          // This will cause an error
```

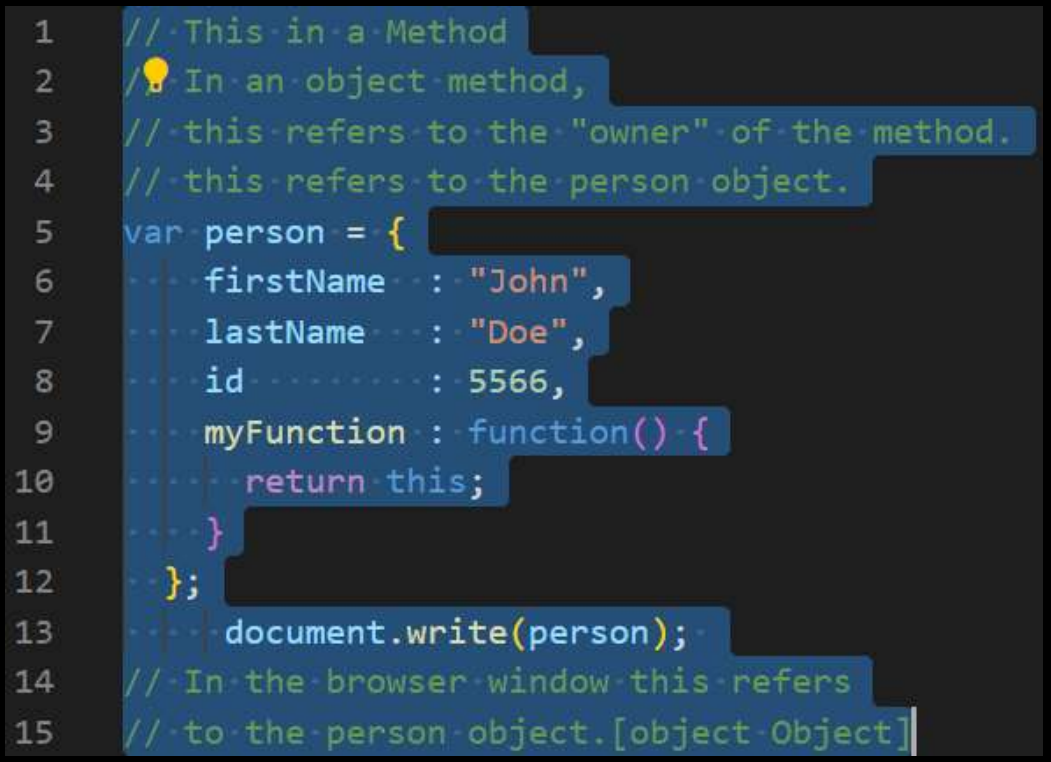
Source: https://www.w3schools.com/js/js_strict.asp

This Keyword

The JavaScript this keyword refers to the object it belongs to.

It has different values depending on where it is used:

- In a method, this refers to the owner object.
- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In a function, in strict mode, this is undefined.
- In an event, this refers to the element that received the event.
- Methods like call(), apply() and bind() can refer this to any object.



```
// This in a Method
// In an object method,
// this refers to the "owner" of the method.
// this refers to the person object.
var person = {
    firstName  : "John",
    lastName   : "Doe",
    id         : 5566,
    myFunction : function() {
      return this;
    }
};
    document.write(person);
// In the browser window this refers
// to the person object.[object Object]
```

Source: https://www.w3schools.com/js/js_this.asp

This Keyword 2

```
1   // In an object method, this refers to the "owner" of the method.
2   // In the example, this refers to the person object.
3   // The person object is the owner of the fullName method.
4   var person = {
5       firstName: "John",
6       lastName : "Doe",
7       id       : 5566,
8       fullName : function() {
9           return this.firstName + " " + this.lastName;
10      }
11  };
12  console.log(person);
13  // logs {firstName: "John", lastName: "Doe", id: 5566, fullName: f}
14  console.log(person.fullName); // logs f () {
15  // return this.firstName + " " + this.lastName;
16  // }
17  console.log(person.fullName()); // logs John Doe
```

Source: https://www.w3schools.com/js/js_this.asp

This Keyword 3

```
1   // this Alone: When used alone, the owner is the Global object,
2   // so this refers to the Global object.
3   // In a browser window the Global object is [object Window]:
4   var x = this;
5   document.write(x);
6   // In strict mode ("use strict"), when used alone,
7   // this also refers to the Global object [object Window]:
8   // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
9   // this in a Function (Default):In a JavaScript function,
10  // the owner of the function is the default binding for this.
11  // So, in a function, this refers to the Global object [object Window].
12  // [object Window]function myFunction() { return this; }
13  function myFunction() {
14      return this;
15  }
16  document.write(myFunction);
17  // Strict mode ("use strict"), does not allow default binding.
18  // So, when used in a function, in strict mode, this is undefined.
19  // function myFunction() { return this; }
```

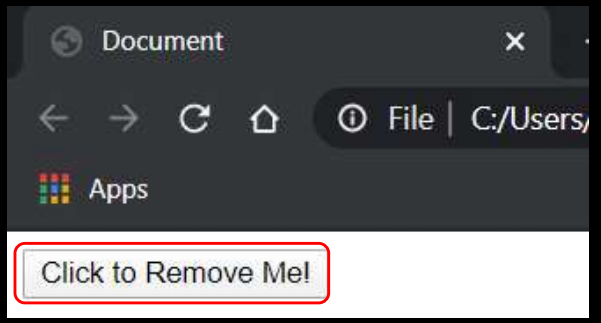
Source: https://www.w3schools.com/js/js_this.asp

This Keyword 4

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <button onclick="this.style.display='none'">Click to Remove Me!</button>
</body>
</html>
<!--this in Event Handlers:In HTML event handlers,
    this refers to the HTML element that received the event:-->
```

In HTML event handlers, this refers to the HTML element that received the event which is button in this case.

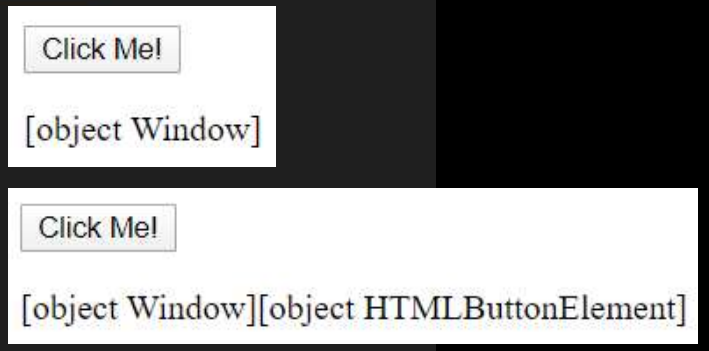
Elements in HTML are mostly "inline" or "block" elements: An inline element has floating content on its left and right side. A block element fills the entire line, and nothing can be displayed on its left or right side. If you set display:none, it hides the entire element, while visibility:hidden means that the contents of the element will be invisible, but the element stays in its original position and size.



Source: https://www.w3schools.com/js/js_this.asp and https://www.w3schools.com/jsref/prop_style_display.asp

This Keyword 5

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript "this"</h2>
<p>This example demonstrate that in a regular function, the "this" keyword
    represents different objects depending on how the function was called.</p>
<p>Click the button to execute the "hello" function again, and you will see
    that this time "this" represents the button object.</p>
<button id="btn">Click Me!</button>
<p id="demo"></p>
<script>
var hello;
hello = function() {
  document.getElementById("demo").innerHTML += this;
}
//The window object calls the function:
window.addEventListener("load", hello);
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//A button object calls the function:
document.getElementById("btn").addEventListener("click", hello);
</script>
</body>
```



Source: https://www.w3schools.com/js/js_arrow_function.asp

This Keyword with Call() and Apply()

```
1  // Explicit Function Binding:The call() and apply() methods are predefined JavaScript methods.
2  // They can both be used to call an object method with another object as argument.
3  // when calling person1.fullName with person2 as argument, this will refer to person2,
4  // even if it is a method of person1:
5  var person1 = {
6      fullName: function() {
7          return this.firstName + " " + this.lastName;
8      }
9  }
10 var person2 = {
11     firstName:"John",
12     lastName: "Doe",
13 }
14 person1.fullName.call(person2);  // Will return "John Doe"
```

The JavaScript call() method: The call() method is a predefined JavaScript method. It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

Sources: https://www.w3schools.com/js/js_this.asp and https://www.w3schools.com/js/js_function_call.asp

This Keyword with Call() and Apply()

```
// With the apply() method, you can write a method that can be used on different objects.
// In this example the fullName method of person is applied on person1:
var person = {
    fullName: function(city, country) {
        return this.firstName + " " + this.lastName + "," + city + "," + country;
    }
  }
  var person1 = {
    firstName: "Mary",
    lastName: "Doe"
  }
person.fullName.apply(person1);  // Returns "Mary Doe,undefined,undefined"
// The apply() method takes arguments as an array.
person.fullName.apply(person1, ["Oslo", "Norway"]);
// returns "Mary Doe,Oslo,Norway"
// The call() method takes arguments separately.
person.fullName.call(person1, "Oslo", "Norway");
// returns "Mary Doe,Oslo,Norway"
```

Source: https://www.w3schools.com/js/js_function_apply.asp

This Keyword with Bind()

```
// bind() enables you to do partial evaluation, you can create new functions by
// filling in parameters of an existing function:
function add(x, y) {
    return x + y;
}
const plus1 = add.bind(undefined, 1);
// With an arrow function:
const plus1 = y => add(1, y);
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// func.bind() produces a function whose name is 'bound '+func.name:
  function foo(x) {
      return x
  }
  const bound = foo.bind(undefined, 123);
  console.log(bound.name); // logs bound foo
```

The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind and https://exploringjs.com/es6/ch_arrow-functions.html#sec_arrow-func-vs-bind and … ch_callables.html#sec_iifes-in-es6

Objects

In JavaScript, almost "everything" is an object.

Booleans can be objects (if defined with the new keyword)

Numbers can be objects (if defined with the new keyword)

Strings can be objects (if defined with the new keyword)

Dates are always objects

Maths are always objects

Regular expressions are always objects

Arrays are always objects

Functions are always objects

Objects are always objects

All JavaScript values, except primitives, are objects.

Source:
https://www.w3schools.com/js/js_object_definition.asp

A primitive value is a value that has no properties or methods.

A primitive data type is data that has a primitive value.

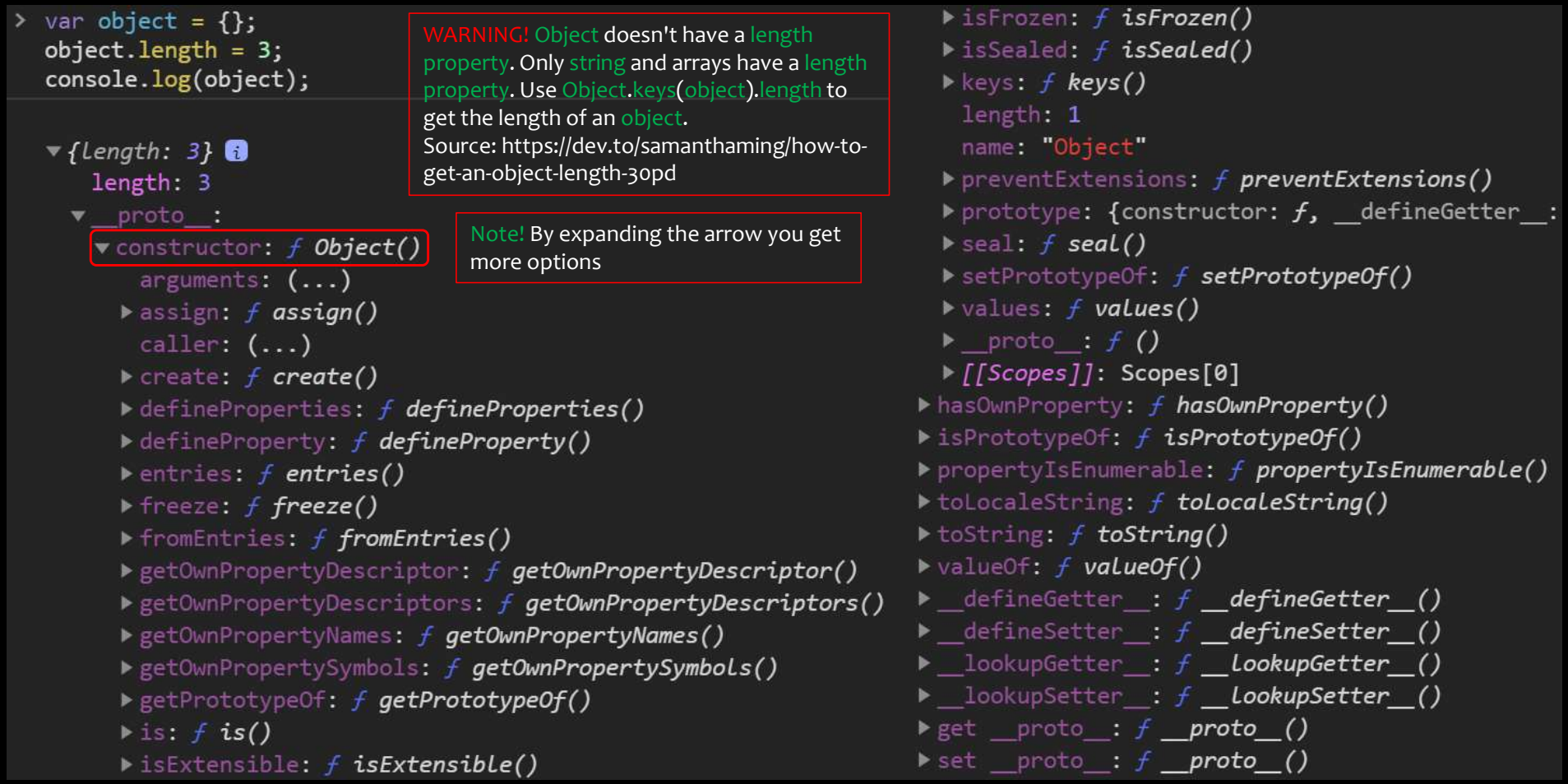
JavaScript defines 6 types of primitive data types:

- string
- number
- boolean
- null
- undefined
- symbol (new in ECMAScript 2015)

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

if x = 3.14, you can change the value of x. But you cannot change the value of 3.14.

Object Methods



WARNING! Object doesn't have a length property. Only string and arrays have a length property. Use Object.keys(object).length to get the length of an object.
Source: https://dev.to/samanthaming/how-to-get-an-object-length-30pd

Note! By expanding the arrow you get more options

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

Object: Properties and Methods

The name: value pairs in JavaScript objects are called properties:

You define (and create) a JavaScript object with an object literal:

Object literal: var person = { }

Objects can also have **methods**.

Methods are **actions** that can be performed on objects.

Methods are stored in properties as function definitions.

```
var person = {
    firstName:"James",
    lastName:"Bond",
fullName:function() {
        return this.firstName + " " + this.lastName;  }};
```

Source: https://www.w3schools.com/js/js_objects.asp

Objects

```
// Objects are mutable: They are addressed by reference, not by value.
// If person is an object, the following statement will not create a copy of person:
var x = person;  // This will not create a copy of person.
// The object x is not a copy of person. It is person. Both x and person are the same object.
// Any changes to x will also change person, because x and person are the same object.
var person = {firstName:"John", lastName:"Doe", age:50}
var x = person;
x.age = 10; // This will change both x.age and person.age
// Displaying a JavaScript object will output [object Object].
// The properties of an object can be displayed as a string:
var displayperson = person.firstName + "," + person.age;
console.log(displayperson); // logs John,10
// The properties of an object can be collected in a loop:
var txt = "";
for (x in person) {
    txt += person[x] + " ";
}; // logs "John Doe 10"
// Any JavaScript object can be converted to an array using Object.values():
var myArray = Object.values(person);
console.log(myArray); // logs ["John", "Doe", 10]
```

Source: https://www.w3schools.com/js/js_object_display.asp

Objects: Getters and Setters

```
// Getters and setters allow you to define Object Accessors (Computed Properties). Getters and setters
// allow you to get and set properties via methods. This example uses a lang property to get the value
// of the language property. JavaScript Getter (The get Keyword)
var person = {
    firstName: "John",
    language : "en",
    get lang() {
        return this.language;
    }
  };
  console.log(person.lang); // logs en
// JavaScript Setter (The set Keyword)
  var person = {
  firstName: "John",
  language : "",
  set lang(lang) {
    this.language = lang;
  }
};
// Set an object property using a setter:
person.lang = "en"; // logs "en"
```

Source: https://www.w3schools.com/js/js_object_accessors.asp

Objects: Delete and In operators

```
// The delete operator deletes a property from an object:
var person = {firstName:"John", lastName:"Doe", age:50};
delete person.age;   // or delete person["age"];
console.log(person); // logs {firstName: "John", lastName: "Doe"}
// The delete operator deletes both the value of the property
// and the property itself. After deletion, the property cannot be used
// before it is added back again.The delete operator is designed to be used
// on object properties. It has no effect on variables or functions.
// Note: The delete operator should not be used on predefined
// JavaScript object properties. It can crash your application.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The in operator returns true if the specified property is
// in the specified object, otherwise false:
// Objects
var person = {firstName:"John", lastName:"Doe", age:50};
"firstName" in person   // Returns true
"age" in person         // Returns true
// Predefined objects
"PI" in Math            // Returns true
"NaN" in Number         // Returns true
"length" in String      // Returns true
```

Source: https://www.w3schools.com/jsref/jsref_operators.asp

Object.entries, Object.keys and Object.values

