WIKIBOOKS

JavaScript/Print version

JavaScript

Contents

Introduction

JS is a programming language that implements the international standard ECMAScript. It is based on the following concepts.

Dynamic data types

JS knows some *primitive data types* (Number, String, Boolean, BigInt, Symbol, Undefined, Null) and diverse derivates of the data type *object* (Array, Date, Error, Function, RegExp). [1] [2] If a variable exists, its type is clearly defined. But the type can be changed at any time by assigning a value of a different type to the variable, e.g.: the code fragment `let x; x = 'Some text'; x = 2; x = [10, 11, 12];` is perfectly correct. It will not create a compile-time or run-time error. Only the type of the variable `x` changes from *Undefined* to *String* to *Number* and lastly to *Object/Array*.

(Note: JSON is a text-based data format, not a data type. As such, it's language-independent. It uses the JavaSript object syntax.)

Functional programming

Functions are *first-class citizens* similar to variables. They can be assigned to variables, passed as arguments to other functions, or returned from functions. The code fragment `function sayHello() {return 'Good morning'};let x = sayHello; console.log(x());` creates a function *sayHello*, assigns it to the variable *x*, and executes it by calling *x()*.

Object-orientated programming

JS supports object-oriented programming and inheritance through prototypes. A prototype is an object which can be cloned and extended. Doing so, a *prototype chain* (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes) arises. This differs from other OO-languages, e.g. Java, which uses *classes* for object-oriented features like inheritance (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming). Nevertheless, at the syntactical level, *classes* are available in JS. But this is only 'syntactical sugar'. Under the hood, JS uses the prototype mechanism.

C-like syntax

The JS syntax is very similar to that of C, Java, or other members of the C-family. But we must always consider that the concepts and runtime behavior are distinctly different.

Relation to Java

JS has no relation to Java aside from having a C-like syntax. To avoid possible confusion, we would like to highlight some distinctions between JS and Java clearly.

In the beginning, *Netscape* developed JavaScript, and *Sun Microsystems* developed Java. Java includes classes and object instances, whereas JavaScript uses prototypes. In Java, variables must be declared before usage, which is unnecessary (but not recommended) in JS.

In Java, variables have an immutable static type (`int` or `String`, for example) that remains the same during the complete lifespan of a running program. In JS they also have a type (`Number` or `String`, for example), but this type can change during the lifespan of a running program. The type is detected from the environment. Therefore it's not necessary and not possible to define the type explicitly.

```
int x = 0;           // Java: 'name of type', 'name of variable', ...
let x = 0;           // JS: 'let' or 'const', 'name of variable', ...
                     //     The type will be 'Number' because of the right side of the equal
sign.
let x = String (0); // JS: explicit change from 'Number' to 'String' BEFORE assignment to x
                     //     The type will be 'String'. Test it with: alert(typeof x)
```

JS engines

JS can run on the client-side as well as on the server-side. First versions of JS have run in Browsers that acted as mere interpreters. Today, the language is handled by just-in-time compilers (JIT). They parse the script, create an Abstract Syntax Tree (AST), optimize the tree, generate a JIT-specific bytecode out of the AST, generate hardware-specific machine code out of the bytecode, and bring the machine code to execution. Such just-in-time compilers exist not only in Browsers. They can also be part of other applications, e.g.: *node.js* which is written mainly in C++.

Widely used JS engines are:

- *V8* from Google: Google Chrome, Electron, Chromium, node.js
- *SpiderMonkey* from Mozilla, Firefox
- *JavaScriptCore* from Apple, Safari
- *ActionScript* from Adobe, Flash

References

1. MDN: Data Types (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Overview)
2. MDN: Details on Data Types (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

Relation to other languages

If you have programming experience or training with another language, learning JS will be easier and more difficult at the same time. Especially the fact that JS is a member of the C-family languages (C, C++, Java, ...) may help and confuse simultaneously. It uses the common syntax but implements concepts that differ from the other family members. Therefore it might entice you with this C- or Java-like syntax. Underneath, it is an entirely different beast. The semantic and design are influenced by the programming languages Self and Scheme.

The easy part for C or Java programmers will be picking up the syntax. Most parts of the flow control, logic, and arithmetic are the same. After diving deeper into the language, you will notice the differences. Here are some **distinctions to Java**.

Behavior of Variables

Starting with the obvious, JS is a loosely typed language. There are several implications for this:

- Integers and floats are both represented by 64-bit floating point numbers (but crazily enough, bitwise operations are still available and sometimes faster).
- You may change a variable's type at will.
- Objects consist of key/value pairs, labeled 'properties'. Values can be changed, and complete properties can be added or removed from an object at will.

The list goes on, and we are granted extraordinary powers to do wonderful and sometimes incredibly stupid things whilst programming. So it's very important to keep sober-minded when attempting to harness the power of the dynamic variables.

Scope of Variables

- From module to module: In Java the key-words `public`, `private`, and `protected` defines the visibility of variables to other modules (classes). In JS there is only the simple `export/import` resp. `module.exports/require` mechanism.
- Within a module: In JS the visibility within a module is defined by the key-words `var`(outdated, don't use it anymore), `let`, and `const`. Java knows the keyword `var` too, but its semantic is a little different. The JS `const` is equivalent to Java's `final`.
- Within blocks: 'Blocks' are code sequences surrounded with brackets { }. The visibility of variables concerning blocks is identical in JS (using `let` or `const`) and Java: The variables are visible within the defining block and the included blocks but not outside of the defining block.
- Closure: A *closure* is an - anonymous or named - function that can refer to variables of its enclosing context. In this case, the function 'sees' the variable's value as of the moment the function was created but not as of the later moment when it is called. JS has used this concept since its early days. Java uses similar but not identical technics to access variables of outer context: inner classes and lambda expressions.

Classes

Unlike Java, JS is a classless language. Instead, JS uses prototypes to implement object-oriented features like inheritance. Classes can be simulated, but it is only 'syntactically sugar', and if you remove the idea of classes from your head, the learning process will become easier.

First program

Here is a single JavaScript statement, which creates a pop-up dialog saying "Hello World!":

```
alert("Hello World!");
```

For the browser to execute the statement, it must be placed inside a HTML `<script>` element. This element describes which part of the HTML code contains executable code. It will be described in further detail later.

```
<script>
  alert("Hello World!");
</script>
```

The `<script>` element should then be nested inside the `<head>` element of an HTML document. Assuming the page is viewed in a browser that has JavaScript enabled, the browser will execute (carry out) the statement as the page is loading.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Some Page</title>
    <script>
      alert("Hello World!");
    </script>
  </head>
  <body>
    <p>The content of the web page.</p>
  </body>
</html>
```

This basic 'Hello World' program can then be used as a starting point for any new program that you need to create.

Exercises

1. Copy and paste the basic program into a file, and save it on your hard disk as 'exercise_1-1.html'. You can run it in different ways:

- By going to the file with a file manager and opening it using a web browser.
- By starting your browser and then opening the file from the menu.
- By starting your browser and then specify the URL to 'exercise_1-1.html' with the *file* protocol. Please note that a) there are 3 slashes after 'file:' and b) replace 'temp' with the name of your directory.
  - Windows: `file:///C:/temp/exercise_1-1.html` (it's Windows syntax, nevertheless use *slash* instead of *backslash*)
  - Linux: `file:///temp/exercise_1-1.html`

What happens?

Click to see solution

```
A dialog appears with the text: Hello World!
```

---

2. Save the above file as 'exercise_1-2.html'. Replace the double quotes in the line `alert("Hello World!");` with single quotes, so it reads `alert('Hello World!');` and save the result. If you open this file in the browser, what happens?

Click to see solution