```
> var person = {firstName:"John", lastName:"Doe", age:50};
  console.log(Object.entries(person)); // [Array(2), Array(2), Array(2)]
  console.log(Object.keys(person)); // ["firstName", "lastName", "age"]
  console.log(Object.values(person)); // ["John", "Doe", 50]
```

```
▼ (3) [Array(2), Array(2), Array(2)] i
  ▶ 0: (2) ["firstName", "John"]
  ▶ 1: (2) ["lastName", "Doe"]
  ▶ 2: (2) ["age", 50]
    length: 3
  ▶ __proto__: Array(0)
```

```
▼ (3) ["firstName", "lastName", "age"] i
    0: "firstName"
    1: "lastName"
    2: "age"
    length: 3
  ▶ __proto__: Array(0)
```

```
▼ (3) ["John", "Doe", 50] i
    0: "John"
    1: "Doe"
    2: 50
    length: 3
  ▶ __proto__: Array(0)
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

Destructuring Objects

```
1   // Basic assignment
2   const user = {
3       id: 42,
4       is_verified: true
5   };
6   const {id, is_verified} = user;
7   console.log(id); // 42
8   console.log(is_verified); // true
9   // Assignment without declaration: A variable can be assigned its value
10  // with destructuring separate from its declaration.
11  let a, b;
12  ({a, b} = {a: 1, b: 2});
13  // Notes: The parentheses ( ... ) around the assignment statement are required
14  // when using object literal destructuring assignment without a declaration.
15  // {a, b} = {a: 1, b: 2} is not valid stand-alone syntax, as the {a, b}
16  // on the left-hand side is considered a block and not an object literal.
17  // However, ({a, b} = {a: 1, b: 2}) is valid, as is const {a, b} = {a: 1, b: 2}
18  // Your ( ... ) expression needs to be preceded by a semicolon or it
19  // may be used to execute a function on the previous line.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Objects 2

```
1  // Assigning to new variable names: A property can be unpacked from an object
2  // and assigned to a variable with a different name than the object property.
3  const o = {p: 42, q: true};
4  const {p: foo, q: bar} = o;
5  console.log(foo); // 42
6  console.log(bar); // true
7  // Here, for example, const {p: foo} = o takes from the object o the property
8  // named p and assigns it to a local variable named foo.
9  // Default values: A variable can be assigned a default, in the case that
10 // the value unpacked from the object is undefined.
11 const {a = 10, b = 5} = {a: 3};
12 console.log(a); // 3
13 console.log(b); // 5
14 // Assigning to new variables names and providing default values:A property
15 // can be both 1) unpacked from an object and assigned to a variable with
16 // a different name and 2) assigned a default value in case the unpacked
17 // value is undefined.
18 const {a: aa = 10, b: bb = 5} = {a: 3};
19 console.log(aa); // 3
20 console.log(bb); // 5
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Objects 3

```
1  // Unpacking fields from objects passed as function parameter:
2  const user = {
3      id: 42,
4      displayName: 'jdoe',
5      fullName: {
6          firstName: 'John',
7          lastName: 'Doe'
8      }
9  };
10 function userId({id}) {
11     return id;
12 }
13 function whois({displayName, fullName: {firstName: name}}) {
14     return `${displayName} is ${name}`;
15 }
16 console.log(userId(user)); // 42
17 console.log(whois(user)); // "jdoe is John"
18 // This unpacks the id, displayName and firstName from the user object and prints them.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Objects 4

```
// Setting a function parameter's default value:
function drawChart({size = 'big', coords = {x: 0, y: 0}, radius = 25} = {}) {
  console.log(size, coords, radius);  // do some chart drawing
}
drawChart({
  coords: {x: 18, y: 30},
  radius: 30
});
// In the function signature for drawChart above, the destructured left-hand
// side is assigned to an empty object literal on the right-hand side:
// {size = 'big', coords = {x: 0, y: 0}, radius = 25} = {}. You could have also
// written the function without the right-hand side assignment. However, if you
// leave out the right-hand side assignment, the function will look for at least
// one argument to be supplied when invoked, whereas in its current form,
// you can simply call drawChart() without supplying any parameters. The current
// design is useful if you want to be able to call the function without supplying
// any parameters, the other can be useful when you want to ensure an object
// is passed to the function.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Objects 5

```js
// Nested object and array destructuring
const metadata = {
    title: 'Scratchpad',
    translations: [
        {
            locale: 'de',
            localization_tags: [],
            url: '/de/docs/Tools/Scratchpad',
            title: 'JavaScript-Umgebung'
        }
    ],
    url: '/en-US/docs/Tools/Scratchpad'
};
let {
    title: englishTitle, // rename
    translations: [
        {
            title: localeTitle, // rename
        },
    ],
} = metadata;
console.log(englishTitle); // "Scratchpad"
console.log(localeTitle);  // "JavaScript-Umgebung"
```

```js
// For of iteration and destructuring
const people = [
    {
        name: 'Mike Smith',
        family: {
            mother: 'Jane Smith',
            father: 'Harry Smith'
        },
    },
    {
        name: 'Tom Jones',
        family: {
            mother: 'Norah Jones',
            father: 'Richard Jones'
        },
    }
];
for (const {name: n, family: {father: f}} of people) {
    console.log('Name: ' + n + ', Father: ' + f);
}
// "Name: Mike Smith, Father: Harry Smith"
// "Name: Tom Jones, Father: Richard Jones"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Objects 6

```
// Computed object property names and destructuring:
// Computed property names, like on object literals, can be used with destructuring.
let key = 'z';
let {[key]: foo} = {z: 'bar'};
console.log(foo); // "bar"
// Rest in Object Destructuring:
// The Rest/Spread Properties for ECMAScript proposal (stage 4) adds the rest syntax
// to destructuring. Rest properties collect the remaining own enumerable property
// keys that are not already picked off by the destructuring pattern.
let {a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40}
a; // 10
b; // 20
rest; // { c: 30, d: 40 }
// Invalid JavaScript identifier as a property name:
// Destructuring can be used with property names that are not valid JavaScript
// identifiers by providing an alternative identifier that is valid.
const foo = { 'fizz-buzz': true };
const { 'fizz-buzz': fizzBuzz } = foo;
console.log(fizzBuzz); // "true"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Objects 7

```
// Combined Array and Object Destructuring:
// Array and Object destructuring can be combined. Say you want the third element
// in the array props below, and then you want the name property in the object,
// you can do the following:
const props = [
  { id: 1, name: 'Fizz'},
  { id: 2, name: 'Buzz'},
  { id: 3, name: 'FizzBuzz'}
];
const [,, { name }] = props;
console.log(name); // "FizzBuzz"
// The prototype chain is looked up when the object is deconstructed:
// When deconstructing an object, if a property is not accessed in itself, it will
// continue to look up along the prototype chain.
let obj = {self: '123'};
obj.__proto__.prot = '456';
const {self, prot} = obj;
// self "123"
// prot "456" (Access to the prototype chain)
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Object Constructors

```
1  // JavaScript Object Constructors. It is considered good practice to name
2  // constructor functions with an upper-case first letter.
3  function Person(first, last, age) {
4      this.firstName = first;
5      this.lastName = last;
6      this.age = age;
7      }
8  // Object Types (Blueprints) (Classes)
9  // Sometimes we need a "blueprint" for creating many objects of the same "type".
10 // The way to create an "object type", is to use an object constructor function.
11 // In the example above, function Person() is an object constructor function.
12 // Objects of the same type are created by calling the constructor function with the new keyword:
13 var myFather = new Person("John", "Doe", 50);
14 var myMother = new Person("Sally", "Rally", 48);
15 // In JavaScript, the thing called this is the object that "owns" the code.
16 // The value of this, when used in an object, is the object itself.
17 // In a constructor function this does not have a value. It is a substitute for the new object.
18 // The value of this will become the new object when a new object is created.
19 // Note that this is not a variable. It is a keyword. You cannot change the value of this.
```

Source: https://www.w3schools.com/js/js_object_constructors.asp

Object Constructors 2

```
// continues from the previous slide
// Adding a new property to an existing object is easy:
myFather.nationality = "English";
// The property will be added to myFather. Not to myMother. (Not to any other person objects).
// Adding a new method to an existing object is easy:
myFather.name = function () {
    return this.firstName + " " + this.lastName;
};
// The method will be added to myFather. Not to myMother. (Not to any other person objects).
// You cannot add a new property to an object constructor
// the same way you add a new property to an existing object:
Person.nationality = "English"; // The nationality of my father is undefined
// To add a new property to a constructor, you must add it to the constructor function:
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.nationality = "English";
}
// This way object properties can have default values.
```

Source: https://www.w3schools.com/js/js_object_constructors.asp

Object Constructors 3

```
// continues from the previous slide
// Adding a Method to a Constructor. Your constructor function can also define methods:
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.name = function() {return this.firstName + " " + this.lastName;};
  }
// You cannot add a new method to an object constructor the same way you add a new method
// to an existing object. Adding methods to an object constructor must be done
// inside the constructor function:
function Person(firstName, lastName, age, eyeColor) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.changeName = function (name) {
      this.lastName = name;
    };
  }
// The changeName() function assigns the value of name to the person's lastName property.
myMother.changeName("Doe");
// JavaScript knows which person you are talking about by "substituting" this with myMother.
```

Source: https://www.w3schools.com/js/js_object_constructors.asp

Built-in Object Constructors

```
// Built-in JavaScript Constructors.
// JavaScript has built-in constructors for native objects:
var x1 = new Object();    // A new Object object
var x2 = new String();    // A new String object
var x3 = new Number();    // A new Number object
var x4 = new Boolean();   // A new Boolean object
var x5 = new Array();     // A new Array object
var x6 = new RegExp();    // A new RegExp object
var x7 = new Function();  // A new Function object
var x8 = new Date();      // A new Date object
// The Math() object is not in the list. Math is a global object.
// The new keyword cannot be used on Math. As you can see above,
// JavaScript has object versions of the primitive data types
// String, Number, and Boolean. But there is no reason
// to create complex objects. Primitive values are much faster.
var x1 = {};           // new object
var x2 = "";           // new primitive string
var x3 = 0;            // new primitive number
var x4 = false;        // new primitive boolean
var x5 = [];           // new array object
var x6 = /()/          // new regexp object
var x7 = function(){}; // new function object
```

Source: https://www.w3schools.com/js/js_object_constructors.asp

Objects: Prototype Inheritance

All JavaScript objects inherit properties and methods from a prototype:

Date objects inherit from Date.prototype

Array objects inherit from Array.prototype

Person objects inherit from Person.prototype

Date objects, Array objects, and Person objects inherit from Object.prototype.

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

Source: https://www.w3schools.com/js/js_object_prototypes.asp

```
// Using the prototype Property. The JavaScript
// prototype property allows you to add new
// properties to object constructors:
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
}
Person.prototype.nationality = "English";
// The JavaScript prototype property also allows
// you to add new methods to objects constructors:
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
}
    Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
```

Inheritance and the Prototype Chain

JavaScript is dynamic and does not provide a class implementation per se (the class keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain.

Nearly all objects in JavaScript are instances of Object which sits on the top of a prototype chain.

While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model.

Inheriting properties:

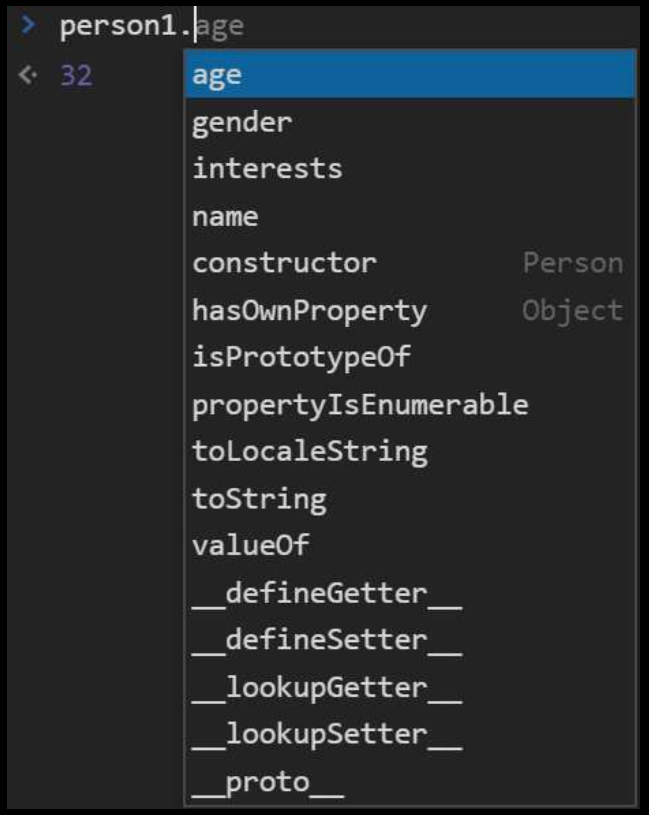
JavaScript objects are dynamic "bags" of properties (referred to as own properties). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

Inheritance and the Prototype Chain 2

Following the ECMAScript standard, the notation someObject.[[Prototype]] is used to designate the prototype of someObject. Since ECMAScript 2015, the [[Prototype]] is accessed using the accessors Object.getPrototypeOf() and Object.setPrototypeOf(). This is equivalent to the JavaScript property __proto__ which is non-standard but de-facto implemented by many browsers.

It should not be confused with the func.prototype property of functions, which instead specifies the [[Prototype]] to be assigned to all instances of objects created by the given function when used as a constructor. The Object.prototype property represents the Object prototype object.



Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

Inheritance and the Prototype Chain 3

```
1  // We have defined a constructor function
2  function Person(first, last, age, gender, interests) {
3  // property and method definitions
4    this.name = {
5      'first': first,
6      'last' : last
7    };
8    this.age = age;
9    this.gender = gender;
10   this.interests = interests;
11 }
12 // We have then created an object instance like this:
13 let person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Inheritance and the Prototype Chain 4

```
function Person(first, last, age, gender, interests) {
  // property and method definitions
    this.name = {
      'first': first,
      'last' : last
    };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }
  // We have then created an object instance like this:
  let person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
  // What happens if you call a method on person1, which is actually defined on Object?
  For example:
  person1.valueOf()
```

```
Person {name: {…}, age: 32, gender: "male", interests: Array(2)}
  age: 32
  gender: "male"
  interests: (2) ["music", "skiing"]
  name: {first: "Bob", last: "Smith"}
  __proto__: Object
```

```
person1.age
32
```

age
gender
interests
name
constructor Person
hasOwnProperty Object
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__

If you type "person1." into your JavaScript console, you should see the browser try to auto-complete this with the member names available on this object:

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Inheritance and the Prototype Chain 5

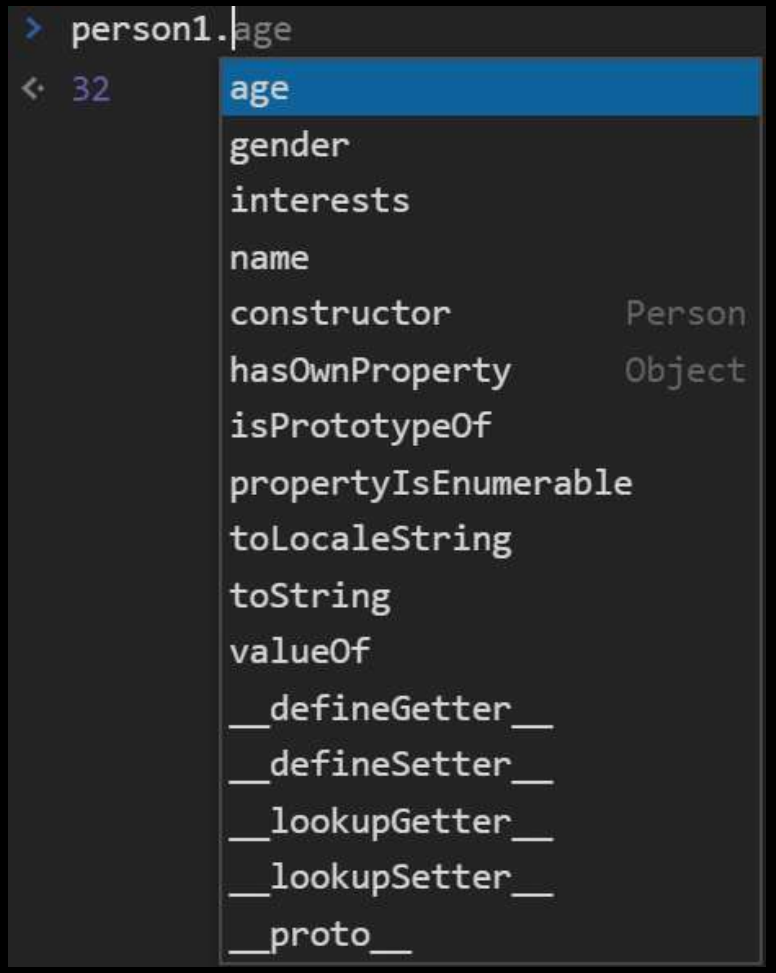
Understanding prototype objects:

Here we'll go back to the example in which we finished writing our Person() constructor .

If you type "person1." into your JavaScript console, you should see the browser try to auto-complete this with the member names available on this object:

In this list, you will see the members defined on person1's constructor — Person() — name, age, gender and interests. You will however also see some other members — toString, valueOf, etc — these are defined on Person() 's constructor's prototype object, which is Object.