```
Nothing changes. A dialog appears with the text: Hello World! Double quotes and single quotes
(apostrophes) are equivalents in JS.
```

JavaScript within HTML

The language JavasScript was originally introduced to run in browsers and handle the dynamic aspects of user interfaces, e.g., validation of user input, modifications of page content (DOM) or appearance of the user interface (CSS), or any event handling. This implies that an interconnection point from HTML to JS must exist. The HTML element `<script>` plays this role. It is a regular HTML element, and its content is JS.

The `<script>` element may appear almost anywhere within the HTML file, within `<head>` as well as in `<body>`. There are only a few criteria for choosing an optimal place; see below.

Internal vs. external JavaScript

The `<script>` element either contains JS code directly, or it points to an external file resp. URL containing the JS code through its `src` attribute. The first variant is called *Internal JavaScript* or *Inline JavaScript*, the second *External JavaScript*.

In the case of *Internal JavaScript* the `<script>` element looks like:

```
<script>
  // write your JS code directly here. (This line is a comment in JS syntax)
  alert("Hello World!");
</script>
```

Internal scripting has the advantage that both your HTML and your JS are in one file, which is convenient for quick development. This is commonly used for temporarily testing out some ideas, and in situations where the script code is small or specific to that one page.

For the *External JavaScript* the `<script>` element looks like:

```
<!--  point to a file or to an URL where the code is located. (This line is a comment in HTML
syntax) -->
<script src="myScript.js"></script>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.1/dist/js/bootstrap.bundle.min.js">
</script>

<!-- although there is nothing within the script element, you should consider that the HTML5
spec  -->
<!-- doesn't allow the abbreviation of the script element to: <script src="myScript.js" />
-->
```

Having your JS in a separate file is recommended for larger programs, especially for such which are used on multiple pages. Furthermore, such splits support the pattern of Separation of Concerns: One specialist works on HTML, and another on JS. Also, it supports the division of the page's content (HTML) from its behavior (JS).

Overall, using *External scripting* is considered a best practice for software development.

External JavaSscript

For more detailed information you can refer to MDN [1].

The `src` attribute

Adding `src="myScript.js"` to the opening `script` tag means that the JS code will be located in a file called *myScript.js* in the same directory as the HTML file. If the JS file is located somewhere else, you must change the `src` attribute to that path. For example, if it is located in a subdirectory called *js*, it reads `src="js/myScript.js"`.

The `type` attribute

JS is not the only scripting language for Web development, but JS is the most common one on client-side (PHP runs on server-side). Therefore it's considered the default script type for HTML5. The formal notation for the type is: `type="text/javascript"`. Older HTML versions know a lot of other script types. Nowadays, all of them are graded as *legacy*. Some examples are: `application/javascript`, `text/javascript1.5`, `text/jscript`, or `text/livescript`.

In HTML5, the spec says that - if you use JS - the `type` attribute should be omitted from the script element [2], for *Internal Scripting* as well as for *External Scripting*.

```
<!-- Nowadays the type attribute is unnecessary -->
<script type="text/javascript">...</script>

<!-- HTML5 code -->
<script>...</script>
```

The `async` and `defer` attributes

Old browsers use only one or two threads to read and parse HTML, JS, CSS, ... This may lead to a bad user experience (UX) because of the latency time when loading HTML, JS, CSS, images, ... sequentially one after the next. When the page loads for the first time, the user may have the impression of a slow system.

Current browsers can execute many tasks in parallel. To initiate this parallel execution with regards to JS loading and execution, the `<script>` element can be extended with the two attributes `async` and `defer`.

The attribute `async` leads to asynchronous script loading (in parallel with other tasks), and execution as soon as it is available.

```
<script async src="myScript.js"></script>
```

`defer` acts similar. It differs from `async` in that the execution is deferred until the page is fully parsed.

```
<script defer src="myScript.js"></script>
```

Location of <script> elements

The `script` element may appear almost anywhere within the HTML file. But there are, however, some best practices for speeding up a website [3]. Some people suggest to locate it just before the closing `</body>` tag. This speeds up downloading, and also allows for direct manipulation of the Document Object Model (DOM) while it is rendered. But a similar behavior is initiated by the above described `async` and `defer` attributes.

```
<!DOCTYPE html>
<html>
<head>
  <title>Example page</title>
</head>
<body>
  <!-- HTML code goes here -->
  <script src="myScript.js"></script>
</body>
</html>
```

The <noscript> element

It may happen that people have deactivated JS in their browsers for security or other reasons. Or, they use very old browsers which are not able to run JS at all. To inform users in such cases about the situation, there is the `<noscript>` element. It contains text that will be shown in the browser. The text shall explain that no JS code will be executed.

```
<!DOCTYPE html>
<html>
<head>
  <title>Example page</title>
  <script>
    alert("Hello World!");
  </script>
  <noscript>
    alert("Sorry, the JavaScript part of this page will not be executed because JavaScript is
not running in your browser. Is JavaScript intentionally deactivated?");
  </noscript>
</head>
<body>
  <!-- HTML code goes here -->
</body>
</html>
```

JavaScript in XHTML files

XHTML uses a stricter syntax than HTML. This leads to small differences.

First, for *Internal JavaScript* it's necessary that the scripts are introduced and finished with the two additional lines shown in the following example.

```
<script>
  // <![CDATA[
  alert("Hello World!");
```

```
  // ]]>
</script>
```

Second, for *External JavaScript* the `type` attribute is required.

Reference

1. MDN: The script element (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script)
2. WHATWG: The type attribute (https://html.spec.whatwg.org/dev/scripting.html#attr-script-type)
3. Yahoo: Best practices for speeding up your website (http://developer.yahoo.com/performance/rules.html)

Handling Events

So far, the shown JS programs start automatically when the HTML page is loaded. This kept everything small, simple, and straightforward.

Now we want to introduce the mechanism of starting a program intentionally at specific points in time. When a user types in some content, it may be necessary to evaluate it; when a user clicks a button, it may be necessary to start a complex action like a database query. The actions (validation, DB query) will be handled in a JS function, and some HTML code will initiate the call to such JS functions. To understand how it works, we must learn about **events**. In a later chapter, we will explain events in detail.

Assume a user clicks on a button. This generates an *onclick* event. The occurrence of such an event is declaratively combined with a call to a named JS function. The browser calls this JS function. The JS function runs.

The complete HTML with its embedded JS reads:

```
<!DOCTYPE html>
<html>
<head>
  <title>Testing an event</title>
  <script>
  function showSomething() {
    alert("Someone clicked the button.");
  }
  </script>
</head>
<body>
  <h1>Click the button</h1>
  <button type="button" onclick="showSomething()">A button</button>
</body>
</html>
```

- In contrast to the previous examples, nothing happens when the page is loaded. This results from the fact that the `alert()` command is embedded in the JS function `showSomething()`. This function is only defined but not started.
- The HTML element `<button>` has an attribute `onclick` with a value `showSomething()`.
- When the button is clicked, an onclick event occurs. The event is bound to the button.
- The browser handles the event. He notices that a function with the given name shall run.
- The browser searches for the function in all available JS scripts (please notice that this name as well as all JS code is case-sensitive!).
- After finding the function with exactly this name in the `<script>...</script>` part, the browser starts the function.
- The function executes and shows the message.

In essence, the JS function with its message-showing behavior will run whenever the user clicks on the button.

Second Example

This example offers the possibility to change the background color of some HTML elements. It uses 3 buttons, 2 JS functions, and 1 event type (*onclick*; there are many other types). Furthermore, it introduces the concept of an *element id*.