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes



Inheritance and the Prototype Chain 6

```
person1.valueOf()
Person {name: {…}, age: 32, gender: "male", interests: Array(2)}
    age: 32
    gender: "male"
  ▶ interests: (2) ["music", "skiing"]
  ▶ name: {first: "Bob", last: "Smith"}
  ▶ __proto__: Object
```

This method — Object.valueOf() is inherited by person1 because its constructor is Person(), and Person()'s prototype is Object(). valueOf() returns the value of the object it is called on — try it and see! In this case, what happens is:

The browser initially checks to see if the person1 object has a valueOf() method available on it, as defined on its constructor, Person().

It doesn't, so the browser then checks to see if the Person() constructor's prototype object (Object()) has a valueOf() method available on it. It does, so it is called, and all is good!

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Inheritance and the Prototype Chain 7

```
> person1.__proto__
{constructor: ƒ} i
  ▶ constructor: ƒ Person(first, last, age, gender, interests)
  ▶ __proto__: Object
> person1.__proto__.__proto__
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__: ƒ, …} i
  ▶ constructor: ƒ Object()
  ▶ hasOwnProperty: ƒ hasOwnProperty()
  ▶ isPrototypeOf: ƒ isPrototypeOf()
  ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
  ▶ toLocaleString: ƒ toLocaleString()
  ▶ toString: ƒ toString()
  ▶ valueOf: ƒ valueOf()
  ▶ __defineGetter__: ƒ __defineGetter__()
  ▶ __defineSetter__: ƒ __defineSetter__()
  ▶ __lookupGetter__: ƒ __lookupGetter__()
  ▶ __lookupSetter__: ƒ __lookupSetter__()
  ▶ get __proto__: ƒ __proto__()
  ▶ set __proto__: ƒ __proto__()
```

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

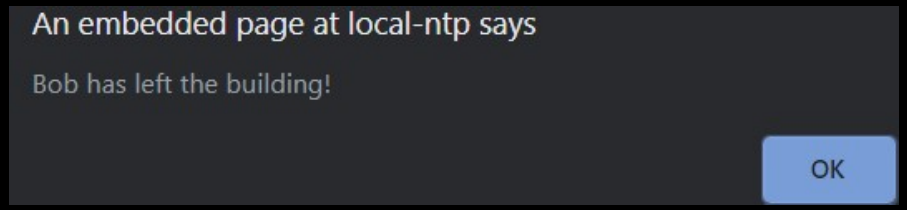
Inheritance and the Prototype Chain 8

```
> Person.prototype
{constructor: ƒ}
  constructor: ƒ Person(first, last, age, gender, interests)
  __proto__: Object
> Object.prototype
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__: ƒ, …}
  constructor: ƒ Object()
  hasOwnProperty: ƒ hasOwnProperty()
  isPrototypeOf: ƒ isPrototypeOf()
  propertyIsEnumerable: ƒ propertyIsEnumerable()
  toLocaleString: ƒ toLocaleString()
  toString: ƒ toString()
  valueOf: ƒ valueOf()
  __defineGetter__: ƒ __defineGetter__()
  __defineSetter__: ƒ __defineSetter__()
  __lookupGetter__: ƒ __lookupGetter__()
  __lookupSetter__: ƒ __lookupSetter__()
  get __proto__: ƒ __proto__()
  set __proto__: ƒ __proto__()
```

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Inheritance and the Prototype Chain 9

```
// Modifying prototypes:Let's have a look at an example
// of modifying the prototype property of a constructor
// function — methods added to the prototype are then
// available on all object instances created from the
// constructor. At this point we'll finally add
// something to our Person() constructor's prototype.
// Below the existing JavaScript, add the following code,
// which adds a new method to the constructor's
// prototype property:
Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building!');
};
// Save the code and load the page in the browser,
// and try entering the following into the text input:
person1.farewell();
// You should get an alert message displayed, featuring
// the person's name as defined inside the constructor.
// This is really useful, but what is even more useful
// is that the whole inheritance chain has updated
// dynamically, automatically making this new method
// available on all object instances derived from
// the constructor.
```




```
> person1.constructor
< ƒ Person(first, last, age, gender, interests) {
  // property and method definitions
    this.name = {
      'first': first,
      'last' : last
    };
    this.age = age;
    this.gender = gender;
    this.interest…
```

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Classes

```
// A class is a type of function, but instead of using the keyword function to initiate it,
// we use the keyword class, and the properties are assigned inside a constructor() method.
// Class Definition: Use the keyword class to create a class, and always add the constructor() method.
// The constructor method is called each time the class object is initialized.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Methods: The constructor method is special, it is where you initialize properties, it is called
// automatically when a class is initiated, and it has to have the exact name "constructor", in fact,
// if you do not have a constructor method, JavaScript will add an invisible and empty constructor method.
// You are also free to make your own methods: Create a method named "present":
class Car {
    constructor(brand) {
      this.carname = brand;
    }
    present(x) {
      return x + ", I have a " + this.carname;
    }
  }
// Now you can create objects using the Car class:
mycar = new Car("Ford");
// The constructor method is special, it is where you initialize properties, it is called automatically
// when a class is initiated, and it has to have the exact name "constructor", in fact, if you do not
// have a constructor method, JavaScript will add an invisible and empty constructor method.
console.log(mycar.present("Hello")); // Hello, I have a Ford
```

Source: https://www.w3schools.com/js/js_classes.asp

Classes 2

```
// Static Methods: Static methods are defined on the class itself, and not on the prototype.
// That means you cannot call a static method on the object (mycar), but on the class (Car):
class Car {
    constructor(brand) {
        this.carname = brand;
    }
    static hello() {
        return "Hello!!";
    }
    static hi(x) {
        return "Hi!! " + x.carname ;
    }
}
mycar = new Car("Ford");
// Call 'hello()' on the class Car:
console.log(Car.hello());  // Hello!!
// If you want to use the mycar object inside the static method, you can send it as a parameter:
console.log (Car.hi(mycar)); // Hi!! Ford
// Do not Call 'hello()' on the 'mycar' object:
console.log (mycar.hello()); // TypeError: mycar.hello is not a function
// this would raise an error.
```

Source: https://www.w3schools.com/js/js_classes.asp

Classes 3

```
// Inheritance: To create a class inheritance, use the extends keyword.
// A class created with a class inheritance inherits all the methods from another class:
class Car {
    constructor(brand) {
        this.carname = brand;
    }
    present() {
        return 'I have a ' + this.carname;
    }
}
class Model extends Car {
    constructor(brand, mod) {
        super(brand);
        this.model = mod;
    }
    show() {
        return this.present() + ', it is a ' + this.model;
    }
}
mycar = new Model("Ford", "Mustang");
console.log(mycar.show()); // I have a Ford, it is a Mustang
// The super() method refers to the parent class.By calling the super() method in the constructor
// method, we call the parent's constructor method and gets access to the parent's properties and methods.
```

Source: https://www.w3schools.com/js/js_classes.asp

Classes 4

```
// Getters and Setters: Classes also allows you to use getters and setters.It can be
// smart to use getters and setters for your properties, especially if you want to do
// something special with the value before returning them, or before you set them.
// To add getters and setters in the class, use the get and set keywords.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The name of the getter/setter method cannot be the same as the name of the property,
// in this case carname. // Many programmers use an underscore character _ before the
// property name to separate the getter/setter from the actual property:
class Car {
    constructor(brand) {
        this._carname = brand;
    }
    get carname() {
        return this._carname;
    }
    set carname(x) {
        this._carname = x;
    }
}
mycar = new Car("Ford");
console.log(mycar.carname); // Ford
```

Source: https://www.w3schools.com/js/js_classes.asp

Classes 5

```
// To use a setter, use the
// same syntax as when
// you set a property value,
// without parentheses:
class Car {
    constructor(brand) {
        this._carname = brand;
    }
    get carname() {
        return this._carname;
    }
    set carname(x) {
        this._carname = x;
    }
}
mycar = new Car("Ford");
mycar.carname = "Volvo";
console.log(mycar.carname);
// Volvo
```

```
// Hoisting: Unlike functions, and other
// JavaScript declarations, class declarations
// are not hoisted. That means that you must
// declare a class before you can use it:
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// You cannot use the class yet.
// mycar = new Car("Ford")
// This would raise an error.
class Car {
    constructor(brand) {
        this.carname = brand;
    }
}
// Now you can use the class:
mycar = new Car("Ford")
```

Source: https://www.w3schools.com/js/js_classes.asp

Date Objects

Date objects are static. The computer time is ticking, but date objects are not.

By default, JavaScript will use the browser's time zone and display a date as a full text string.

A JavaScript date is fundamentally specified as the number of milliseconds that have elapsed since midnight on January 1, 1970, UTC (Zero time). This date and time is the same as the UNIX epoch, which is the predominant base value for computer-recorded date and time values.

Sources: https://www.w3schools.com/js/js_dates and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

```
// new Date() creates a new date object
// with the current date and time:
var d = new Date();
console.log(d);
// logs: Tue Mar 31 2020 23:33:26 GMT+0300
// (Eastern European Summer Time)
// new Date(year, month, ...) creates a new
// date object with a specified date and time.
// 7 numbers specify year, month, day, hour,
// minute, second, and millisecond (in that order):
var d = new Date(2018, 11, 24, 10, 33, 30, 0);
console.log(d);
// Logs: Mon Dec 24 2018 10:33:30 GMT+0200
// (Eastern European Standard Time)
// Note: JavaScript counts months from 0 to 11.
// January is 0. December is 11.
// Note: In JavaScript, the first day
// of the week (0) means "Sunday", even if some
// countries in the world consider the first
// day of the week to be "Monday"
```

Date Objects 2

```
let today = new Date() // 7 numbers specify year, month, day, hour, minute, second,
// and millisecond (in that order)
let birthday = new Date(1995, 11, 17) // Commas are ignored. Names are case insensitive:
let birthday = new Date(1995, 11, 17, 3, 24, 0, 0)   // the month is 0-indexed
// The ISO 8601 syntax (YYYY-MM-DD) is the preferred JavaScript date format:
let birthday = new Date('2015-03-25')
// ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):
let birthday = new Date('2015-03-25T12:00:00Z')
// Date and time is separated with a capital T. UTC time is defined with a capital letter Z.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Short dates are written with an "MM/DD/YYYY" syntax like this:
let birthday = new Date('03/25/2015')
// Long dates are most often written with a "MMM DD YYYY" syntax like this:
let birthday = new Date('Mar 25 2015')
// If you have a valid date string, you can use the Date.parse() method to convert it to milliseconds.
// Date.parse() returns the number of milliseconds between the date and January 1, 1970:
// You can then use the number of milliseconds to convert it to a date object:
var msec = Date.parse('March 21, 2012');
console.log(msec); // logs 1332280800000
var d = new Date(msec);
console.log(d); // logs Wed Mar 21 2012 00:00:00 GMT+0200 (Eastern European Standard Time)
```

Source: https://www.w3schools.com/js/js_date_formats.asp

Object and Array rules

DOT NOTATION . You can use dot with strings

```
var person = {}; //is an empty object
person.name = "Esa";
var person = {
    "name" : "Esa"
};
```

- strings
- numbers
- quotations
- weird characters
- expressions

Source: https://slides.com/bgando/f2f-final-day-1#/0/1

BRACKETS [ ] You can use brackets with all forms

```
var person = [ ] ; // [ ] is an empty array
person.name = "Esa";
person[0] = "I drink milk";
```

"quotes required"

- "strings"
- numbers
- variables
- "weird characters"
- expressions

Source: https://slides.com/bgando/f2f-final-day-1#/0/1

Arrays

An array can hold many values under a single name, and you can access the values by referring to an index number.

Array indexes start with 0.

[0] is the first element. [1] is the second element.

Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.

The length property of an array returns the length of an array (the number of array elements).

The length property is always one more than the highest array index.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length;    // the length of fruits is 4
var first = fruits[0]; // logs "Banana"
var last = fruits[fruits.length - 1]; // logs "Mango"
```

Source:
https://www.w3schools.com/js/js_arrays.asp

Array Methods

```
> var numbers = [];
  numbers.length = 3;
  console.log(numbers); // [undefined, undefined, undefined]

▼ (3) [empty × 3] i
    length: 3
  ▼ __proto__: Array(0)
    ▶ concat: ƒ concat()
    ▶ constructor: ƒ Array()
    ▶ copyWithin: ƒ copyWithin()
    ▶ entries: ƒ entries()
    ▶ every: ƒ every()
    ▶ fill: ƒ fill()
    ▶ filter: ƒ filter()
    ▶ find: ƒ find()
    ▶ findIndex: ƒ findIndex()
    ▶ flat: ƒ flat()
    ▶ flatMap: ƒ flatMap()
    ▶ forEach: ƒ forEach()
    ▶ includes: ƒ includes()
    ▶ indexOf: ƒ indexOf()
    ▶ join: ƒ join()
    ▶ keys: ƒ keys()
    ▶ lastIndexOf: ƒ lastIndexOf()
      length: 0
    ▶ map: ƒ map()
    ▶ pop: ƒ pop()
    ▶ push: ƒ push()
    ▶ reduce: ƒ reduce()
    ▶ reduceRight: ƒ reduceRight()
    ▶ reverse: ƒ reverse()
    ▶ shift: ƒ shift()
    ▶ slice: ƒ slice()
    ▶ some: ƒ some()
    ▶ sort: ƒ sort()
    ▶ splice: ƒ splice()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ unshift: ƒ unshift()
    ▶ values: ƒ values()
    ▶ Symbol(Symbol.iterator): ƒ values()
    ▶ Symbol(Symbol.unscopables): {copyWithin:
    ▶ __proto__: Object
```

Note! By expanding the arrow you get more options

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length

Array Methods 2

```
let fruits = ['Apple','Banana'];
// ["Apple", "Banana"]
// Add to the end of an Array
let newLength = fruits.push('Orange') //add to the end
// ["Apple", "Banana", "Orange"]
// Remove from the end of an Array
let last = fruits.pop() // remove Orange (from the end)
// ["Apple", "Banana"]
// Remove from the front of an Array
let first = fruits.shift() // remove Apple from the front
// ["Banana"]
// Add to the front of an Array
let newLength = fruits.unshift('Strawberry') // add to the front
// ["Strawberry", "Banana"]
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Array Methods 3

```
let fruits = ['Apple','Banana','Mango'];
// Remove an item by index position
// this is how to remove an item and optionally replace it
let removedItem = fruits.splice(1, 1,'Orange')
    console.log(removedItem) // ['Banana']
    console.log(fruits) // logs ['Apple','Orange','Mango']
// splice(index, count_to_remove, addElement1, addElement2, ...)
// removes elements from an array and (optionally) replaces them.
// It returns the items which were removed from the array.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
let shallowCopy = fruits.slice(0,2) // this is how to make a copy
    console.log(shallowCopy) // logs ['Apple','Mango']
// slice(start_index, upto_index) extracts a section of an array
// and returns a new array.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Array Methods 4

```
let fruits = ['Apple','Banana','Mango'];
// Loop over an Array
fruits.forEach(function(item, index, array) {
    console.log(item, index) })
// Apple 0
// Banana 1
// Mango 2
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Find the index of an item in the Array
let pos = fruits.indexOf('Banana')
    console.log(pos)
// 1
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// reverse() transposes the elements of an array, in place:
// the first array element becomes the last and the last becomes the first.
// It returns a reference to the array.
fruits.reverse()
console.log(fruits)
// ["Mango", "Banana", "Apple"]
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections

Array Methods 5

```
let fruits = ['Kiwi','Banana','Mango'];
// sort() sorts the elements of an array in place,
// and returns a reference to the array.
fruits.sort()
console.log(fruits) // ["Banana", "Kiwi", "Mango"]
// concat() joins two or more arrays and returns a new array.
let cars = ['Mazda','BMW','Volvo']
fruits = fruits.concat(cars)
console.log(fruits) // ["Banana", "Kiwi", "Mango", "Mazda", "BMW", "Volvo"]
// join(delimiter = ',') joins all elements of an array into a string.
let list = cars.join('-')
console.log(list) // Mazda-BMW-Volvo
// The typeof operator returns a string indicating
// the type of the unevaluated operand.
typeof(list) // "string"
// The Array.isArray() method determines whether the passed value is an Array.
Array.isArray(cars) //true
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections

Array Methods 6: Array.reduce() and Array.reduceRight()

Both methods reduces the array to a single value.

The reduce() method executes a provided function for each value of the array (from left-to-right). The reduceRight() method executes a provided function for each value of the array (from right-to-left).

The return value of both methods of the function are stored in an accumulator (result/total).

Note! Both methods do not execute the function for array elements without values. Both methods do not change the original array.

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce and https://www.w3schools.com/jsref/jsref_reduce.asp

Parameter	Description
*function(total,currentValue, index,arr) or function(*Accumulator, Current Value, Current Index, Source Array)	Required. A function to be run for each element in the array. Function arguments:
Argument	Description
Total or Accumulator (acc)	Required. The *initialValue*, or the previously returned value of the function
CurrentValue or Current Value (cur)	Required. The value of the current element
CurrentIndex or Current Index (idx)	Optional. The array index of the current element
Arr or Source Array (src)	Optional. The array object the current element belongs to
initialValue	Optional. A value to be passed to the function as the initial value

Array Methods 7: Array.reduce() and ReduceRight()

```
// The reduce() method executes a provided function
// for each value of the array (from left-to-right).
var numbers = [175, 50, 25];
function myFunc(total, num) {
  return total - num;
}
numbers.reduce(myFunc); // logs 100
// The reduceRight() method executes a provided function
// for each value of the array (from right-to-left).
numbers.reduceRight(myFunc); // logs -200
// The reduce() method executes a provided function
// for each value of the array (from left-to-right).
var numbers = [2, 45, 30, 100];
function getSum(total, num,) {
  return total - num;
}
numbers.reduce(getSum, 1); // logs -176 (with initialValue of 1)
// The reduceRight() method executes a provided function
// for each value of the array (from right-to-left).
numbers.reduceRight(getSum, 47); // logs -130 (with initialValue of 47)
```

Source: https://www.w3schools.com/jsref/jsref_reduce.asp and https://www.w3schools.com/jsref/jsref_reduceright.asp

Array Methods 8: Array.map() and Array.filter()

```
// The map() method creates a new array by performing a function on each array element.
// The map() method does not execute the function for array elements without values.
// The map() method does not change the original array.
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction);
function myFunction(value, index, array) {
  return value * 2;
}
console.log(numbers2); // logs [90, 8, 18, 32, 50]
// When a callback function uses only the value parameter,
// the index and array parameters can be omitted:
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The filter() method creates a new array with array elements that passes a test.
// This example creates a new array from elements with a value larger than 18:
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
console.log(over18); // logs [45, 25]
```

Source: https://www.w3schools.com/js/js_array_iteration.asp

Array Methods 9

```
// The every() method check if all array values pass a test.
// This example check if all array values are larger than 18:
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.every(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
console.log(allOver18); // logs false
// The some() method check if some array values pass a test.
// This example check if some array values are larger than 18:
var numbers = [45, 4, 9, 16, 25];
var someOver18 = numbers.some(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
console.log(someOver18); // logs true
// The indexOf() method searches an array for an element value and returns its position.
var fruits = ["Apple", "Orange", "Apple", "Mango"];
var a = fruits.indexOf("Apple");
console.log(a); // logs 0
// Syntax: array.indexOf(item, start). Array.indexOf() returns -1 if the item is not found.
// If the item is present more than once, it returns the position of the first occurrence.
```

Source: https://www.w3schools.com/js/js_array_iteration.asp

Array Methods 10

```
// Array.lastIndexOf() is the same as Array.indexOf(),
// but returns the position of the last occurrence of the specified element.
var fruits = ["Apple", "Orange", "Apple", "Mango"];
var a = fruits.lastIndexOf("Apple");
console.log(a); // logs 2
// The find() method returns the value of the first array element that passes a test function.
// This example finds (returns the value of) the first element that is larger than 18:
var numbers = [4, 9, 16, 25, 29];
var first = numbers.find(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
console.log(first); // logs 25
// The findIndex() method returns the index of the first array element that passes a test function.
// This example finds the index of the first element that is larger than 18:
var numbers = [4, 9, 16, 25, 29];
var first = numbers.findIndex(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
console.log(first); // logs 3
```

Source: https://www.w3schools.com/js/js_array_iteration.asp

Array: Adding Elements with High Indexes

```
1  const fruits = ['Apple','Banana'];
2  fruits[5] = 'mango'
3  console.log(fruits[5])            // 'mango'
4  console.log(Object.keys(fruits))  // ['0', '1', '5']
5  console.log(fruits.length)        // 6
6  console.log(fruits) // ["Apple", "Banana", empty × 3, "mango"]
```

WARNING ! Adding elements with high indexes can create undefined “empty holes" in an array. When setting a property on a JavaScript array when the property is a valid array index and that index is outside the current bounds of the array, the engine will update the array's length property accordingly:

Sources: https://www.w3schools.com/js/js_arrays.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Increasing and Deacreasing the Lenght of an Array

```
1  const cars = ['Volvo','Mazda'];
2  cars.length = 10 // Increasing the length.
3  console.log(Object.keys(cars)) // ['0', '1',]
4  console.log(cars.length)       // 10
5  console.log(cars) // ['Volvo','Mazda', empty × 8,]
6  cars.length = 1 // Decreasing the length does delete items
7  console.log(Object.keys(cars)) // ['0']
8  console.log(cars.length) // 1
9  console.log(cars) // ['Volvo']
```

Note! JavaScript arrays are zero-indexed: the first element of an array is at index 0, and the last element is at the index equal to the value of the array's length property minus 1. Using an invalid index number returns undefined. A JavaScript array's length property and numerical properties are connected. Several of the built-in array methods (e.g., join(), slice(), indexOf(), etc.) take into account the value of an array's length property when they're called.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

4 Ways to Empty an Array in JavaScript

```
// Suppose you have the following array and want to remove all of its elements:
let a = [1,2,3];
// The following shows you several methods to make an array empty.1) Assigning it to a
// new empty array. This is the fastest way to empty an array:
a = [];
// This code assigned the array a to a new empty array. It works perfectly if you do not
// have any references to the original array.See the following example:
let b = a;
a = [];
console.log(b); // [1,2,3]
// In this example, first, the b variable references the array a. Then, the a is assigned
// to an empty array. The original array still remains unchanged. 2) Setting its length to zero
// The second way to empty an array is to set its length to zero:
a.length = 0;
// The length property is read/write property of an Array object. When the length property is set
// to zero, all elements of the array are automatically deleted.3) Using splice() method
// The third way to empty an array is to remove all of its elements using the splice() method as
// shown in the following example:
a.splice(0,a.length);
// In this solution, the splice() method removed all the elements of the a array and returned the
// removed elements as an array. 4) Using pop() method. The fourth way to empty an array is to
// remove each element of the array one by one using the while loop and pop() method:
while(a.length > 0) {
    a.pop();
}
// This solution is quite trivial and is the slowest one in terms of performance.
```

Source: https://www.javascripttutorial.net/array/4-ways-empty-javascript-array/

Destructuring Arrays

```
// The object and array literal expressions provide an easy way to create ad hoc packages of data.
const x = [1, 2, 3, 4, 5];
// The destructuring assignment uses similar syntax, but on the left-hand side of the assignment
// to define what values to unpack from the sourced variable.
const x = [1, 2, 3, 4, 5];
const [y, z] = x;
console.log(y); // 1
console.log(z); // 2
// Array destructuring: Basic variable assignment
const foo = ['one', 'two', 'three'];
const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
// Assignment separate from declaration
// A variable can be assigned its value via destructuring separate from the variable's declaration.
let a, b;
[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Arrays 2

```
1   // Default values: A variable can be assigned a default, in the case that the
2   // value unpacked from the array is undefined.
3   let a, b;
4   [a=5, b=7] = [1];
5   console.log(a); // 1
6   console.log(b); // 7
7   // Swapping variables: Two variables values can be swapped in one destructuring expression.
8   // Without destructuring assignment, swapping two values requires a temporary variable
9   // (or, in some low-level languages, the XOR-swap trick).
10  let a = 1;
11  let b = 3;
12  [a, b] = [b, a];
13  console.log(a); // 3
14  console.log(b); // 1
15  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
16  const arr = [1,2,3];
17  [arr[2], arr[1]] = [arr[1], arr[2]];
18  console.log(arr); // [1,3,2]
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Arrays 3

```
// Parsing an array returned from a function: It's always been possible to return an array
// from a function. Destructuring can make working with an array return value more concise.
// In this example, f() returns the values [1, 2] as its output, which can be parsed in
// a single line with destructuring.
function f() {
  return [1, 2];
}
let a, b;
[a, b] = f();
console.log(a); // 1
console.log(b); // 2
// Ignoring some returned values: You can ignore return values that you're not interested in:
function f() {
  return [1, 2, 3];
}
const [a, , b] = f();
console.log(a); // 1
console.log(b); // 3
// You can also ignore all returned values:
[,,] = f();
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Arrays 4

```
// Assigning the rest of an array to a variable:
// When destructuring an array, you can unpack and assign
// the remaining part of it to a variable using the rest pattern:
const [a, ...b] = [1, 2, 3];
console.log(a); // 1
console.log(b); // [2, 3]
// Be aware that a SyntaxError will be thrown if a trailing comma
// is used on the left-hand side with a rest element:
const [a, ...b,] = [1, 2, 3];
// SyntaxError: rest element may not have a trailing comma
// Always consider using rest operator as the last element
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring Arrays 5

```
// Unpacking values from a regular expression match: When the regular
// expression exec() method finds a match, it returns an array containing
// first the entire matched portion of the string and then the portions of
// the string that matched each parenthesized group in the regular expression.
// Destructuring assignment allows you to unpack the parts out of this array
// easily, ignoring the full match if it is not needed.
function parseProtocol(url) {
    const parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.*)$/.exec(url);
    if (!parsedURL) {
        return false;
    }
    console.log(parsedURL);
    // ["https://developer.mozilla.org/en-US/Web/JavaScript",
    // "https", "developer.mozilla.org", "en-US/Web/JavaScript"]
    const [, protocol, fullhost, fullpath] = parsedURL;
    return protocol;
}
console.log(parseProtocol('https://developer.mozilla.org/en-US/Web/JavaScript'));
// "https"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Loops and Iteration

A for loop repeats until a specified condition evaluates to false.

The JavaScript for loop is similar to the Java and C for loop.

A for statement looks as follows:

```
for ([initialExpression]; [condition];
[incrementExpression])
  statement
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration

The statements for loops provided in JavaScript are:

- for statement
- do...while statement
- while statement
- labeled statement
- break statement
- continue statement
- for...in statement
- for...of statement

Loops and Iteration 2

- Step 1 When a for loop executes, the following occurs:

  The initializing expression initialExpression, if any, is executed.

  This expression usually initializes one or more loop counters.

- Step 2 The condition expression is evaluated. If the value of condition is true,

  the loop statements execute.

  If the value of condition is false, the for loop terminates.

  (If the condition expression is omitted entirely,

  the condition is assumed to be true.)

- Step 3 The statement executes. To execute multiple statements, use a block statement ({ ... }) to group those statements.

  If present, the update expression incrementExpression is executed.

- Step 4 Control returns to Step 2.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration

For Loop and For In Loop

```
const cities = ["Seattle","San Francisco"];
// In for loop, we're using that i control variable which gets incremented each loop.
// We use that i to access each item in the array on each iteration of the loop.
// We have the loop stop when i gets equal to the length of cities.
for (let i = 0; i < cities.length; i++) {
  console.log(cities[i]);
}
// logs:
// Seattle
// San Francisco
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// The JavaScript for/in statement loops through the properties of an object:
var person = {fname:"John", lname:"Doe"};
var quotes = "";
var x;
for (x in person) {
  quotes += person[x] + " ";
}
// logs: "John Doe"
```

Source: https://btholt.github.io/intro-to-web-dev-v2/objects-and-arrays

For Of Loops

```
// The JavaScript for/of statement loops through the values of an iterable objects
// for/of lets you loop over data structures that are iterable such as
// Arrays, Strings, Maps, NodeLists, and more. Looping over an Array:
var cars = ['BMW', 'Volvo'];
var x;
for (x of cars) {
  console.log(x) + "\n";
}
// logs:
// BMW
// Volvo
// ~~~~~~~~~~~~~~~~~~~~~~~~~~
// Looping over a String:
var txt = 'OK';
var x;
for (x of txt) {
  console.log(x + "\n");
}
// logs:
// O
// K
```

Source: https://www.w3schools.com/js/js_loop_for.asp

While Loop

```
// The while loop loops through a block of code as long
// as a specified condition is true.
// The following while loop iterates as long as
// n is less than three.
var n = 0;
var x = 0;
while (n < 3) {
  n++;
  x += n;
}
// Each iteration, the loop increments n and adds it to x.
// Therefore, x and n take on the following values:
// After the first pass: n = 1 and x = 1
// After the second pass: n = 2 and x = 3
// After the third pass: n = 3 and x = 6
// After completing the third pass, the condition
// n < 3 is no longer true, so the loop terminates.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while

Do While Loop

```
1   // The do...while statement creates a loop
2   // that executes a specified statement
3   // until the test condition evaluates to false.
4   // The condition is evaluated
5   // after executing the statement, resulting in
6   // the specified statement executing at least once.
7   let result = "";
8   let i = 0;
9   do {
10    i = i + 1;
11    result = result + i;
12  } while (i < 5);
13  console.log(result); // results 12345
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/do...while

Break and Continue

The break and the continue statements are the only JavaScript statements that can "jump out of" a code block.

A code block is a block of code between { and }.

The continue statement (with or without a label reference) can only be used to skip one loop iteration.

Source:
https://www.w3schools.com/js/js_break.asp

The break statement, without a label reference,

can only be used to jump out of a loop or a switch.

With a label reference, the break statement

can be used to jump out of any code block.

To label JavaScript statements you precede the statements with a label name and a colon.

label:
statements

break labelname;

continue labelname;

Break and Continue 2

```
const cities = ["Seattle","San Francisco", "Salt Lake City"];

for (let i = 0; i < cities.length; i++) {
    if (i === 1) {continue;}
    console.log(cities[i]);
}
// results:
// Seattle
// Salt Lake City
```

```
const cities = ["Seattle","San Francisco", "Salt Lake City"];

for (let i = 0; i < cities.length; i++) {
    if (i === 1) {break;}
    console.log(cities[i]);
}
// results:
// Seattle
```

Sources: https://www.w3schools.com/js/js_break.asp and https://btholt.github.io/intro-to-web-dev-v2/objects-and-arrays/

Infinite Loops

```
1   // WARNING! Avoid infinite loops.
2   // Make sure the condition in a loop
3   // eventually becomes false—otherwise,
4   // the loop will never terminate!
5   // The statements in the following
6   // while loop execute forever
7   // because the condition never becomes false:
8   while (true) {
9       console.log('Hello, world!');
10  }
11  // Infinite loops are bad!
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration

Conditionals: If Else and Else If and Switch

Conditional statements are used to perform different actions based on different conditions.

In JavaScript we have the following conditional statements:

Use if to specify a block of code to be executed, if a specified condition is true.

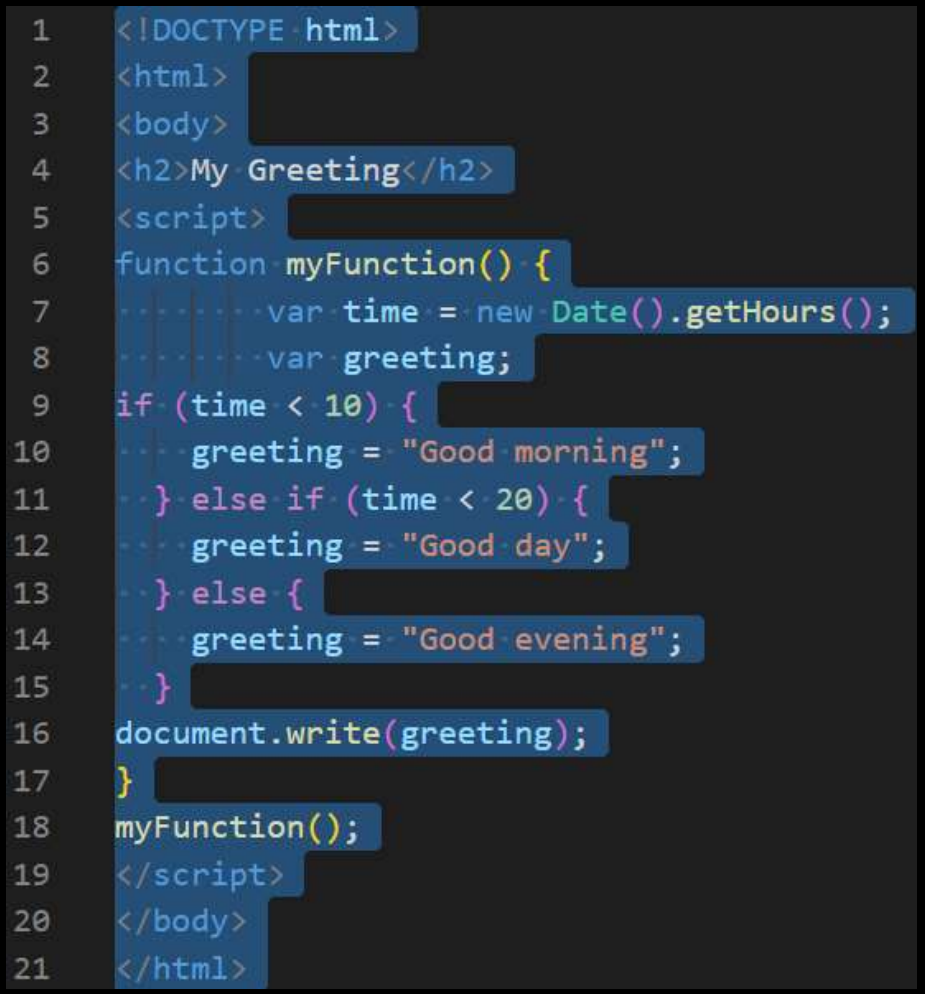
Use else to specify a block of code to be executed, if the same condition is false.

Use else if to specify a new condition to test, if the first condition is false.

Use switch to specify many alternative blocks of code to be executed.

Note! that if is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

Source: https://www.w3schools.com/js/js_if_else.asp



```
<!DOCTYPE html>
<html>
<body>
<h2>My Greeting</h2>
<script>
function myFunction() {
    var time = new Date().getHours();
    var greeting;
if (time < 10) {
    greeting = "Good morning";
  } else if (time < 20) {
    greeting = "Good day";
  } else {
    greeting = "Good evening";
  }
document.write(greeting);
}
myFunction();
</script>
</body>
</html>
```

Switch Statement

```
var Animal = 'Giraffe';
switch (Animal) {
  case 'Cow':
  case 'Giraffe':
  case 'Dog':
  case 'Pig':
    console.log('This animal will go on Noah\'s Ark.');
    break;
  case 'Dinosaur':
    console.log('Left behind.');
    break;
  default:
    console.log('This animal will not.');
} // results This animal will go on Noah's Ark.
```

The switch statement is used to perform different actions based on different conditions. The switch expression is evaluated once. The value of the expression is compared with the values of each case. If there is a match, the associated block of code is executed. When JavaScript reaches a break keyword, it breaks out of the switch block. This will stop the execution of inside the block.