```
<!DOCTYPE html>
<html>
<head>
  <title>Second test for events</title>
  <script>
  function makeGreen() {
    document.getElementById("mainPart")
            .style.background = "green";
  }
  function makeBlue() {
    document.getElementById("mainPart")
            .style.background = "blue";
    document.getElementById("blueButton")
            .style.background = "aqua";
  }
  </script>
</head>
<body id="mainPart">
  <h1>Click the buttons</h1>
  <button type="button" onclick="makeGreen()">GREEN</button>
  <button type="button" onclick="makeBlue()"
          id="blueButton">BLUE</button>
  <button type="button">Nothing</button>
</body>
</html>
```

What are the differences to the first example?

- There are 3 buttons. The first two are associated with a JS function; the third is not.
- The `<body>` and the second `<button>` elements contain an attribute `id="..."`. This assigns an identifier to them. He can be used in JS to locate them.
- The `<script>` element contains two functions with different names.
- When you click on one of the buttons, the associated JS function is called. Because the third button has no associated JS function, nothing happens there.
- In JS, the code part `document.getElementById("...")` locates the element with this ID. The code part `.style.background = "..."` modifies the background color of this element. Please ignore that the two parts are written in separate lines. This is only for better reading of the source code on small devices. You can link the two lines together.
- The `makeBlue()` function executes two statements and changes the background color of the body and of itself.

Development Tools

When working with JS (or HTML, CSS, ...) you need some tools for handling the source code and becoming more productive. First, there are tools helping you to develop the source code. Second, the code can be validated against syntax rules. Next, the code can be 'beautified' (indentations, line breaks, empty lines, ...). And lastly, the code can be optimized - in regard to size, performance, or encryption.

Development Stage

Locally installed Tools

The simplest way to write the code is using a pure text editor like *Notepad++*, *Vim*, or Gnome *gedit* or Gnome *Text editor*. You shall create a subdirectory where all the files of your project are located, e.g. `.../firstProject`. There you can create the files, e.g. `excercise_1.1.html`. Start the files as described in the chapter First program. The result shall be seen in your browser.

This approach has its pros and cons. The tools are easy to install, and they have simple user interfaces. But they don't give you much help; they just collect the text you write. Sometimes they support syntax highlighting or automatic indentation, mostly no hints concerning syntax errors. The user is forced to know the syntax of the language with which he is currently working. We believe that this approach is well suited for beginners: they shall perform their exercises at a low level.

At the next level, you can use a more complex tool, e.g. *Visual Studio Code*. In addition to text handling, they offer additional features from syntax highlighting up to debugging or extensive support for developer teams.

Online

As an alternative to locally installed tools, you can work with an online JS tool, e.g. JsFiddle (https://jsfiddle.net/), W3Schools (https://www.w3schools.com/js/tryit.asp?filename=tryjs_default), or PlayCode (https://playcode.io/javascript). They usually offer an integrated environment for HTML, CSS, and JS, including a browser preview and access to the browser's console.

Validation

It's likely that in a first step your code contains some syntactical errors, e.g. `f0r` instead of `for`. Browsers are very lenient and try to repair or ignore simple errors silently. Of course, all your source code shall conform to the given syntactical rules. But because the browsers offer limited access to validate it or to locate the erroneous line, it's necessary to use a tool for locating and explaining syntactical errors.

Such validators sometimes offer not only validation but also some more features like formatting or a preview, e.g. JS validator (https://javascriptvalidator.net/), JS + HTML validator (https://www.htmlstrip.com/javascript-validator), W3C validator for HTML (https://validator.w3.org/#validate_by_input), FreeFormatter (https://www.freeformatter.com/html-validator.html).

Optimization, Obfuscating

The time necessary to transfer HTML and JS code from a server to a browser can be shortened by stripping out all unnecessary blanks, newlines, tabs, etc. .

When JS code is loaded into a browser, it can easily be viewed by the user. If you want to hide your source code, you can use the obfuscating technique. It generates a more or less unreadable code (for humans) out of the original code.

The previous JS + HTML validator (https://www.htmlstrip.com/javascript-validator) offers all of these techniques.

Self Test

You can test yourself by answering the following questions. When in doubt, you should undertake a test in your development environment (editor, IDE, browser). So it's also a test of whether you have installed all the necessary tools to execute the examples of the Wikibook.

Introduction

1 Inside which HTML element do you put the JavaScript code?

- ○ <javascript>
- ○ <source>
- ○ <script>
- ○ <scripting>
- ○ None of the above.

2 Does this sequence of statements lead to an error?

```
let x = 0;
x = 1;
x = 'one';
```

- ○ Yes
- ○ No

Submit

Syntax and Semantic

1 Is there a syntax error?

```
let x = 1:
```

- ○ Yes
- ○ No

2 Is there a syntax error?

```
let x = 1;
let y = 2;
x+y = 3;
```

- ○ Yes
- ○ No

3 Which line contains a syntax error?

```
/* 1 */ let x = 1;
/* 2 */ let y = 1;
/* 3 */ let x = 1, y = 1;
/* 4 */ let x, y = 1;
```

- ○ Line 1
- ○ Line 2
- ○ Line 3
- ○ Line 4
- ○ None of the above

4 Which lines contain a syntax error?

```
/* 1 */ let firstName_1 = "Michael";
/* 2 */ let firstName_2 = 'Michael';
/* 3 */ let firstName_4 = 'Mikhaïl';
/* 4 */ let firstName_5 = "Михаил";
```

- ☐ Line 1
- ☐ Line 2
- ☐ Line 3
- ☐ Line 4
- ☐ None of the above

5 Which line leads to an error message?

```
/* 1 */ let x = 1;
/* 2 */ const y = 2;
/* 3 */ x = 3;
/* 4 */ y = 4;
```

- ○ Line 1
- ○ Line 2
- ○ Line 3
- ○ Line 4
- ○ None of the above

6 Which line leads to an error message?

```
/* 1 */ let x = 1;
/* 2 */ x = 2 + 3(4 + 5);
/* 3 */ x = 3;
/* 4 */ x = -x;
```

- ○ Line 1
- ○ Line 2
- ○ Line 3
- ○ Line 4
- ○ None of the above

7 Which line(s) leads to an error message?

```
/* 1 */ let x = [1, 2];
/* 2 */ let x = [[1], [2]];
/* 3 */ let x = [[1, 2], [3, 4]];
/* 4 */ let x = [[[1], [2]], [[3], [4]]];
/* 5 */ let x = [1], [2];
/* 6 */ let x = [1, '2'];
```

- ☐ Line 1
- ☐ Line 2
- ☐ Line 3
- ☐ Line 4
- ☐ Line 5
- ☐ Line 6
- ☐ None of the above

8 Is there a syntax error?

```
"use strict";
let persons = ['Alice', 'Bert', 'Caesar'];
for (let i = 0, i < persons.length, i++) {
```

```
  alert(persons[i]);
} //
```

- ○ Yes
- ○ No

9 Which line contains a syntax error?

```
/* 1 */ alert("1 + 2 = " + 3);
/* 2 */ alert("1 + 2 = " + "3");
/* 3 */ alert("1 + 2 = " 3)
/* 4 */ alert(1 + 2 == 3)
```

- ○ Line 1
- ○ Line 2
- ○ Line 3
- ○ Line 4
- ○ None of the above

Submit

Type Conversion

1 Which message(s) will **not** be shown?

```
let x = 1, y = 2, z = '2';
alert(x + y);
alert(x + z);
alert(x - z);
```

- ☐ 3
- ☐ 12
- ☐ -1
- ☐ All will be shown.

2 What message(s) will **not** be shown?

```
let x = 1;
alert(x);
alert(x = 5);
alert(x == '5');
alert(x === 5);
```

- [ ] Error Message
- [ ] true
- [ ] false
- [ ] 5
- [ ] 1

Submit

Loops

1 What will be the result in `sum`?

```
"use strict";
let sum = 0;
for (let i = 1; i < 5; i++) {
  sum = sum + i;
} //
alert(sum);
```

The result in sum will be [ ] .

2 What will be the result in `sum`?

```
"use strict";
let sum = 0;
for (let i = 0; i < 5; i++) {
  for (let j = 10; j >= 0; j--) {
    if (i === j) {
      sum = sum + i - j + 1;
    } //
  } //
} //
alert(sum);
```

The result in sum will be [ ] .

3 Is there a syntactical error?

```
"use strict";
let sum = 0;
for (let i = 0; i < 5; i++)
  for (let j = 10; j >= 0; j--) {
    if (i === j) {
      sum = sum + i - j + 1;
    } //
  } //
} //
alert(sum);
```

- ◯ Yes
- ◯ No

4 What will be the result?