Sources: https://www.w3schools.com/js/js_switch.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch

Switch Statement 2

```
var Animal = 'Fish';
switch (Animal) {
  case 'Cow':
  case 'Giraffe':
  case 'Dog':
  case 'Pig':
    console.log('This animal will go on Noah\'s Ark.');
    break;
  case 'Dinosaur':
    console.log('Left behind.');
    break;
  default:
    console.log('This animal will not.');
} // results This animal will not.
```

If multiple switch cases matches a case value, the first case is selected. If no matching swich cases are found, the program continues to the default label. If no default label is found, the program continues to the statement(s) after the switch. Switch cases use strict comparison (===).

Sources: https://www.w3schools.com/js/js_switch.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch

Functions

A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:

The name of the function.

A list of parameters to the function, enclosed in parentheses and separated by commas.

The JavaScript statements that define the function, enclosed in curly brackets {...}.

The code inside the function will execute when "something" **invokes** (calls) the function:

When an event occurs (when a user clicks a button)

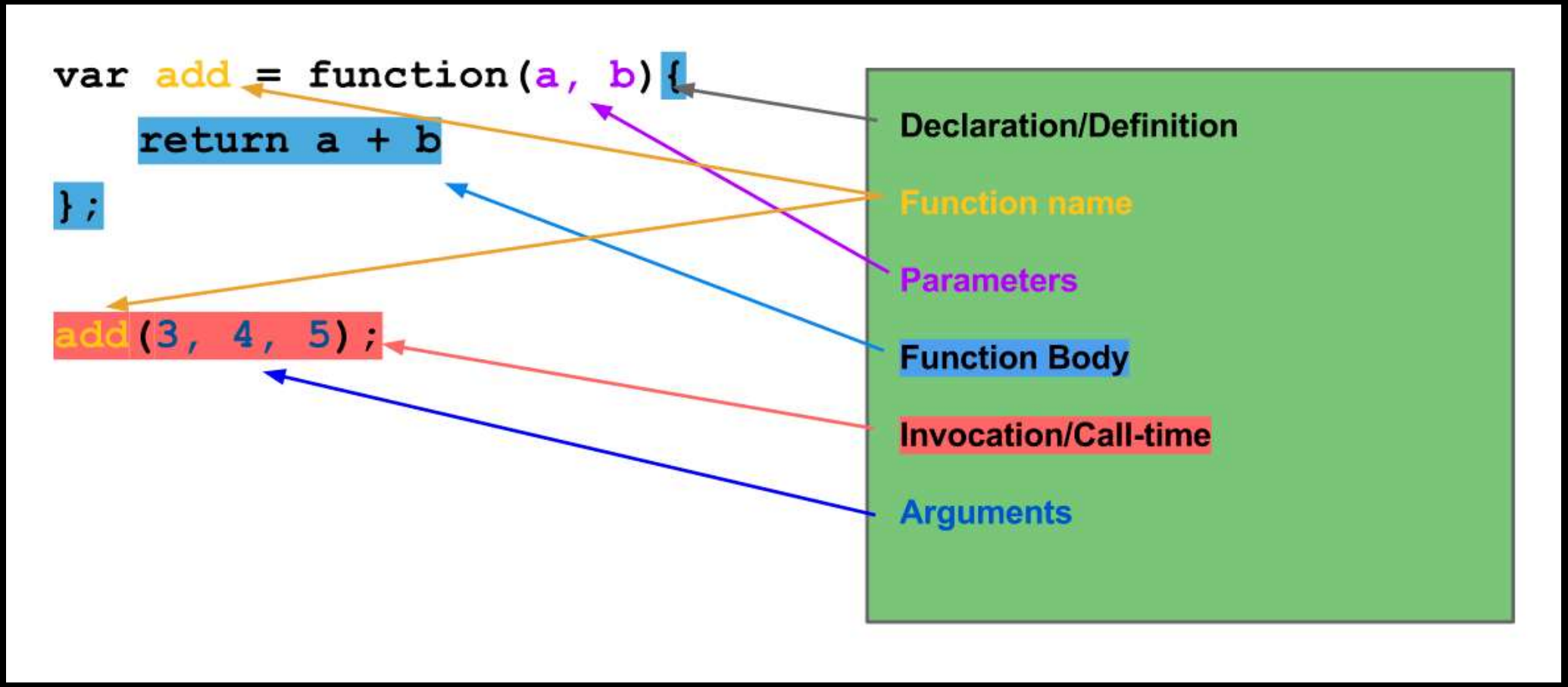
When it is invoked (called) from JavaScript code

Automatically (self invoked)

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
var x = toCelsius(100);
console.log(x)
// 37.77777777777778
```

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions and https://www.w3schools.com/js/js_functions.asp

Anatomy of Functions



Source: https://slides.com/bgando/f2f-final-day-1#/0/1

Terms related to Functions

Definitions

Fn Names

Invocations

Arguments

Parameteres

Fn Bodies

Return Values

Side Effects

```
1  var add = function(a, b){
2      return a + b
3  }
4  add(3,4,5);
```

Example of a function from the previous slide

```
var x = 7
3 + 4 //7
```

Example of a Side Effect

Side Effects: An expression is any valid unit of code that resolves to a value. Every syntactically valid expression resolves to some value but conceptually, there are two types of expressions: with side effects (for example: those that assign value to a variable) and those that in some sense evaluate and therefore resolve to a value. The expression `x = 7` is an example of the first type. This expression uses the = operator to assign the value seven to the variable x. The expression itself evaluates to seven. The code `3 + 4` is an example of the second expression type. This expression uses the + operator to add three and four together without assigning the result, seven, to a variable.

Sources: https://slides.com/bgando/f2f-final-day-1#/0/1, https://www.w3schools.com/js/js_functions.asp and

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

Higher-order Functions versus First-order Functions

In computer science, a higher-order function is a function that does at least one of the following:

- Takes one or more functions as arguments
- Returns a function as its result.

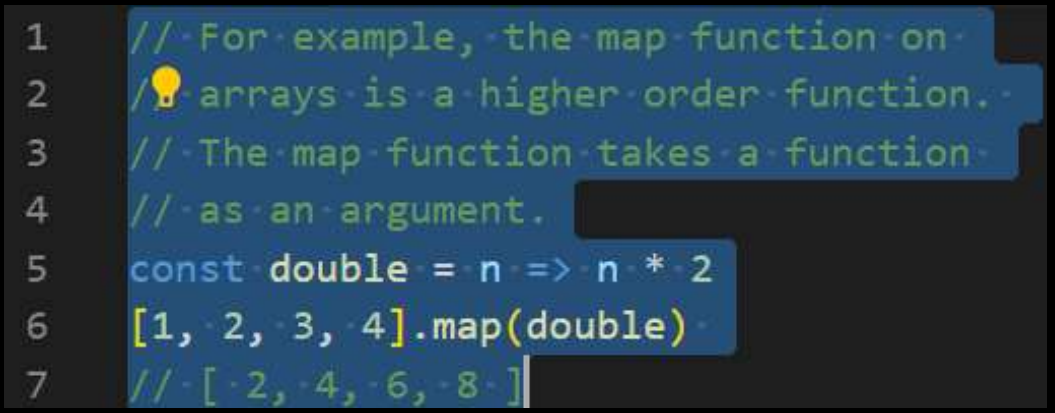
All other functions are first-order functions.

Taking an other function as an argument is often referred as a callback function, because it is called back by the higher-order function. This is a concept that Javascript uses a lot.

Sources: https://dev.to/damcosset/higher-order-functions-in-javascript-4j8b and https://en.wikipedia.org/wiki/Higher-order_function

The map function is one of the many higher-order functions built into the language.

Sort, reduce, filter, forEach are other examples of higher-order functions built into the language.



We can create smaller functions that only take care of one piece of logic. Then, we compose more complex functions by using different smaller functions.

Functions: Hoisting

```
1   // In the case of functions,
2   // only function declarations are hoisted
3   // but not the function expressions.
4   // Function declaration
5   foo();
6   function foo() {
7     console.log('bar'); // logs bar
8   }
9   // Function expression
10  baz();
11  var baz = function() {
12    console.log('bar2');
13  };
14  // logs TypeError: baz is not a function
```

Note! The function on the line 11 is actually an anonymous function (a function without a name). Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

Hoisting is JavaScript's default behavior of moving declarations to the top. Only function declarations are hoisted—but not the function expressions. (the same is with variables)

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types and https://www.w3schools.com/js/js_function_definition.asp

Functions: Return Statement

```
1   function myFunction() {
2     return Math.PI;
3   }
4   myFunction();
5   // returns the value of PI: 3.141592653589793
6   var x = myFunction(4, 3);
7   function myFunction(a, b) {
8     return a * b;
9   }
10  console.log(x);
11  // Function is called on line 6, return value will end up in x
12  // Function returns the product of a and b which is 12
```

The return statement ends function execution and specifies a value to be returned to the function caller.

Syntax: return value; If value is omitted, return returns undefined.

Sources: https://www.w3schools.com/jsref/jsref_return.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/return

Function Invocation

```
// Invoking a Function as a Function:
function myFunction(a, b) {
  return a * b;
}
myFunction(10, 2); // Will return 20
// The function above does not belong to any object.
// But in JavaScript there is always a default global object.
// In HTML the default global object is the HTML page itself,
// so the function above "belongs" to the HTML page.
// In a browser the page object is the browser window.
// The function above automatically becomes a window function.
// myFunction() and window.myFunction() is the same function:
```

Invoking a JavaScript function. The code inside a function is not executed when the function is defined. The code inside a function is executed when the function is invoked. It is common to use the term "call a function" instead of "invoke a function". It is also common to say "call upon a function", "start a function", or "execute a function".

Source: https://www.w3schools.com/js/js_function_invocation.asp

Functions: Arguments and Parameters

A JavaScript function does not perform any checking on parameter values (arguments).

Function parameters are the names listed in the function definition.

Function arguments are the real values passed to (and received by) the function.

Parameter rules:

- JavaScript function definitions do not specify data types for parameters.
- JavaScript functions do not perform type checking on the passed arguments.
- JavaScript functions do not check the number of arguments received.

```
1  function toCelsius(fahrenheit) {
2    return (5/9) * (fahrenheit-32);
3  }
4  toCelsius(100); // 37.77777777777778
```

```
1  function toCelsius(fahrenheit) {
2    return (5/9) * (fahrenheit-32);
3  }
4  var x = toCelsius(100);
5  console.log(x)
6  // 37.77777777777778
```

Sources:
https://www.w3schools.com/js/js_function_parameters.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Functions: Arguments and Parameters

```
1  // You can call any function in JavaScript with an arbitrary amount of arguments;
2  // the language will never complain. It will, however, make all parameters
3  // available via the special variable arguments. arguments looks like an array,
4  // but has none of the array methods:
5  function f() { return arguments }
6  var args = f('a', 'b', 'c');
7  args.length // 3
8  args[0] // read element at index 0 'a'
9  // Let's use the following function to explore how too many or too few parameters
10 // are handled in JavaScript
11 function f(x, y) {
12     console.log(x, y);
13     return toArray(arguments);
14 }
15 // Additional parameters will be ignored (except by arguments):
16 f('a', 'b', 'c') //a b //[ 'a', 'b', 'c' ]
17 // Missing parameters will get the value undefined:
18 f('a') // a undefined // [ 'a' ]
19 f() // undefined undefined // []
```

Source: http://speakingjs.com/es5/ch01.html

Functions: Default and Rest Parameters

```
1  // Parameter Defaults: If a function is called with missing arguments (less than declared),
2  // the missing values are set to: undefined. // Sometimes this is acceptable, but sometimes
3  // it is better to assign a default value to the parameter:
4  function myFunction(x, y) {
5      if (y === undefined) {
6          y = 0;
7      }
8  }
9  // ECMAScript 2015 allows default parameter values in the function declaration:
10 function yourFunction(a=1, b=1) {
11     "function code"
12 }
13 // The rest parameter syntax allows us to represent an indefinite number of arguments as an array.
14 // In the following example, the function multiply uses rest parameters to collect arguments from
15 // the second one to the end. The function then multiplies these by the first argument.
16 function multiply(multiplier, ...theArgs) {
17   return theArgs.map(x => multiplier * x);
18 }
19 var arr = multiply(2, 1, 2, 3);
20 console.log(arr); // [2, 4, 6]
```

Sources: https://www.w3schools.com/js/js_function_parameters.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Functions: Default Parameters and Arity

```
1  // The following is a common pattern for assigning default values to parameters:
2  function pair(x, y) {
3      x = x || 0; // (1)
4      y = y || 0;
5      return [ x, y ];
6  }
7  // In line (1), the || operator returns x if it is truthy (not null, undefined, etc.).
8  // Otherwise, it returns the second operand:
9  pair() // [ 0, 0 ]
10 pair(3) // [ 3, 0 ]
11 pair(3, 5) // [ 3, 5 ]
12 // If you want to enforce an arity (a specific number of parameters),
13 // you can check arguments.length:
14 function pair(x, y) {
15     if (arguments.length !== 2) {
16         throw new Error('Need exactly 2 arguments');
17     }
18 }
19 pair(6,7,8); // 'Need exactly 2 arguments'
```

Source: http://speakingjs.com/es5/ch01.html

Functions: Arguments Object

```javascript
// The arguments object:You can refer to a function's arguments within the function by using the arguments object.
// arguments: An array-like object containing the arguments passed to the currently executing function.
// arguments.callee : The currently executing function.
// arguments.caller : The function that invoked the currently executing function.
// arguments.length: The number of arguments passed to the function.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Using the arguments object:The arguments of a function are maintained in an array-like object. Within a function,
// you can address the arguments passed to it as follows:arguments[i]where i is the ordinal number of the argument,
// starting at 0. So, the first argument passed to a function would be arguments[0]. The total number of arguments
// is indicated by arguments.length. Using the arguments object, you can call a function with more arguments than
// it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be
// passed to the function. You can use arguments.length to determine the number of arguments actually passed to the
// function, and then access each argument using the arguments object.
function myConcat(separator) {
   var result = ''; // initialize list
   var i;
      for (i = 1; i < arguments.length; i++) { // iterate through arguments
      result += arguments[i] + separator;
   }
   return result;
 }
 myConcat(', ', 'red', 'orange', 'blue'); // returns "red, orange, blue,"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions

Functions: Recursion

```
// A function can refer to and call itself. There are three ways within the function body
// for a function to refer to itself:
// The function's name, arguments.callee and an in-scope variable that refers to the function
// A function that calls itself is called a recursive function. In some ways,
// recursion is analogous to a loop. Both execute the same code multiple times, and both require
// a condition (to avoid an infinite loop, or rather, infinite recursion in this case).
function foo(i) {
    if (i < 0)
        return;
    console.log('begin: ' + i);
    foo(i - 1); // the recursive call inside the function
    console.log('end: ' + i);
}
foo(1); // the normal call outside the function
// logs:
// begin: 1
// begin: 0
// end: 0
// end: 1
// In fact, recursion itself uses a stack: the function stack.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Funtions: Local and Global Scope

```javascript
// In JavaScript there are two types of scope: Local scope and Global scope
// JavaScript has function scope: Each function creates a new scope.
// Scope determines the accessibility (visibility) of these variables.
// Variables defined inside a function are not accessible (visible) from outside the function.
// Variables declared within a JavaScript function, become LOCAL to the function.
// Local variables have Function scope: They can only be accessed from within the function.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // code here can NOT use carName
function myFunction() {
    var carName = "Volvo";
    // code here CAN use carName
  }
// Since local variables are only recognized inside their functions,
// variables with the same name can be used in different functions.
// Local variables are created when a function starts, and deleted when the function is completed.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// A variable declared outside a function, becomes GLOBAL.
// A global variable has global scope: All scripts and functions on a web page can access it.
var carName = "Volvo";
    // code here can use carName
function myFunction() {
    // code here can also use carName
}
```

Note! We keep track of the functions being called in JavaScript with a Call stack. One global execution context, multiple function contexts.

Sources: https://www.w3schools.com/js/js_scope.asp and https://drive.google.com/file/d/18e2nJUf2QC3JZ_3N07vRSy8G8xvvjA1X/view

Functions: Local versus Global memory

As soon as we start running our code, we create a global execution context

Thread of execution (parsing and executing the code line after line)

Live memory of variables with data (known as a Global Variable Environment)

When you execute a function you create a new execution context comprising:

1. The thread of execution (we go through the code in the function line by line)

2. A local memory ('Variable environment') where anything defined in the function is stored

Source: https://drive.google.com/file/d/18e2nJUf2QC3JZ_3N07vRSy8G8xvvjA1X/view

```
1  const num = 3;
2  function multiplyBy2 (inputNumber){
3      const result = inputNumber*2;
4      return result; }
5  const name = "Will"
6  const output = multiplyBy2(4);
7  const newOutput = multiplyBy2(10);
```

1. Javascript executes/compiles (runs) the code: Global Execution Context and Global Variable Environment Lines 1, 2 and 5 are saved globally.

2. Running/calling/invoking a function: Local Execution Context and Local Variable Environment Line 6 Local Execution Context is created and when done then popped of the stack.

3. Running/calling/invoking a function: Local Execution Context and Local Variable Environment Line 7 Another Local Execution Context is created.

Functions: Passed by Value versus Passed by Reference

Arguments are Passed by Value

The parameters, in a function call, are the function's arguments.

JavaScript arguments are passed by **value**. The function only gets to know the values, not the argument's locations.

If a function changes an argument's value, it does not change the parameter's original value.

Changes to arguments are not visible (reflected) outside the function.

Source: https://www.w3schools.com/js/js_function_parameters.asp

Objects are Passed by Reference

In JavaScript, object references are values.

Because of this, objects will behave like they are passed by **reference:**

If a function changes an object property, it changes the original value.

Changes to object properties are visible (reflected) outside the function.

Functions: Passed by Value versus Passed by Reference 2

```
function square(number) {
  return number * number;
 }
 // Primitive parameters (such as a number) are passed to functions by value;
 // the value is passed to the function, but if the function changes the value
 // of the parameter, this change is not reflected globally or in the calling function.
 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 // If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object)
 // as a parameter and the function changes the object's properties,  that change is visible
 // outside the function, as shown in the following example:
 function myFunc(theObject) {
   theObject.make = 'Toyota';
 }
 var mycar = {make: 'Honda', model: 'Accord', year: 1998};
 var x, y;
 x = mycar.make; // x gets the value "Honda"
 myFunc(mycar);
 y = mycar.make; // y gets the value "Toyota"
 // (the make property was changed by the function)
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Functions: Closure

Minimize the use of global variables. This includes all data types, objects, and functions. Global variables and functions can be used (and overwritten) by all scripts in the page (and in the window).

In a web page, global variables belong to the window object. Global variables live until the page is discarded, like when you navigate to another page or close the window.

Use local variables instead and learn how to use closures.

JavaScript variables can belong to the local or global scope. Global variables can be made local (private) with closures.

Local variables have short lives. They are created when the function is invoked and deleted when the function is finished.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

A closure is a function having access to the parent scope, even after the parent function has closed.

JavaScript Nested Functions:

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

Sources:
https://www.w3schools.com/js/js_best_practices.asp and
https://www.w3schools.com/js/js_function_closures.asp

Functions: Closure 2

Recipe

1. create your parent function
   ex. the checkscope function
2. define some variables in the parent's local scope (this can be accessed by the child function)
   ex. var innerVar = "local scope"
3. define a function inside the parent function. We call this a child.
   ex. the innerFunc function
4. return that function from inside the parent function
   ex. where it says 'return innerFunc'

```
function checkscope() {
    var innerVar = "local scope";
    function innerFunc() {
        return innerVar;
    };
return innerFunc;
};
```

Source: https://slides.com/bgando/f2f-final-day-2#/

Functions: Closure 3

1. Run parent function and save to a variable. This variable will hold whatever that function RETURNS.

```
8   var test = checkscope();
```

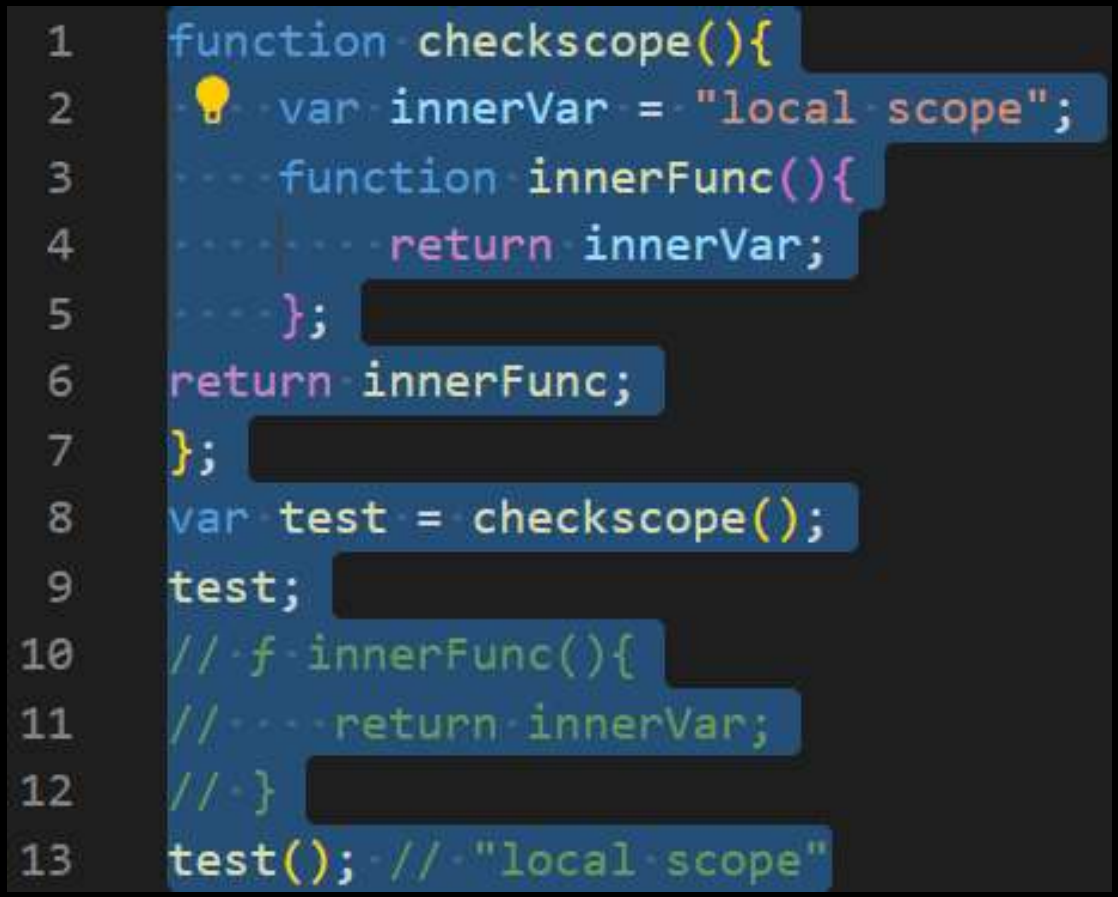
2. Optional: Check what that variable hold as its value. (Hint: It should be the inner function)

```
10  // ƒ innerFunc(){
11  //     return innerVar;
12  // }
```

3. Run the inner function

```
13  test(); // "local scope"
```

Source: https://slides.com/bgando/f2f-final-day-2#/



Arrow Function

An arrow function expression is a syntactically compact alternative to a regular function expression.

It does not own bindings to the this, arguments, super, or new.target keywords.

Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

Since arrow functions do not have their own this, the methods call() and apply() can only pass in parameters. Any this argument is ignored.

An arrow function cannot contain a line break between its parameters and its arrow.

An empty arrow function returns undefined.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions and

What About this keyword?

The handling of this is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of this.

In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the this keyword always represents the object that defined the arrow function.

Source: https://www.w3schools.com/js/js_arrow_function.asp

Arrow Function with this

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript "this"</h2>
<p>This example demonstrate that in Arrow Functions, the "this" keyword represents
    the object that owns the function, no matter who calls the function.</p>
<p>Click the button to execute the "hello" function again, and you will see that
    "this" still represents the window object.</p>
<button id="btn">Click Me!</button>
<p id="demo"></p>
<script>
var hello;
hello = () => {
  document.getElementById("demo").innerHTML += this;
}
// The window object calls the function:
window.addEventListener("load", hello);
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//A button object calls the function:
document.getElementById("btn").addEventListener("click", hello);
</script>
</body>
```

Source: https://www.w3schools.com/js/js_arrow_function.asp

Arrow Function 2

```
1  // Arrow functions can have either a "concise body" or the usual "block body".
2  // In a concise body, only an expression is specified,
3  // which becomes the implicit return value.
4  // In a block body, you must use an explicit return statement.
5  var func = x => x * x;
6  // concise body syntax, implied "return"
7  var func = (x, y) => { return x + y; };
8  // with block body, explicit "return" needed
9  // Arrow functions allow us to write shorter function syntax:
10 hello = () => {
11     return "Hello World!";
12 }
13 // If the function has only one statement, and the statement returns a value,
14 // you can remove the brackets and the return keyword:
15 hello = () => "Hello World!";
16 // If you have parameters, you pass them inside the parentheses:
17 hello = (val) => "Hello " + val;
18 // In fact, if you have only one parameter, you can skip the parentheses as well:
19 hello = val => "Hello " + val;
```

Note! Arrow Functions Return Value by Default: This works only if the function has only one statement.

Sources: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions and https://www.w3schools.com/js/js_arrow_function.asp

Arrow Function 3

```
// More examples:
// An empty arrow function returns undefined
let empty = () => {};
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(() => 'foobar')();
// Returns "foobar"
// (this is an Immediately Invoked Function Expression)
var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
let max = (a, b) => a > b ? a : b;
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Easy array filtering, mapping, ...
var arr = [5, 6, 13, 0, 1, 18, 23];
var sum = arr.reduce((a, b) => a + b);
// 66
var even = arr.filter(v => v % 2 == 0);
// [6, 0, 18]
var double = arr.map(v => v * 2);
// [10, 12, 26, 0, 2, 36, 46]
```

```
// More examples:
// More concise promise chains
promise.then(a => {
    // ...
}).then(b => {
    // ...
});
// Parameterless arrow functions
// that are visually easier to parse
setTimeout( () => {
    console.log('I happen sooner');
    setTimeout( () => {
        // deeper code
        console.log('I happen later');
    }, 1);
}, 1);
```

Note! Line breaks:
An arrow function cannot contain a line break between its parameters and its arrow.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Functions: IIFE

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

Self-Invoking Functions

Function expressions can be made "self-invoking".

A self-invoking expression is invoked (started) automatically, without being called.

Function expressions will execute automatically if the expression is followed by ().

You cannot self-invoke a function declaration.

You have to add parentheses around the function to indicate that it is a function expression.

```
1  (function () {
2      var x = "Hello!!";
3      return x;
4  })();
5  // returns "Hello!!"
6  // The function above
7  // is actually an anonymous
8  // self-invoking function
9  // (function without name)
```

Assigning the IIFE to a variable stores the function's return value, not the function definition itself.

Variable x is not accessible from the outside scope

Source: https://developer.mozilla.org/en-US/docs/Glossary/IIFE and https://www.w3schools.com/js/js_function_definition.asp

Functions Callstack: Concurrency Model and Event Loop

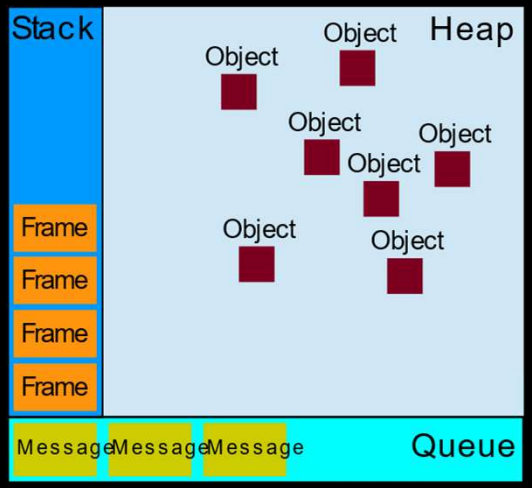
JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks.

Stack: Function calls form a stack of frames.

Heap: Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

Queue: A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. The message is removed from the queue and its corresponding function is called with the message as an input parameter.



Calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Event Loop Runtime Concepts

"Run-to-completion“: Each message is processed completely before any other message is processed.

This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be pre-empted and will run entirely before any other code runs (and can modify data the function manipulates).

A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog. A good practice to follow is to make message processing short and if possible cut down one message into several messages.

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Adding messages: In web browsers, messages are added anytime an event occurs and there is an event listener attached to it. If there is no listener, the event is lost. So a click on an element with a click event handler will add a message—likewise with any other event.

Several runtimes communicating together : A web worker or a cross-origin iframe has its own stack, heap, and message queue. Two distinct runtimes can only communicate through sending messages via the postMessage method. This method adds a message to the other runtime if the latter listens to message events.

Never blocking: A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks.

Event Loop Runtime Concepts: SetTimeout()

```
// The function setTimeout is called with 2 arguments: a message to add to the queue,
// and a time value (optional; defaults to 0). The time value represents the (minimum) delay
// after which the message will actually be pushed into the queue. If there is no other message
// in the queue, and the stack is empty, the message is processed right after the delay.
// However, if there are messages, the setTimeout message will have to wait for other messages to be
// processed. For this reason, the second argument indicates a minimum time—not a guaranteed time.
const s = new Date().getSeconds();
setTimeout(function() {
// prints out "2", meaning that the callback is not called immediately after 500 milliseconds.
  console.log("Ran after " + (new Date().getSeconds() - s) + " seconds");
}, 500)
while (true) {
  if (new Date().getSeconds() - s >= 2) {
    console.log("Good, looped for 2 seconds")
    break;
  }
}
// logs:
// Good, looped for 2 seconds
// Ran after 2 seconds
```

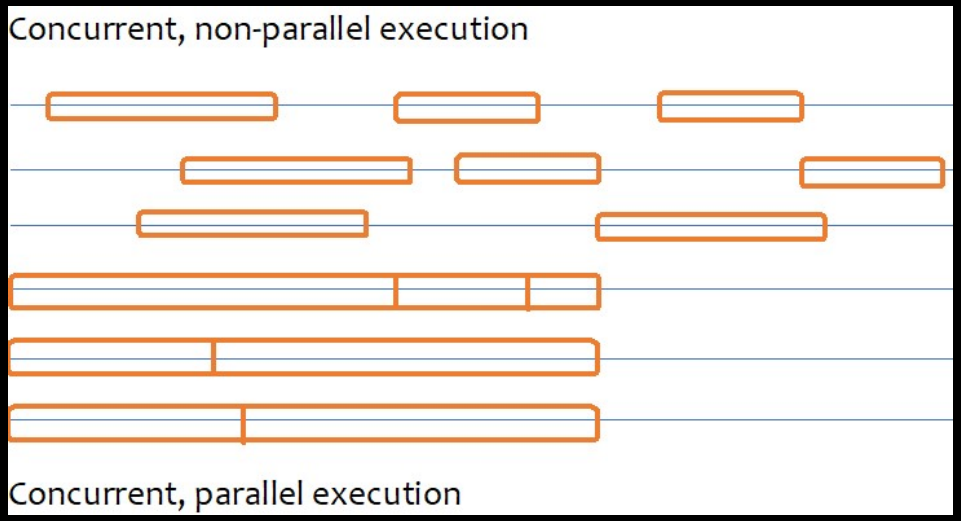
Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

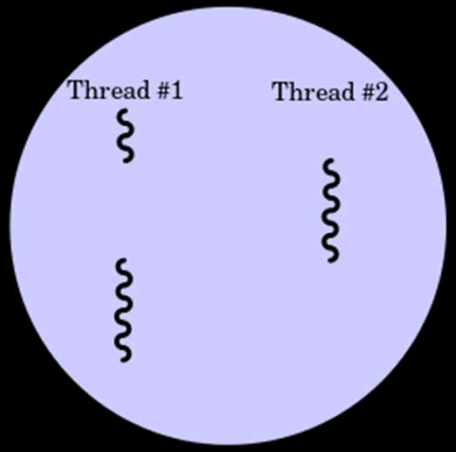
Event Loop Runtime Concepts: SetTimeout() 2

```
// Zero delay doesn't actually mean the call back will fire-off after zero milliseconds.
// Calling setTimeout with a delay of 0 (zero) milliseconds doesn't execute the
// callback function after the given interval. The execution depends on the number of waiting
// tasks in the queue. In the example below, the message ''this is just a message''
// will be written to the console before the message in the callback gets processed,
// because the delay is the minimum time required for the runtime to process the request
// (not a guaranteed time). Basically, the setTimeout needs to wait for all the code for queued
// messages to complete even though you specified a particular time limit for your setTimeout.
(function() {
    console.log('this is the start');
    setTimeout(function cb() {
        console.log('Callback 1: this is a msg from call back');
    }); // has a default time value of 0
    console.log('this is just a message');
})();
// "this is the start"
// "this is just a message"
// "Callback 1: this is a msg from call back"
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Concurrency





Concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system.

Sources: https://books.google.ie/books/about/Introduction_to_Concurrency_in_Programmi.html?id=J5-ckoCgc3IC&redir_esc=y, https://en.wikipedia.org/wiki/Multithreading_(computer_architecture) and https://en.wikipedia.org/wiki/Concurrency_(computer_science)

JavaScript is Single Threaded

A thread is basically a single process that a program can use to complete tasks. Each thread can only do a single task at once:

Task A --> Task B --> Task C

Each task will be run sequentially; a task has to complete before the next one can be started.

Many computers now have multiple cores, so cores can do multiple things at once. Programming languages that can support multiple threads can use multiple cores to complete multiple tasks simultaneously:

Thread 1: Task A --> Task B

Thread 2: Task C --> Task D

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts

JavaScript is single threaded:

JavaScript is traditionally single-threaded. Even with multiple cores, you could only get it to run tasks on a single thread, called the main thread.

After some time, JavaScript gained some tools to help with such problems.

Web workers allow you to send some of the JavaScript processing off to a separate thread, called a worker, so that you can run multiple JavaScript chunks simultaneously.

You'd generally use a worker to run expensive processes off the main thread so that user interaction is not blocked.

Main thread: Task A --> Task C

Worker thread: Expensive task B

Asynchronous Javascript

Asynchronous code:

Web workers are pretty useful, but they do have their limitations. A major one is they are not able to access the DOM — you can't get a worker to directly do anything to update the UI. We couldn't render our 1 million blue circles inside our worker; it can basically just do number crunching.