```
"use strict";
let sum = 0;
for (let i = 0; i = 3; i++) {
  sum = sum + i;
} //
alert(sum);
```

- ◯ 0
- ◯ 6
- ◯ Infinite loop
- ◯ None of the above

Submit

Helpful hints

Predefined functions

All over the Wikibook, code examples are given. Usually, they show their results somewhere in the browser. This is done via methods that are aligned with Web API interfaces (https://developer.mozilla.org/en-US/docs/Web/API) and implemented in browsers.

`alert()`

The function `alert()` creates a modal window showing the given text.

```
let x = 5;
window.alert("The value of 'x' is: " + x); // explicit notation
alert("The value of 'x' is: " + x);        // short notation
```

`log()`

The function `log()` writes the message to the console of the browser. The console is a window of the browser that is normally not opened. In most browsers, you can open the console via the function key `F12`.

```
let x = 5;
console.log("The value of 'x' is: " + x); // explicit notation
                                          // no short notation
```

`print()`

The function `print()` opens the print dialog to print the current document.

```
window.print(); // explicit notation
print();        // short notation
```

`prompt()`

The function `prompt()` opens a modal window and reads the user input.

```
let person = window.prompt("What's your name?"); // explicit notation
person = prompt("What's your name?");            // short notation
alert(person);
```

`write()`

The use of `document.write()` is strongly discouraged (https://developer.mozilla.org/en-US/docs/Web/API/Document/write).

Coding style

Our code examples mostly contain the line `"use strict";` before any other statement. This advises the compiler to perform additional lexical checks. He especially detects the use of variable names which are not been declared before they receive a value.

```
"use strict";
let happyHour = true;
hapyHour = false; //  ReferenceError: assignment to undeclared variable hapyHour
```

Without the `"use strict";` statement the compiler would generate a second variable with the misspelled name instead of generating an error message.

Faulty visualization in some code examples

Due to a minor fault in the Wikibook template *quiz*, you will see `} //` instead of `}` in some examples. The two slashes at the end are only shown to overcome this problem; they are not necessary.

```
// the two slashes in the last line are not necessary
if (x == 5) {
  ...
} //
```

Lexical structure

Summary

- All elements of the language are case-sensitive, e.g.: `const x = 0; const X = 0;` defines two different variables; `CONST x = 0;` leads to a syntax error.
- Single line comments are introduced by `//`. Multi-line comments are surrounded with `/* */`.
- Semicolons for ending a statement are optional - with few exceptions.
- The first character of a variable name cannot be a number: `const 1x = 0;` leads to a syntax error.

Case Sensitivity

JavaScript is case-sensitive. This means that all keywords, variable names, and function names must be written consistently. If you create a function `Hello()` it is different from the function `HELLO()`, `hello()`, or `hEllo()`. Or, the statement `IF (x > 0) { }` leads to a syntax error because the keyword `if` is defined in lower-case.

Whitespaces

Whitespace includes spaces, tabs, and line breaks.[note 1] If there is a sequence of multiple whitespaces, JavaScript reduces them to a single whitespace, e.g: ' ' -> ' ', '\n\t' -> '\n'. This single remaining whitespace delimits the language elements like keywords and variable names, ... from each other. The resulting source code is hard to read for people [2] but (a little) easier to parse by the browser. The main advantage is the smaller size of the code which must be transferred between server and client.

The following script is an example with very little whitespace.

```
function filterEmailKeys(event){
event=event||window.event;
const charCode=event.charCode||event.keyCode;
const char=String.fromCharCode(charCode);
if(/[a-zA-Z0-9_\-\.@]/.exec(char)){return true;}
return false;
}
```

The following is the same script with a typical amount of whitespace.

```
function filterEmailKeys(event) {
  event = event || window.event;
  const charCode = event.charCode || event.keyCode;
  const char = String.fromCharCode(charCode);
  if (/[a-zA-Z0-9_\-\.@]/.exec(char)) {
    return true;
  }
  return false;
}
```

The following is the same script with a lot of whitespaces.

```
function filterEmailKeys( evt ) {

    evt = evt || window.event;

    const charCode = evt.charCode || evt.keyCode;
    const char = String.fromCharCode ( charCode );

    if ( /[a-zA-Z0-9_\-\.@]/.exec ( char ) ) {
        return true;
    }

    return false;
  }
```

Comments

Comments are parts of the source code that will - per definition - not be executed.

They allow you to leave notes in your code to help other people understand it. They also allow you to comment out code that you want to hide from the parser but you don't want to delete.

Single-line comments

A double slash `//` turns all of the following text on the same line into a comment that will not be processed by the JavaScript interpreter.

```
// Show a welcome message
alert("Hello, World!")
```

Multi-line comments

Multi-line comments start with `/*` and end with the reverse `*/`. Multi-line comments don't nest.

Here is an example of how to use the different types of commenting techniques.

```
/* This is a multi-line comment
that contains multiple lines
of commented text. */
let a = 1;
/* commented out to perform further testing
a = a + 2;
a = a / (a - 3); // is something wrong here?
*/
alert('a: ' + a);

/* This comment has two /* but they're both canceled out by */
```

Semicolons

In many programming languages, semicolons are required at the end of each code statement. In JavaScript, the use of semicolons is optional, as a new line indicates the end of the statement (with some exceptions). This is called automatic semicolon insertion.

```
// the line
x = a + b;
// is equivalent to:
x = a + b
```

But the exceptions can be quite surprising.[3] Automatic semicolon insertion can create hard to debug problems.

```
a = b + c
(d + e).print()
```

The above code is not interpreted as two statements. Because of the parentheses on the second line, JavaScript interprets the above as if it were

```
a = b + c(d + e).print();
```

when instead, you may have meant it to be interpreted as

```
a = b + c;
(d + e).print();
```

Even though semicolons are optional, it's preferable to end statements with a semicolon to prevent any misunderstandings from taking place.

Literals

A literal is a hard-coded value. Literals provide a means of expressing specific values in your script. For example, to the right of the equals sign:

```
const myLiteral = "a fixed value";
```

There are several types of literals available. The most common are the string literals, but there are also numeric literals, booleans, undefined, null, regex literals, array literals, and object literals.

Examples of an object, a boolean, and a string literal:

```
const myObject = { name:"value", anotherName:"anotherValue" };
const isClosed = true;
const mayBeWrong = "true";
```

Details of these different types are covered in Variables and Types.

Identifiers

An identifier is a name for a piece of data, such as a variable, array, or function. There are rules:

- Letters, dollar signs, underscores, and numbers are allowed in identifiers.
- The first character cannot be a number.

Examples of valid identifiers:

- `u`
- `$hello`
- `_Hello`
- `hello90`

`1A2B3C` is an invalid identifier, as it starts with a number.

An example of 'identifiers' are variable names. They obey such rules.

- Uppercase and lowercase letters, underscores, and dollar signs can be used.
- Numbers are allowed after the first character.
- Non-latin characters like "á" can be used in variable names as long as they have the Unicode properties "ID_Start" or "ID_Continue" for the start or rest of the name respectively.[4] Special characters are not allowed.
- Variable names are case sensitive: different case means a different name.
- A variable may not be a reserved word.

Exercises

> ... are available on another page (click here).

Notes

1. Technically vertical tab, zero-width non-breaking space, and any Unicode character with the "Space Separator" category also count as whitespace.[1]

References

1. ECMAScript Language Specification, Chapter 12.2 - White Space (https://tc39.es/ecma262/multipage/ecmascript-language-lexical-grammar.html#sec-white-space)
2. Khan Academy (https://www.khanacademy.org/computing/computer-programming/programming/writing-clean-code/pt/readable-code)
3. ECMA-262 (https://tc39.es/ecma262/multipage/ecmascript-language-lexical-grammar.html#sec-automatic-semicolon-insertion) ECMAScript Language Specification, Chapter 12.9 - Automatic Semicolon Insertion
4. ECMAScript Language Specification, the *IdentifierName* production (https://tc39.es/ecma262/multipage/ecmascript-language-lexical-grammar.html#prod-IdentifierName)

Automatic semicolon insertion

Automatic Semicolon Insertion (ASI)

In languages of the C-family, the semicolon denotes the end of a statement. Unlike other C-like languages, JavaScript doesn't enforce that. Instead, the semicolon is optional, and the interpreter adds missing semicolons - mostly at the end of a line - to terminate statements. Doing so, he takes complex rules (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#automatic_semicolon) into account. This may conflict with the intended purpose.

If you write your code without semicolons at the end of statements, you must take care of problematic situations. Here are some rules-of-thumb to avoid problems. But there are much more rules.

1. The expression after one of the keywords `return`, `throw`, or `yield` must be on the same line as the keyword itself.
2. The label identifier after `break` or `continue` must be on the same line as the keyword.
3. If a line starts with one of `(`, `[`, `` ` ``, `+`, `-`, or `/`, end the previous line with a semicolon.

While ASI would make your code easier to write (no need to type all of those semicolons), in practice, the lack of semicolons makes your program harder to debug. Because of this, it is universally recognized as a best practice to use semicolons at the end of statements anyway. However, the existence of ASI can still create some bugs that are hard to troubleshoot if you don't know what to look for.

Examples

Entered code ..	.. interpreted as	intended code
`return` `2a + 1`	`return;` `2a + 1;`	`return 2*a + 1;`
`function getObject() {` `  return` `  {` `  // some lines` `  }` `}`	`function getObject() {` `  return;` `  {` `  // some lines` `  };` `}`	`function getObject() {` `  return {` `  // some lines` `  };` `}`
`i` `++`	`i;` `++;`	`i++;`

In the first case, the programmer intended 2*a + 1 to be returned; instead, the code returned `undefined`. Similarly, in the second case, the programmer intended to return the lines enclosed by the braces `{}`, but the code returned `undefined`. Due to this oddity in JavaScript, it is considered a best practice never to have lines break within a statement and never have the opening brace on a separate line.

See also

- MDN reference (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#automatic_semicolon_insertion)
- ECMA reference (https://262.ecma-international.org/13.0/#sec-automatic-semicolon-insertion)
- Whitespaces & semicolons in JavaScript

Reserved words

In JavaScript, some tokens (words) have a special semantic (meaning). Therefore they cannot be used as names of variables, functions, classes, etc [1] [2]. Some of them are **generally reserved words**; others are reserved only in a special context; others are reserved for possible future usage without having a special functionality nowadays; others have been defined in outdated ECMAScript versions of the years 1997 - 99.

The list of such special words as of 2022 follows. For some of the words, we offer further information.

- abstract
- await
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- debugger
- default
- delete
- do
- double
- else
- enum
- 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

Furthermore, there are predefined methods like `forEach()`, predefined modules like `Math`, or predefined objects like `BigInt` whose names should be avoided also.

References

1. ECMA: Keywords (https://262.ecma-international.org/#sec-keywords-and-reserved-words)
2. MDN: Keywords (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#keywords)

Variables

Purpose

Computer languages need to use variables. Why this? In most cases, programs don't solve single problems like the very concrete question: What is the circumference of a circle with a radius of 5 cm? Such a concrete question can be solved without using variables: `alert (2 * 5 * 3.14);` Instead, most questions are more general: What is the circumference of a circle with an arbitrary radius? You don't want to write a program for a radius of 5 cm, another program for a radius of 6 cm, another one for a radius of 7 cm, and so on. You want to write a single program that computes the circumference for all possible radii. The program needs input (from a user, from another program, from a database, ...) that tells him for which value it shall run. `let r = prompt("How big is the radius of your circle?"); alert (2 * r * 3.14);`, or even better: `let r = prompt("How big is the radius of your circle?"); alert (2 * r * Math.PI);`.

Those two examples are flexible. They ask the user for the desired radius, store the given value **in a variable** with the name *r*, and compute the circumference using this variable. The variable *r* is introduced by the keyword `let`. And there is a second variable. The module *Math* has predefined a variable *PI* with the keyword `const`: `const PI = 3.141592653589793;`.

In JavaScript, variables can be used similarly to variables in mathematical formulas. During runtime, the values of variables are stored in the main memory of the computer (RAM), from where they can be used at a later moment in the lifetime of the program. You can imagine a variable as a small box where you deposit some value and take it out whenever you need it.

Variables are a cornerstone in the transition from individual problem solving to a strategy, respectively an **algorithm**.

Declaration and initialization

If you want to use a variable, we recommend declaring (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types#declarations) them explicitly. This is not mandatory but has strong benefits.

In many cases, the declaration is accompanied by an initialization, e.g. `let x = 0;`. The declaration is `let x;`, and the initialization is `x = 0;`. But it's also possible to omit the initialization part: `let x;` which causes the value of the variable to be `undefined`.

Keyword `let`

The keyword `let` introduces a variable whose value may be changed multiple times.

```
let x = 0;
// ...
x = x + 5;
// ...
```

```
x = -4;
// ...
```

Keyword const

The keyword `const` introduces a variable that **must** be initialized immediately. Moreover, this very first value can never be changed. That helps the JavaScript engine to optimize the code for better performance. Use `const` as often as possible.

```
const maxCapacity = 100;
// ...
maxCapacity = maxCapacity + 10; // not possible
let maxCapacity = 110;          // not possible
```

When you work with objects, e.g., with arrays, in combination with the keyword `const`, it's the same: you cannot assign another value (object, array, number, or whatever) to the variable. Nevertheless, its elements can be changed.

```
const arr = [1, 2, 3];
arr = [1, 2, 3, 4];     // not possible
arr = new Array(10);    // not possible

arr[0] = 5;             // ok:  5, 2, 3
alert(arr);

arr.push(42);           // ok:  5, 2, 3, 42
alert(arr);
```

In some cases `const` variables are written in uppercase, e.g., `PI`. This is a convention and not mandatory.

Keyword var

At first glance, `var` is identical to `let`. But the range where such variables are known is different from the variables declared by `var`; see chapter Scope below.

Omitting the declaration

You can assign a value to a variable without declaring the variable previously. "JavaScript used to allow assigning to undeclared variables, which creates an *undeclared global variable*. This is an error in *strict mode* and should be avoided altogether."[1] In other words: The variable goes to global scope, see below. As long as you don't have good reasons you should avoid using the global scope because its usage tends to create unwanted side effects.

```
// direct usage of 'radius' without any keyword for its declaration
/* 1 */ radius = 5;
/* 2 */ alert (2 * radius * 3.14);
```

Sometimes such situations occur by accident. If there is a typing error in the source code, JavaScript uses two different variables: the original one and a new one with the wrongly typed name - even if you use one of the keywords `let`, `const`, or `var`.

```
let radius = 5;  // or without 'let'
alert("Test 1");
// ... later in the code
radus = 1; // typo will not be detected
alert("Test 2");
```

You can instruct JavaScript to search for such typos by inserting the command `"use strict";` as the first line of your scripts.

```
"use strict";
let radius = 5;
alert("Test 1");
// ... later in the code
radus = 1;       // typo will be detected and an error message given
alert("Test 2"); // will never execute
```

Data types

Programmers who are familiar with (strict) typed languages like Java may miss in the above chapter the possibility of defining the type of variables. JavaScript knows many different data types. But their handling and behavior is very different from that in Java. In the next chapter you will learn about that.

Scope

A *scope* is a range of consecutive JavaScript statements with a clearly defined start and end. JavaScript knows four types of scopes: block, function, module, and global scope. Depending on the kind of declaration and the location of the declaration, variables are within such scopes. They are 'visible' respectively 'accessible' only within their scope. If you try to access them from outside, an error will occur.

Block scope

A pair of curly brackets `{}` creates a *block*. Variables declared within a block by `let` or `const` are bound to this block and cannot be accessed outside.