The second problem is that although code run in a worker is not blocking, it is still basically synchronous. This becomes a problem when a function relies on the results of multiple previous processes to function. Consider the following thread diagrams:

Main thread: Task A --> Task B

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts

In this case, let's say Task A is doing something like fetching an image from the server and Task B then does something to the image like applying a filter to it. If you start Task A running and then immediately try to run Task B, you'll get an error, because the image won't be available yet.

Main thread: Task A --> Task B --> |Task D|

Worker thread: Task C ----------> | |

In this case, let's say Task D makes use of the results of both Task B and Task C. If we can guarantee that these results will both be available at the same time, then we might be OK, but this is unlikely. If Task D tries to run when one of its inputs is not yet available, it will throw an error.

Asynchronous Javascript 2

Browsers allow us to run certain operations asynchronously. Features like promises allow you to set an operation running (e.g. the fetching of an image from the server), and then wait until the result has returned before running another operation:

Main thread: Task A Task B

Promise: |__async operation__|

Since the operation is happening somewhere else, the main thread is not blocked while the async operation is being processed.

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts

Blocking code:

Asynchronous techniques are very useful, particularly in web programming.

When a web app runs in a browser and it executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen.

This is called blocking; the browser is blocked from continuing to handle user input and perform other tasks until the web app returns control of the processor.

Modern software design increasingly revolves around using asynchronous programming, to allow programs to do more than one thing at a time.

Web Worker

What is a Web Worker?

A web worker is a JavaScript running in the background, without affecting the performance of the page.

When executing scripts in an HTML page, the page becomes unresponsive until the script is finished.

A web worker is a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page.

You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background.

```
// Create a Web Worker File: Let's create
// our web worker in an external JavaScript.
// We create a script that counts. The script
// is stored in the "demo_workers.js" file:
var i = 0;
function timedCount() {
  i = i + 1;
  postMessage(i);
  setTimeout("timedCount()",500);
}
timedCount();
// The important part of the code above is the
// postMessage() method - which is used to
// post a message back to the HTML page. Note:
// Normally web workers are not used for such
// simple scripts, but for more CPU intensive tasks.
```

Source: https://www.w3schools.com/html/html5_webworkers.asp

Web Worker 2

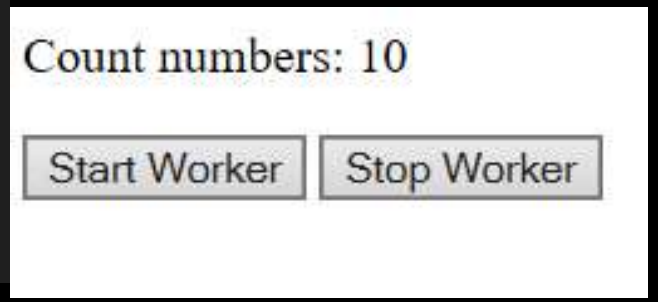
```
// Create a Web Worker Object:Now that we have the web worker file, we need to call
// it from an HTML page.// The following lines checks if the worker already exists,
// if not - it creates a new web worker object and runs the code in "demo_workers.js":
if (typeof(w) == "undefined") {
  w = new Worker("demo_workers.js");
}
// Then we can send and receive messages from the web worker.
// Add an "onmessage" event listener to the web worker.
w.onmessage = function(event){
  document.getElementById("result").innerHTML = event.data;
};
// When the web worker posts a message, the code within the event listener is executed.
// The data from the web worker is stored in event.data.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Terminate a Web Worker: When a web worker object is created, it will continue to listen
// for messages (even after the external script is finished) until it is terminated.
// To terminate a web worker, and free browser/computer resources, use the terminate() method:
w.terminate();
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Reuse the Web Worker: If you set the worker variable to undefined, after it has been
// terminated, you can reuse the code:
w = undefined;
```

Source: https://www.w3schools.com/html/html5_webworkers.asp

Web Worker 3

```
<!DOCTYPE html>
<html>
<body>
<p>Count numbers: <output id="result"></output></p>
<button onclick="startWorker()">Start Worker</button>
<button onclick="stopWorker()">Stop Worker</button>
<script>
var w;
function startWorker() {
  if (typeof(Worker) !== "undefined") {
    if (typeof(w) == "undefined") {
      w = new Worker("demo_workers.js");
    }
    w.onmessage = function(event) {
      document.getElementById("result").innerHTML = event.data;
    };
  } else {
    document.getElementById("result").innerHTML = "Sorry! No Web Worker support.";
  }
}
function stopWorker() {
  w.terminate();
  w = undefined;
}
</script>
</body>
</html>
```

Note ! Chrome or Firefox don't let you load web workers when running scripts from a local file and the counting will not work. Open the html file with Edge and then the counting works fine.



Source: https://www.w3schools.com/html/html5_webworkers.asp

Promises

A promise is an object that represents an intermediate state of an operation — in effect, a promise that a result of some kind will be returned at some point in the future. There is no guarantee of exactly when the operation will complete and the result will be returned, but there *is* a guarantee that when the result is available, or the promise fails, the code you provide will be executed in order to do something else with a successful result, or to gracefully handle a failure case.

You are less interested in the amount of time an async operation will take to return its result and more interested in being able to respond to it being returned, whenever that is. And of course, it's nice that it doesn't block the rest of the code execution.

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises#Promise_terminology_recap

```
1  // At their most basic, promises are similar
2  // to event listeners, but with a few differences:
3  // A promise can only succeed or fail once. It cannot
4  // succeed or fail twice and it cannot switch from
5  // success to failure or vice versa once the operation
6  // has completed. If a promise has succeeded or failed
7  // and you later add a success/failure callback,
8  // the correct callback will be called, even though
9  // the event took place earlier.
10 chooseToppings()
11 .then(function(toppings) {
12     return placeOrder(toppings);
13   })
14   .then(function(order) {
15     return collectOrder(order);
16   })
17   .then(function(pizza) {
18     eatPizza(pizza);
19   })
20   .catch(failureCallback);
```

Promises 2

When a promise is created, it is neither in a success or failure state. It is said to be pending.

When a promise returns, it is said to be resolved.

A successfully resolved promise is said to be fulfilled. It returns a value, which can be accessed by chaining a .then() block onto the end of the promise chain. The executor function inside the .then() block will contain the promise's return value.

An unsuccessful resolved promise is said to be rejected. It returns a reason, an error message stating why the promise was rejected. This reason can be accessed by chaining a .catch() block onto the end of the promise chain.

Sources: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises and .../Web/API/Fetch_API/Using_Fetch

```
1  // A basic fetch request is really simple
2  // to set up. Have a look at the following code:
3  fetch('http://example.com/movies.json')
4    .then((response) => {
5      return response.json();
6    })
7    .then((data) => {
8      console.log(data);
9    });
10 // Here we are fetching a JSON file across
11 // the network and printing it to the console.
```

Here we are fetching a JSON file across the network and printing it to the console. The simplest use of fetch() takes one argument — the path to the resource you want to fetch — and returns a promise containing the response (a response object). This is just an HTTP response, not the actual JSON.

The basics of Async/Await

```
1  // The async keyword:First of all we have the async keyword, which you put in front of
2  // a function declaration to turn it into an async function. An async function is a
3  // function which knows how to expect the possibility of the await keyword being used
4  // to invoke asynchronous code.Type the following lines into your browser's JS console:
5  function hello() { return "Hello" };
6  hello(); // The function returns "Hello" — nothing special, right?
7  // But what if we turn this into an async function? Try the following:
8  async function hello() { return "Hello" };
9  hello(); // Invoking the function now returns a promise. This is one of the traits
10 // of async functions — their return values are guaranteed to be converted to promises.
```

```
> function hello() { return "Hello"
  hello();
< "Hello"
```

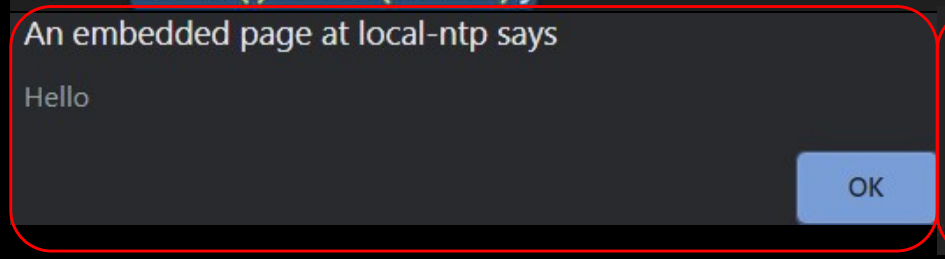
```
> async function hello() { return "Hello"
  hello();
< ▶ Promise {<resolved>: "Hello"}
```

```
> async function hello() { return "Hello" };
  hello();
< ▼ Promise {<resolved>: "Hello"} i
    ▶ __proto__: Promise
      [[PromiseStatus]]: "resolved"
      [[PromiseValue]]: "Hello"
```

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

The basics of Async/Await

```
1  // The await keyword: The real advantage of async functions becomes
2  // apparent when you combine it with the await keyword — in fact,
3  // await only works inside async functions. This can be put in front of
4  // any async promise-based function to pause your code on that line until
5  // the promise fulfills, then return the resulting value. In the meantime,
6  // other code that may be waiting for a chance to execute gets to do so.
7  // You can use await when calling any function that returns a Promise,
8  // including web API functions.
9  // Here is a trivial example:
10 async function hello() {
11   return greeting = await Promise.resolve("Hello");
12 };
13 hello().then(alert);
```




```
> async function hello() {
    return greeting = await Promise.resolve("Hello");
  };
  hello().then(alert);
< ▼Promise {<resolved>: undefined} i
```

Source: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

Callbacks

```
// Callbacks are used in the implementation of languages such as JavaScript,
// including support of JavaScript functions as callbacks through js-ctypes
// and in components such as addEventListener. A native example of a callback
// can be written without any complex code:
function calculate(num1, num2, callbackFunction) {
    return callbackFunction(num1, num2);
}
function calcProduct(num1, num2) {
    return num1 * num2;
}
function calcSum(num1, num2) {
    return num1 + num2;
}
alert(calculate(5, 15, calcProduct)); // alerts 75, the product of 5 and 15
alert(calculate(5, 15, calcSum)); // alerts 20, the sum of 5 and 15
// First a function calculate is defined with a parameter intended for callback:
// callbackFunction. Then a function that can be used as a callback to calculate
// is defined, calcProduct. Other functions may be used for callbackFunction,
// like calcSum. In this example, calculate() is invoked twice, once with calcProduct
// as a callback and once with calcSum. The functions return the product and sum,
// respectively, and then the alert will display them to the screen.
```

Source: https://en.wikipedia.org/wiki/Callback_(computer_programming)

Promises versus Callback Hell

Promises are essentially a returned object to which you attach callback functions. Promises are specifically made for handling async operations.

You can chain multiple async operations together using multiple .then() operations, passing the result of one into the next one as an input. Using callbacks ends up with a messy "pyramid of doom" (a callback hell).

Promise callbacks are always called in the strict order they are placed in the event queue. All errors are handled by a single .catch() block at the end of the block.

Promises avoid inversion of control when passing a callback to a third-party library.

Sources: https://developer.mozilla.org/en-US/docs/Glossary/Callback_function and .../Learn/JavaScript/Asynchronous/Introducing

```
// A callback function is a function passed
// into another function as an argument,
// which is then invoked inside the outer function
// to complete some kind of routine or action.
// This example is a synchronous callback,
// as it is executed immediately.
function greeting(name) {
    alert('Hello ' + name);
}
function processUserInput(callback) {
    var name = prompt('Please enter your name.');
    callback(name);
}
processUserInput(greeting);
// callbacks are often used to continue code
// execution after an asynchronous operation has
// completed — these are called asynchronous
// callbacks. A good example is the callback
// functions executed inside a .then() block
// chained onto the end of a promise after that
// promise fulfills or rejects. This structure
// is used in many modern web APIs, such as fetch().
```

Functions: Callstack 2

A call stack is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc.

When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.

Any functions that are called by that function are added to the call stack further up and run where their calls are reached.

When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing.

If the stack takes up more space than it had assigned to it, it results in a "stack overflow" error.

```
1   // function greeting() {
2   //   // [1] Some codes here
3   //   sayHi();
4   //   // [2] Some codes here
5   // }
6   function sayHi() {
7     return "Hi!";
8   }
9   // Invoke the `greeting` function
10  greeting();
11  // [3] Some codes here
```

Source: https://developer.mozilla.org/en-US/docs/Glossary/Call_stack

Functions: Callstack 3

```
// The code from previous page would be executed like this:
// Ignore all functions, until it reaches the
// greeting() function invocation.
// Push the greeting() function to the call stack list.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Execute all lines of code inside the greeting() function.
// Get to the sayHi() function invocation.
// Push the sayHi() function to the call stack list.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Execute all lines of code inside the sayHi() function,
// until reaches its end. Return execution to the line
// that invoked sayHi() and continue executing the rest
// of the greeting() function. Pop the sayHi() function
// from our call stack list.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// When everything inside the greeting() function return to
// its invoking line to continue executing the rest
// of the JS code. Pop the greeting() function
// from the call stack list.
```

```
/* Call stack list:
- greeting */
// ~~~~~~~~~~~~~~~~~~~
/* Call stack list:
- greeting
- sayHi */
// ~~~~~~~~~~~~~~~~~~~
/* Call stack list:
- greeting */
// ~~~~~~~~~~~~~~~~~~~
/* Call stack list:
EMPTY */
```

Source: https://developer.mozilla.org/en-US/docs/Glossary/Call_stack

Functions: Callstack 4

The order in which elements come off a stack gives rise to its alternative name, LIFO (last in, first out).

push
greeting()
to callstack

Callstack
greeting()
main()

push
sayHi()
to callstack

Callstack
sayHi()
greeting()
main()

pop
sayHi()
from callstack

Callstack
greeting()
main()

pop
greeting()
from callstack

Callstack
EMPTY

In computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection and pop, which removes the most recently added element that was not yet removed. When the script runs, the JavaScript engine places the global execution context (denoted by main() or global() function on the call stack.

Sources: https://developer.mozilla.org/en-US/docs/Glossary/Call_stack, https://en.wikipedia.org/wiki/Stack_(abstract_data_type) and https://www.javascripttutorial.net/javascript-call-stack/

Other Technologies Complementing JavaScript

There are more technologies than just HTML5 that complement JavaScript and make the language more useful:

Libraries:

JavaScript has an abundance of libraries, which enable you to complete tasks ranging from parsing JavaScript (via Esprima) to processing and displaying PDF files (via PDF.js).

Node.js:

The Node.js platform lets you write server-side code and shell scripts (build tools, test runners, etc.).

NoSQL databases: (such as CouchDB and MongoDB)

These databases tightly integrate JSON and JavaScript.

Source: http://speakingjs.com/es5/ch02.html

JSON (JavaScript Object Notation):

JSON is a data format rooted in JavaScript that has become popular for exchanging data on the Web (e.g., the results of web services).

Javascript tools:

JavaScript is getting better build tools (e.g., Grunt) and test tools (e.g., mocha). Node.js makes it possible to run these kinds of tools via a shell (and not only in the browser).

The complexity and dynamism of web development make this space a fertile ground for innovation. Two open source examples are Brackets and Light Table.

There is much JavaScript-related innovation (e.g., the asm.js and ParallelJS, Microsoft's TypeScript, etc.).

The Javascript language is evolving steadily.

JSON

JavaScript Object Notation

JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation.

Exchanging Data:

When exchanging data between a browser and a server, the data can only be text.

JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server.

We can also convert any JSON received from the server into JavaScript objects.

This way we can work with the data as JavaScript objects, with no complicated parsing and translations.

The JSON syntax is a subset of the JavaScript syntax.

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON names require double quotes. JavaScript names do not. The file type for JSON files is ".json".

Source: https://www.w3schools.com/js/js_json_intro.asp

JSON

```
// This JSON syntax defines an employees object: an array of 2 employee records (objects):
// JSON arrays are written inside square brackets.Just like in JavaScript,
// an array can contain objects:
{
  "employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"}
  ]
}
// The JSON format is syntactically identical to the code for creating JavaScript objects.
// Because of this similarity, a JavaScript program can easily convert JSON data
// into native JavaScript objects.
// JSON data is written as name/value pairs, just like JavaScript object properties.
// A name/value pair consists of a field name (in double quotes), followed by a colon,
// followed by a value:
"firstName":"John"
// JSON objects are written inside curly braces.
// Just like in JavaScript, objects can contain multiple name/value pairs:
{"firstName":"John", "lastName":"Doe"}
```

Source: https://www.w3schools.com/js/js_json.asp

JSON Data Types

JSON values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- Null

Objects and Arrays: Property names must be double-quoted strings; trailing commas are forbidden.

JSON values **cannot** be one of the following data types:

- a function
- a date
- undefined

Numbers: Leading zeros are prohibited. A decimal point must be followed by at least one digit. NaN and Infinity are unsupported.