```
"use strict";
let a = 0;
// ...
if (a == 0) {
  let x = 5;
  alert(x);   // shows the number 5
} else {
  alert(x);   // ReferenceError (with a different 'a')
}
alert(x);     // ReferenceError
```

The variable `x` is declared inside a block (in this simple example, the block consists of only two lines.) It is not accessible behind the end of the block, which is the closing curly bracket `}` in the `else` line. The same applies to the case that the variable `x` is declared with `const` instead of `let`.

Be careful with the keyword `var`; its semantics is different! First, `var` is not block-scoped. Second, it leads to a technique called hoisting (https://developer.mozilla.org/en-US/docs/Glossary/Hoisting), which has been used in JavaScript since its first days. Hoisting changes the semantics of different declarations

'under the hood'. Concerning `var`, it splits declaration and initialization into two separate statements and shifts the declaration part to the top of the current scope. Hence the variable is declared but not initialized if you use it before the line where it is declared **in the source**.

The script

```
"use strict";
alert(x);   // undefined, not ReferenceError !
x = 1;      // correct, despite of "use strict"
alert(x);   // shows 1
var x = 0;
```

is changed to:

```
"use strict";
var x;
alert(x);   // undefined, not ReferenceError !
x = 1;
alert(x);   // shows 1
x = 0;
```

On the other hand, the keyword `let` keeps the declaration in the line where it is written.

```
"use strict";
alert(x);     // ReferenceError
x = 1;        // ReferenceError
alert(x);     // ReferenceError
let x = 0;
```

There are more differences. Here is a version of the first example of this chapter replacing `let` by `var`:

```
"use strict";
let a = 0;
// ...
if (a == 0) {
  var x = 5;  // 'var' instead of 'let'
  alert(x);   // shows the number 5
} else {
  alert(x);   // ReferenceError (with a different 'a')
}
alert(x);     // shows the number 5  !!
```

We recommend avoiding `var` completely because of two reasons:

- JavaScript's hoisting technique is not easy to understand.
- Other members of the C-family languages don't know it.

Instead of using `var`, use the keyword `let`.

Function scope

A function creates its own scope. Variables declared in the function's scope cannot be accessed from outside.

```
"use strict";
function func_1() {
  let x = 5; // x can only be used in func_1
  alert("Inside function: " + x);
}
func_1();
alert(x); // Causes an error
```

The function scope is sometimes called the *local* scope because this was the name in older ECMAScript versions.

See also: Closures works the other way round - access of outer variables inside the function.

Module scope

It is possible to divide huge scripts into multiple files and let the functions and variables communicate with each other. Each file creates its own scope, the module scope. The chapter JavaScript/Modules explains more about that.

Global scope

Variables or functions are in global scope if they are declared at the top level of a script (outside of all blocks and functions).

```
"use strict";
let x = 42;   // 'x' belongs to global scope

// define a function
function func_1() {
  // use variable of the global context
  alert("In function: " + x);
}

// start the function
func_1();   // shows "In function: 42"
alert(x);   // shows "42"
```

`x` is declared at the top level, hence it is in the global scope and can be used everywhere. But in the next example the declaration of `x` is wrapped by `{ }`. Hence it is no longer in global scope.

```
"use strict";
{
  let x = 42;   // 'x' is not in global scope
}
alert(x);       // ReferenceError
```

Hint: The use of the global scope isn't considered good practice. It tends to create unwanted side effects. Instead, try to modularize your code and let the parts communicate via interfaces.

Exercises

... are available on another page (click here).

See also

- Closures

References

1. MDN: Variable declarations (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types#declarations)

Data types

Introduction

Each variable in JavaScript has a certain data type (Number, String, Boolean, ...) - as long as a value is stored in the variable. The type of the value determines the type of the variable. In contrast to strongly typed languages, it is possible to assign - over time - values of different types to the variable so that the type of the variable may change. This is called *weakly* or *loosely typing*. The advantage is that JavaScript programmers have a wealth of possibilities and freedom to use (or abuse) variables. On the other hand, in strongly typed languages, a lot of formal errors can be detected during compile-time.

JavaScript knows seven *primitive data types* (Number, String, Boolean, BigInt, Symbol, Undefined, Null) and diverse other data types, which are all derived from *Object* (Array, Date, Error, Function, RegExp) [1] [2]. *Objects* contain not only a value, they also have methods and properties. The same can happen with *primitive data types*. If they try to invoke methods, the JS engine 'wraps (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values)' them with a corresponding object wrapper and calls its methods instead. This technique is sometimes called *boxing*.

You may wonder why we describe data types and initialization in the same chapter. The reason is that they are very closely related to each other. The initialization (and subsequent assignments) of a value to a variable determines its type - as noted above. That's why there is no designation of a type during initialization, in contrast to some other languages `private int i = 0; /* Java */`

(Note: JSON is a text-based data format, not a data type. As such, it’s language-independent. It uses the JavaScript object syntax.)

Categories of data types

Data types are explained in the following chapters.

- Primitive data type
- Objects
- Arrays
- Dates
- Regular Expressions

References

1. MDN: Data Types (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Overview)
2. MDN: Details on Data Types (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

Primitive data types

Primitive types use a fixed format; some can contain only a limited number of certain values. In contrast, objects are more complex, especially including methods and properties.

With the exception of `null` and `undefined`, primitive types have a corresponding object wrapper (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values) with data type specific methods. Therefore you will find on this page descriptions of some methods.

String

String is a datatype to hold text of arbitrary length. String variables are created by assigning a string literal to them. String literals can be enclosed in `" "` or `' '`.

```
"use strict";
const myName_1 = "Mike";      // double quote
const myName_2 = 'Monica';    // apostrophe
const myName_3 = "naḽɯçito";  // non-latin characters
```

If your string literal contains a `"` or `'`, you can use the other one as the outer delimiter, or you escape them with a `\` .

```
"use strict";
const book_1 = "Mike's book";
const monica_1 = 'Here name is "Monica".';
const book_2 = 'Mike\'s book';
const monica_2 = "Here name is \"Monica\".";
```

If your string literal is computed out of some fixed text plus some dynamic parts, you can use the *template literal* technique. Here, the literal is enclosed in backticks `` ` ` `` and contains variables and expressions.

```
"use strict";
const a = 1;
const b = 2;
const resultMessage = `The sum of ${a} and ${b} is: ${a + b}.`;
// same as:
         'The sum of ' + a + ' and ' + b + ' is: ' + (a + b);
alert(resultMessage);
```

The `+` operator concatenates two strings, e.g. `alert("Hello " + "world!");`. Additionally, there are a lot of methods (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#instance_methods) for strings.

Hint: JavaScript doesn't have something like a 'character' or 'byte' data type.

Properties and methods for strings

We show some methods which are often used. For a complete list, please refer to MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#instance_methods).

length

`length` is a property, not a method. Hence there are no parenthesizes `()` . It returns the length of the string as a whole number.

```
const foo = "Hello!";
alert(foo.length);    // 6
```

concat(text)

The method returns a string where 'text' is appended to the original string.

```
const foo = "Hello";
const bar = foo.concat(" World!");
alert(bar); // Hello World!
```

indexOf(searchText)

The method returns the position of the first occurrence of 'searchText', starting with 0. If 'searchText' cannot be found, -1 is returned. The method is case sensitive.

```
const foo = "Hello, World! How do you do?";
alert(foo.indexOf(" "));    // 6

const hello = "Hello world, welcome to the universe.";
alert(hello.indexOf("welcome"));      // 13
```

lastIndexOf(searchText)

The method returns the position of the last occurrence of 'searchText'. If 'searchText' cannot be found, -1 is returned. The method is case sensitive.

```
const foo = "Hello, World! How do you do?";
alert(foo.lastIndexOf(' '));    // 24
```

replace(text, newtext)

The method returns a string where 'text' is replaced by 'NewText' on the original string. Only the first occurrence is replaced. The method is case sensitive.

```
const foo = "foo bar foo bar foo";
const newString = foo.replace("bar", "NEW");
alert(foo);         // foo bar foo bar foo
alert(newString);   // foo NEW foo bar foo
```

As you can see, the `replace` method only returns the new content and does not modify the origin string in 'foo'.

slice(start [, end])

The method returns a substring beginning at the 'start' position.

```
"hello".slice(1);   // "ello"
```

When the 'end' is provided, they are extracted up to, but not including the end position.

```
"hello".slice(1, 3);    // "el"
```

`slice` allows extracting text referenced from the end of the string by using negative indexing.

```
"hello".slice(-4, -2);  // "el"
```

Unlike `substring`, the slice method never swaps the 'start' and 'end' positions. If the 'start' is after the 'end', `slice` will attempt to extract the content as presented, but will most likely provide unexpected results.

```
"hello".slice(3, 1);    // ""
```

substr(start [, number of characters])

The method is deprecated (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substr). Use `substring` or `slice` instead.

substring(start [, end])

The method extracts a substring starting at the 'start' position.

```
"hello".substring(1);   // "ello"
```

When the 'end' is provided, they are extracted up to, but not including the end position.

```
"hello".substring(1, 3);    // "el"
```

`substring` always works from left to right. If the 'start' position is larger than the 'end' position, `substring` will swap the values; although sometimes useful, this is not always what you want; different behavior is provided by slice.

```
"hello".substring(3, 1);    // "el"
```

toLowerCase()

The method returns the current string in lower case.

```
const foo = "Hello!";
alert(foo.toLowerCase());   // hello!
```

toUpperCase()

The method returns the current string in upper case.

```
const foo = "Hello!";
alert(foo.toUpperCase());   // HELLO!
```

Number

Number is one of the two numeric types (the other one is *BigInt*). *Number* stores integer values as well as floating point values in a unified 64-bit format defined by IEEE 754. That means, that JavaScript doesn't have different data types for integers and float like some other languages.

The possible range for such values is approximate -10^{300} to $+10^{300}$ with different precision depending on the distance to 0.

In the range from $-(2^{53} - 1)$ to $+2^{53} - 1$ there is no uncertainness for integer operations. 2^{53} = 9,007,199,254,740,992 which is a little smaller than 10^{16}.

```
"use strict";
let counter = 20;   // no decimal point
alert(counter + " " + typeof counter);
let degree = 12.1;  // with decimal point
alert(degree + " " + typeof degree);
```

For *Number* the usual arithmetic operators ('power' is `**` and 'modulo' is %), comparison operators (<, >, ...), and bitwise operators are available.

In opposite to some other languages, the division of two whole numbers can result in a number with decimal places, e.g. `alert(1/3);`.

Properties and methods for numbers

Working with numbers is supported by many properties and methods. Internally, they are implemented at different areas:

- The built-in object `Math` provides properties that represent common mathematical constants like π or e. Syntax: `Math.xyz` (no parenthesis)

- The build-in object `Math` provides common mathematical functions like *sin* or *log*. Syntax: `Math.xyz()` (with parenthesis)
- The object `Number` provides properties that characterize the implementation of the data type number, like *MAX_VALUE* or *NEGATIVE_INFINITY*. Syntax: `Number.xyz` (no parenthesis)
- The object `Number` provides *static* methods that check the relation between numbers and other data types, e.g., *isInteger* or *parseFloat*. Syntax: `Number.xyz()` (with parenthesis)
- The object `Number` provides *instance* methods that act on concrete number values or variables, e.g., *toExponential* or *toFixed*. Syntax: `value.xyz()` (with parenthesis)

```
// some examples
"use strict";

const var_1 = Math.PI;
alert(var_1);
alert(var_1.toFixed(2));

const var_2 = Math.sqrt(3);
alert(var_2);

alert(Number.MAX_VALUE);

alert(Number.isInteger(123));    // true
alert(Number.isInteger(12.3));   // false
```

We show some properties and methods which are often used. For a complete list, please refer to MDN Math (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) and MDN Numbers (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number).

Properties

Most commonly used constants:

- `Math.E` Returns the constant e.
- `Math.PI` Returns the constant pi.
- `Math.LN10` Returns the natural logarithm of 10.
- `Math.LN2` Returns the natural logarithm of 2.
- `Math.SQRT2` Returns the square root of 2.

Math.ceil(number)

Returns the **smallest integer** greater than the number passed as an argument.

```
const myInt = Math.ceil(90.8);
alert(myInt);              //  91
alert(Math.ceil(-90.8));   // -90
```

Math.floor(number)

Returns the **greatest integer** less than the number passed as an argument.

```
const myInt = Math.floor(90.8);
alert(myInt);              //  90
alert(Math.floor(-90.8));  // -91
```

Math.round(number)

Returns the **closest integer** to the number passed as an argument.

```
alert(Math.round( 90.8));  //  91
alert(Math.round(-90.8));  // -91

alert(Math.round( 90.3));  //  90
alert(Math.round(-90.3));  // -90
```

Math.max(number_1, number_2)

Returns the higher number from the two numbers passed as arguments.

```
alert(Math.max(8.3,  9));  //  9
alert(Math.max(8.3, -9));  //  8.3
```

Math.min(number_1, number_2)

Returns the lower number from the two numbers passed as arguments.

```
alert(Math.min(8.3,  9));   //  8.3
alert(Math.min(8.3, -9));   // -9
```

Math.random()

Generates a pseudo-random number between 0 and 1.

```
alert(Math.random());
```

Number.parseInt(string)

Number.parseFloat(string)

The two methods `parseInt` and `parseFloat` convert strings into numbers. They scan the given string from left to right. When they recognize a character distinct from 0 - 9, they finish scanning and return the converted numbers read so far (parseFloat accepts the decimal point). If the first character is distinct from 0 - 9, they return *Math.NaN*, meaning *Not a Number*.

Hint: It's not necessary to specify 'Number.' in the source code.

```
const x = parseInt("7.5");
alert(x);  // 7

const y = parseInt("Five");
alert(y);  // NaN

const z = parseFloat("2.8") + 3;
alert(z);  // 5.8

// scientific notation is accepted
alert(parseFloat("123.456e6"));  // 123456000
```

BigInt

BigInt is a data type that represents **integers** of arbitrary length. Hence, it's a variable-length format (conceptually) delimited only by the available RAM.

*BigInt*s are created either by adding a 'n' to the end of an integer literal or by using the `BigInt()` function.

```
"use strict";
let hugeNumber_1 = 12345678901234567890n;        // 'n' at the end
alert(typeof hugeNumber_1);
let hugeNumber_2 = BigInt("12345678901234567890"); // no 'n'
alert(typeof hugeNumber_2);
// a 'small' BigInt can be created out of an integer value
let hugeNumber_3 = BigInt(123);
alert(typeof hugeNumber_3);
```

BigInt supports the arithmetic operators `+ - * / **`, comparison operators, and most of the bitwise operators.

Boolean

Boolean variables can contain one of two possible values, `true` or `false`. JavaScript represents `false` by either a Boolean false, the number 0, NaN, an empty string, or the built-in types undefined or null. Any other values are treated as `true`.

```
"use strict";
let firstLoop = true;
alert(typeof firstLoop);
```

Undefined

Variables that have just been declared but not initialized with any value have the data type *undefined*.

```
"use strict";
let x;
// ...
alert(typeof x);
if (x == undefined) {
  // remedy the missing initialization
  x = 0;
}
```

```
alert(typeof x);
// ...
```

Null

In JavaScript, *null* is marked as one of the primitive values, because its behavior is seemingly primitive. However, when using the `typeof` operator, it returns "object". This is considered a bug, but one which cannot be fixed because it will break too many scripts.[1]

Symbol

Symbol represents a **unique** identifier. *Symbol*s may have a descriptor that is given as a string to its constructor.