Sources: https://www.w3schools.com/js/js_json_datatypes.asp and https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

JSON Data Types

```
// Valid Data Types:
// JSON Strings: Strings in JSON must be written in double quotes.
{ "name":"John" }
// JSON Numbers: Numbers in JSON must be an integer or a floating point.
{ "age":30 }
// JSON Objects: Values in JSON can be objects.
{
    "employee":{ "name":"John", "age":30, "city":"New York" }
    }
// JSON Arrays: Values in JSON can be arrays.
{
    "employees":[ "John", "Anna", "Peter" ]
    }
// JSON Booleans: Values in JSON can be true/false.
{ "sale":true }
// JSON null: Values in JSON can be null.
{ "middlename":null }
```

Source: https://www.w3schools.com/js/js_json_datatypes.asp

JSON.parse()

```
// The JSON.parse() method parses a JSON string, constructing the JavaScript value
// or object described by the string. An optional reviver function can be provided
// to perform a transformation on the resulting object before it is returned.
const json = '{"result":true, "count":42}';
const obj = JSON.parse(json);
console.log(obj.count); // expected output: 42
console.log(obj.result); // expected output: true
// Syntax: JSON.parse(text[, reviver])
JSON.parse('{}');              // {}
JSON.parse('true');            // true
JSON.parse('"foo"');           // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('null');            // null
// JSON.parse() does not allow trailing commas
JSON.parse('[1, 2, 3, 4, ]'); //will throw a SyntaxError
// JSON.parse() does not allow single quotes
JSON.parse("{'foo': 1}"); // will throw a SyntaxError
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse

JSON.parse()

```
> JSON.parse("{"foo": 1}");
Uncaught SyntaxError: missing ) after argument list
> JSON.parse("{'foo': 1}");
▶Uncaught SyntaxError: Unexpected token ' in JSON at position 1
    at JSON.parse (<anonymous>)
    at <anonymous>:1:6
> JSON.parse('{"foo": 1}');
<· {foo: 1}
```

Test conclusion:

JSON from a server is a string with single quotes (the bottom working example). Double quotes are not allowed (the top not working example).

JSON.parse() does not allow single quotes (the middle not working example).

JSON.stringify()

```
// The JSON.stringify() method converts a JavaScript object or value to a JSON string,
// optionally replacing values if a replacer function is specified
// or optionally including only the specified properties if a replacer array is specified.
var obj = { name: "John", age: 30, city: "New York" };
var myJSON = JSON.stringify(obj);
console.log(myJSON); // logs {"name":"John","age":30,"city":"New York"}
// Syntax:JSON.stringify(value[, replacer[, space]])
JSON.stringify({});                  // '{}'
JSON.stringify(true);                // 'true'
JSON.stringify('foo');               // '"foo"'
JSON.stringify([1, 'false', false]); // '[1,"false",false]'
JSON.stringify([NaN, null, Infinity]); // '[null,null,null]'
JSON.stringify({ x: 5 });            // '{"x":5}'
// Properties of non-array objects are not guaranteed
// to be stringified in any particular order.
// Do not rely on ordering of properties within the same object within the stringification.
```

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify and https://www.w3schools.com/js/js_json_stringify.asp

JSON versus XML

Both JSON and The Extensible Markup Language (XML) can be used to receive data from a web server. The biggest difference is: XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

XML is much more difficult to parse than JSON. JSON is parsed into a ready-to-use JavaScript object.

JSON is Like XML because:

Both JSON and XML are "self describing" (human readable)

Both JSON and XML are hierarchical (values within values)

Both JSON and XML can be parsed and used by lots of programming languages

Both JSON and XML can be fetched with an XMLHttpRequest

JSON is Unlike XML because:

JSON doesn't use end tag. JSON is shorter. JSON is quicker to read and write. JSON can use arrays.

For AJAX applications, JSON is faster and easier than XML:

Using XML:

Fetch an XML document. Use the XML DOM to loop through the document. Extract values and store in variables.

Using JSON:

Fetch a JSON string. JSON.parse the JSON string.

Source:
https://www.w3schools.com/js/js_json_xml.asp

AJAX

AJAX = Asynchronous JavaScript and XML. AJAX is not a programming language. AJAX just uses a combination of:

A browser built-in XMLHttpRequest object (to request data from a web server).

JavaScript and HTML DOM (to display or use the data)

AJAX is a developer's dream, because you can:

Read data from a web server - after the page has loaded

Update a web page without reloading the page

Send data to a web server - in the background

AJAX is a misleading name. AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text.

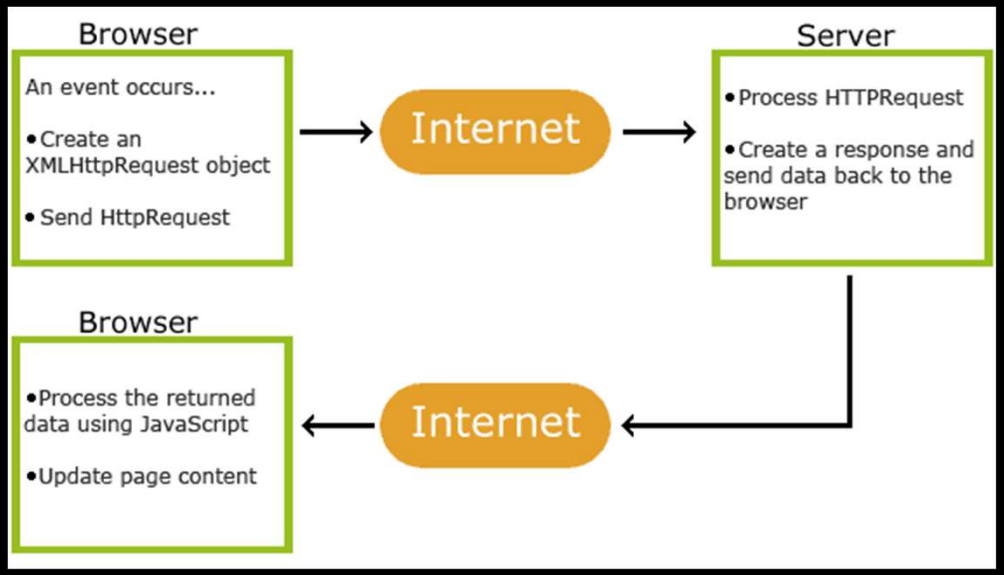
AJAX is an approach to using a number of existing technologies together, including HTML or XHTML, CSS, JavaScript, DOM, XML, XSLT, and most importantly the XMLHttpRequest object.

Although X in AJAX stands for XML, JSON is used more than XML nowadays because of its many advantages such as being lighter and a part of JavaScript.

Both JSON and XML are used for packaging information in the AJAX model.

Sources:
https://www.w3schools.com/js/js_ajax_intro.asp and https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

AJAX 2



How AJAX Works:

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

Source: https://www.w3schools.com/js/js_ajax_intro.asp

1. An event occurs in a web page (the page is loaded, a button is clicked)
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action (like page update) is performed by JavaScript

Full Stack JavaScript versus Full Stack

The idea of "Full Stack JavaScript" is that all software in a web application, both client side and server side, should be written using JavaScript only.

Server-side software (Back End):

PHP, ASP, C++, C#, Java, Python, Ruby, REST, GO, SQL, MongoDB, Firebase.com, Sass, Less, Parse.com and PaaS (Azure and Heroku).

Client-side software (Front End):

HTML, CSS, Bootstrap, W3.CSS, JavaScript, ES5, HTML DOM, JSON, XML, jQuery, Angular, React, Backbone.js, Express.js, Ember.js, Redux, Storybook, GraphQL, Meteor.js, Grunt and Gulp

A full stack web developer is a person who can develop both client and server software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a browser (like using JavaScript, jQuery, Angular, or Vue)
- Program a server (like using PHP, ASP, Python, or Node.js)
- Program a database (like using SQL, SQLite, or MongoDB)

Source: https://www.w3schools.com/whatis/whatis_fullstack_js.asp

Full Stack

Popular stacks related to full stack:

LAMP stack: JavaScript - Linux - Apache - MySQL - PHP
WAMP stack: JavaScript - Windows - Apache - MySQL - PHP
LEMP stack: JavaScript - Linux - Nginx - MySQL - PHP
MEAN stack: JavaScript - MongoDB - Express - AngularJS - Node.js
Django stack: JavaScript - Python - Django - MySQL
Ruby on Rails: JavaScript - Ruby - SQLite - Rails

Front end: It is the visible part of website or web application which is responsible for user experience.

Back end: It refers to the server-side development of web application or website with a primary focus on how the website works.

Sources: https://www.w3schools.com/whatis/whatis_fullstack.asp and https://www.geeksforgeeks.org/what-is-full-stack-development/

Node.js

Node.js is a cross-platform open-source JavaScript runtime environment that allows developers to build server-side and network applications with JavaScript.

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others.

Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)

Node.js files have extension ".js". The official Node.js website (https://nodejs.org) has installation instructions for Node.js.

The following Javascript concepts are key to understand asynchronous programming, which is one fundamental part of Node.js:

Asynchronous programming, callbacks, timers, promises, async and await, closures and the event loop.

REPL stands for read-eval-print loop and basically means command line. To use it, you must first start Node.js from an operating system command line, via the command node.

Sources: https://nodejs.dev/, https://v8.dev/, https://developer.mozilla.org/fi/docs/WebAssembly/Concepts (...docs/Glossary/Node.js) , https://www.w3schools.com/nodejs/nodejs_intro.asp and https://exploringjs.com/impatient-js/ch_console.html#trying-out-code

Node.js and MongoDB

Node.js can be used in database applications. One of the most popular NoSQL database is MongoDB.

To access a MongoDB database with Node.js. Download abd install the official MongoDB driver, open the Command Terminal and execute the following:

```
C:\Users\Your Name>npm install mongodb
```

Now you have downloaded and installed a mongodb database driver.

Node.js can use this module to manipulate MongoDB databases:

```
var mongo = require('mongodb');
```

Sources: https://www.mongodb.com/ and https://www.w3schools.com/nodejs/nodejs_mongodb.asp

MongoDB is a document database, which means it stores data in JSON-like documents. Is much more expressive and powerful than the traditional row/column model.

Rich JSON Documents:

Supports arrays and nested objects as values.

Allows for flexible and dynamic schemas.

Powerful query language

Allows you to filter and sort by any field, no matter how nested it may be within a document.

Support for aggregations and other modern use-cases such as geo-based search, graph search, and text search.

Queries are themselves JSON. No more concatenating strings to dynamically generate SQL queries.

Libraries

The ECMAScript Internationalization API helps with tasks related to internationalization: collation (sorting and searching strings), number formatting, and date and time formatting.

Underscore.js complements JavaScript's relatively sparse standard library with tool functions for arrays, objects, functions, and more. As Underscore predates ECMAScript 5, there is some overlap with the standard library.

Lo-Dash is an alternative implementation of the Underscore.js API, with a few additional features.

XRegExp is a regular expression library with several advanced features such as named captures and free-spacing. (which allows you to spread out a regular expression across multiple lines and document per line).

Source: http://speakingjs.com/es5/ch30.html

Shims Versus Polyfills:

Shims and polyfills are libraries that retrofit newer functionality on older JavaScript engines:

A shim is a library that brings a new API to an older environment, using only the means of that environment.

A polyfill is a shim for a browser API. It typically checks if a browser supports an API. If it doesn't, the polyfill installs its own implementation. That allows you to use the API in either case. The term polyfill comes from a home improvement product; according to Remy Sharp:

Polyfilla is a UK product known as Spackling Paste in the US. With that in mind: think of the browsers as a wall with cracks in it. These [polyfills] help smooth out the cracks and give us a nice smooth wall of browsers to work with.

jQuery Library versus JavaScript

```
$(document).ready(function(){
    alert("Hello World!");
    $("#blackBox").hide();
});
// The above jQuery code carries out the same
// function as the following Javascript code:
window.onload = function() {
    alert( "Hello World!" );
    document.getElementById("blackBox").style.display = "none";
};
```

jQuery was created in 2006 by John Resig. It was designed to handle browser incompatibilities and to simplify HTML DOM manipulation, event handling, animations, and Ajax. jQuery has been the most popular JavaScript library in the world. It is possible to only use vanilla (basic) Javascript instead of jQuery.

jQuery uses a format, $(selector).action() to assign an element(s) to an event. To explain it in detail, $(selector) will call jQuery to select selector element(s) and assign it to an event API called .action().

Sources: https://developer.mozilla.org/en-US/docs/Glossary/jQuery and https://www.w3schools.com/js/js_jquery_selectors.asp

API (Application Programming Interface)

An API (Application Programming Interface) is a set of features and rules that exist inside the application enabling interaction with it through software . The API can be seen as a simple contract (the interface) between the application offering it and other items, such as third party software or hardware.

In Web development, an API is generally a set of code features (e.g. methods, properties, events, and URLs) that a developer can use in their apps for interacting with components of a user's web browser, or other software/hardware on the user's computer, or third party websites and services.

Sources: https://developer.mozilla.org/en-US/docs/Glossary/API, … en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction and …/en-US/docs/Web/API

Web APIs are typically used with JavaScript, although this doesn't always have to be the case.

Common browser APIs:

APIs for manipulating documents: DOM API

APIs that fetch data from the server: Fetch API

APIs for drawing and manipulating graphics: WebGL API

Audio and Video APIs: GetUserMedia API, HTMLMediaElement

Device APIs: Vibration API, Notifications API

Client-side storage APIs: Web Storage API, IndexedDB API

Common third-party APIs:

Twitter API, Google Maps API , YouTube API

Fetch API versus AJAX

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using AJAX XMLHttpRequest. Fetch provides a better alternative that can be easily used by other technologies such as Service Workers. Fetch also provides a single logical place to define other HTTP-related concepts such as Cross-Origin Resource Sharing (CORS) and extensions to HTTP.

```
1  // A basic fetch request is really simple
2  // to set up. Have a look at the following code:
3  fetch('http://example.com/movies.json')
4    .then((response) => {
5      return response.json();
6    })
7    .then((data) => {
8      console.log(data);
9    });
10 // Here we are fetching a JSON file across
11 // the network and printing it to the console.
```

Sources: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API and https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Will Sentance and Douglas Crockford

Will Sentence is a CEO at Codesmith

Frontend Masters teacher

JavaScript: The Hard Parts cources on Frontend Masters

Learn Javascript: https://csx.codesmith.io/ and watch videos https://www.youtube.com/channel/UCAU_6P-M2VHKePIpu5736ag/videos

Source: http://willsentance.com/

Douglas Crockford popularized the data format JSON (JavaScript Object Notation) and has developed various JavaScript related tools such as JSLint and JSMin. Frontend Masters teacher

Learn Javascript and watch videos: https://www.crockford.com/2020.html

Sources: https://en.wikipedia.org/wiki/Douglas_Crockford and https://github.com/douglascrockford?tab=repositories

Brian Holt and Bianca Gandolfo

https://github.com/btholt

Brian Holt is a cloud developer advocate at Microsoft. Azure is a pretty cool place to deploy your code. I write a lot of code demos and help with some open source libraries.

Frontend Masters teacher. You can find a lot of his talks about programming in Youtube.

Source: https://btholt.github.io/intro-to-web-dev-v2/

https://github.com/bgando

Bianca Gandolfo is a JavaScript trainer @Bitovi.

Frontend Masters teacher. You can find a lot of her talks about Javascript programming in Youtube.

Sources:

https://btholt.github.io/intro-to-web-dev-v2/

https://slides.com/bgando/f2f-final-day-2#/

Jem Young and Jafar Husain

Jem Young is a a software Engineer at Netflix. Frontend Masters teacher.

You can find a lot of his talks about Javascript programming in Youtube.

Ducking in Subways: https://blog.jemyoung.com/

https://frontendhappyhour.com/

Sources: https://github.com/young, https://jemyoung.com/about/ and https://twitter.com/JemYoung

Jafar Husain is an engineer at Facebook. Frontend Masters teacher.

You can find a lot of his talks about Javascript asyncronous programming in Youtube.

A highly-rated speaker. He has trained hundreds of developers to build event-driven systems in Javascript.

Sources: https://github.com/jhusain, https://twitter.com/jhusain and https://egghead.io/instructors/jafar-husain

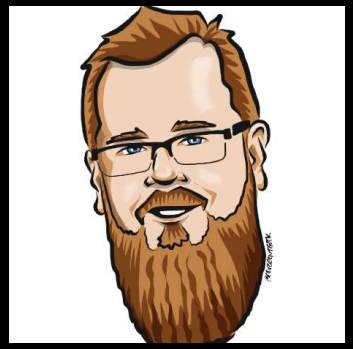
Axel Rauschmayer and Kyle Simpson

Axel Rauschmayer specializes in JavaScript and web development. In 2010, he received a Ph.D. in Informatics.

Since 2011, he has been blogging at www.2ality.com and written several books on JavaScript.

Learn Javascript: Speaking Javascript online books at https://exploringjs.com/

Source: http://dr-axel.de/



Kyle Simpson is an evangelist of the Open Web, passionate about all things JavaScript.

He writes, speaks, teaches, and contributes to OSS (Open-source software). Learn Javascript: You Don't Know JS Yet online books at https://github.com/getify/You-Dont-Know-JS/

Frontend Masters teacher

Source: https://github.com/getify

Learn more Javascript

Search Youtube JS videos with these names:

Tony Alicea, Kyle Simpson, Brian Holt, Bianca Gandolfo, Axel Rauschmeyer, Will Sentence, Douglas Crockford, Jem Young, Bucky Robert, Jafar Husain and Caleb Curry.

Search in Youtube these channels with JS:

Caleb Curry, Certified fresh events, Codesmith, JSConf, thenewBoston, Tony Alicea and Telusko.

Read online books:

You Don't Know JS Yet online books at https://github.com/getify/You-Dont-Know-JS/

Speaking Javascript online books at https://exploringjs.com/

Study online tutorials:

W3Schools Online Web Tutorials https://www.w3schools.com/js/default.asp

Mozilla Developer Networ (MDN) web docs which is the official source for Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript

Look answers to Javascript questions from:

Stackoverflow https://stackoverflow.com/

Study blogs:

www.2ality.com

Study the latest version of Javascript:

https://tc39.es/ecma262