```
"use strict";
// 'person' is a symbol with the description "Olaf"
const person = Symbol("Olaf");
alert(person.description); // Olaf
```

Symbols (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global%20Objects/Symbol) are used as keys in objects (embedded in `[ ]`) to guarantee uniqueness.

See also

ECMAScript definitions on data types (https://262.ecma-international.org/#sec-ecmascript-data-types-and-values)

Exercises

> ... are available on another page (click here).

References

1. MDN: Null (https://developer.mozilla.org/en-US/docs/Glossary/Null)

Objects

In JavaScript, an *Object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects)* is a collection of *properties*; *properties* are key-value pairs. The keys are Strings or Symbols. Values can be of any data type - including functions (in this case, they are called methods).

Objects look like associative arrays implemented as hashtables. But in practice, JavaScript engines might have implemented them in other ways to achieve optimal performance.

Please consider that JavaScript objects are not identical to JSON. JSON is a format for a textual representation ('serialization' / 'de-serialization') of objects, which can also be done in other languages or in a pure text editor.

Create an object

The JavaScript syntax supports different ways to create objects.

'**Literal notation**' using curly braces `{ }`

```
"use strict";

// an empty object (no properties)
const obj_0 = {};
alert(JSON.stringify(obj_0));

// 'obj_1' contains a set of 3 properties. The value of key 'c' is
//         a second object with an empty set of properties.
const obj_1 = { a: "Any string", b: 42, c: {} };
alert(JSON.stringify(obj_1));

// using variables
const a = "Any string";
const b = 42;
const c = {};
const obj_2 = { a: a, b: b, c: c };
alert(JSON.stringify(obj_2));

// shorthand usage of variables
const obj_3 = { a, b, c };
alert(JSON.stringify(obj_3));
```

Constructor

The `new Object()` constructor creates a new object.

```
"use strict";
const obj_4 = new Object({ a: 5, b: 42 });
alert(JSON.stringify(obj_4));
```

Object.create()

The `Object.create()` method creates a new object from an existing object.

```
"use strict";
const obj_5 = Object.create({ a: 5, b: 42 });
alert(JSON.stringify(obj_5));  // ???
```

```
alert(JSON.stringify(obj_5.a));
console.log (obj_5);
```

Read a property

Each property consists of a key-value pair. To read a property's values, you can use its key in one of two syntactical ways, *dot notation* or *bracket notation*.

```
"use strict";
const person = {firstName: "Albert", lastName: "Einstein" };

// dot notation
alert(person.firstName);   // no " " or ' '

// bracket notation
alert(person["firstName"]); // must use a string

// bracket notation using a variable
const fn = "firstName";
alert(person[fn]);
```

What if you don't know the keys? To get a list of all existing keys, use the method `Object.keys()`. It returns an array whose elements are strings. Those strings may be used in a loop to access the values. Because you need such loops in many cases, JavaScript offers a special language element for that situation. The `for .. in` loop selects the keys and loops over all properties.

```
"use strict";
const person = {firstName: "Albert", lastName: "Einstein" };
alert(Object.keys(person));   // firstName, lastName

// The for .. in loop selects all existing keys and loops over the properties
for (const k in person) {
  alert('The key is: ' + k + '. The value is: ' + person[k]);
}
```

Analog to the `Object.keys()` method there are the two methods `Object.values()` and `Object.entries()` to get the values respectively the properties (key and value). In the latter case, you get an array of arrays (with two elements each).

```
"use strict";
const person = {firstName: "Albert", lastName: "Einstein" };
alert(Object.values(person));   // Albert, Einstein

// for .. of loops differ from for .. in loops in syntax AND semantic; see chapter 'Loops'
for (const [k, v] of Object.entries(person)) {
  alert('The key is: ' + k + '. The value is: ' + v);
}
```

Add or modify a property

You can use the *dot notation* as well as the *bracket notation* to add or modify properties. There is no difference in the syntax between the above read operations and the write operations.

```
"use strict";
const person = {firstName: "Albert", lastName: "Einstein" };

// add properties
```

```
person.bornIn = "Ulm";
person["profession"] = "Physicist";
alert(JSON.stringify(person));

// modify properties
person.bornIn = "Germany";
person["profession"] = "Theoretical Physics";
alert(JSON.stringify(person));
```

In the above example, the variable 'person' is created with the keyword `const`. So it's not possible to assign a new value to it, but it's possible to manipulate its properties. When manipulating properties, the original object keeps unchanged.

Delete a property

The `delete` operator removes the complete property from an object - the key as well as the value. This is different from the case where someone stores `null` or `undefined` in the value.

Again, you can use the *dot notation* as well as the *bracket notation*.

```
"use strict";
const person = {firstName: "Albert", lastName: "Einstein", bornIn: "Ulm", profession:
"Physicist" };

delete person.bornIn;
delete person["profession"];
alert(JSON.stringify(person));
```

Merge objects

JavaScript offers a 'spread syntax' (3 dots). It expands an Object to its properties (key-value pairs) or an array to its elements. This can be helpful when you want to merge multiple objects into a single one.

```
"use strict";
const person = {firstName: "Albert", lastName: "Einstein" };
const address = {city: "Ulm" };

// expand the two objects (and merge them)
const newPerson = {...person, ...address};
alert(JSON.stringify(newPerson));
// The result is an object with 3 properties. All values are strings.
// {firstName: "Albert", lastName: "Einstein",city: "Ulm"}

// which is different from the version without the 'spread syntax':
const newPerson1 = {person, address};
alert(JSON.stringify(newPerson1));
// The result is an object with 2 properties. Both values are objects.
// {person: {firstName: "Albert", lastName: "Einstein"}, address: {city: "Ulm"}}
```

Functions/methods as part of an object

The value part of a property may contain the body of a function; see here (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_methods). In this case, the function is called a *method*.

```
"use strict";

// only definitions here, no execution!
const city = {
  showName: function (cityName) {
              if (typeof cityName === "undefined") {
                alert("Sorry, I don't know my name.");
              } else {
                alert("The name of the city is: " + cityName);
              }
            },

  // this works too!
  showPopulation(count) {
    alert(count + " Inhabitants");
  },
};

// JSON's 'stringify()' doesn't support the serialization of methods. Use 'console.log()'
instead.
console.log(city);

// executions
city.showName();
city.showName("Nairobi");
city.showPopulation(4500000);
```

Exercises

... are available on another page (click here).

Arrays

In JavaScript, an array is an object where you can store a set of values under a single variable name. So far, it's the same as in many other languages. But there are distinctions.

- It's not necessary that the values are of the same data type. You can put everything you want in an array and worry about the data type later. (But there are also typed arrays (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays) where all values have the same data type.)
- When you create an array, you do not need to declare a size - but you can. Arrays grow automatically. This makes arrays very convenient to use, but not well-suited for applications in numerical analysis.
- Because arrays are objects, they have methods and properties you can invoke at will. For example, the `.length` property indicates how many elements are currently in the array. If you add more elements to the array, the value of the `.length` gets larger.
- The element-counting starts at 0, which means, for instance, that the 5th element is located with [4].

(Hint: When using arrays, you should always use the *bracket notation* with non-negative integers, e.g., `arr[3] = 42;`. Technically, it's also possible to use the *dot notation*, but this leads - at least for beginners - to unexpected behavior.)

Create an array

First, as with all objects, there is a constructor.

```
"use strict";

const arr_1 = new Array(); // empty array
alert(arr_1.length);
const arr_2 = new Array(0, 2, 4); // 3 elements
alert(arr_2);
```

Next, the JavaScript syntax supports square brackets when creating or working with arrays.

```
"use strict";

const arr_1 = []; // empty array
alert(arr_1.length);
const arr_2 = [0, 2, 4]; // 3 elements
alert(arr_2);
```

You can predefine the size of an array when declaring it.

```
"use strict";
```

```
const arr_3 = new Array(50); // 50 elements
alert(arr_3.length);
```

Access an array element

Array elements are accessed for reading or writing with the usual bracket notation.

```
"use strict";

const arr_4 = [0, 2, 4, 6, 8];
alert(arr_4[3]);  // 6
arr_4[0] = 9;
alert(arr_4); // 9, 2, 4, 6, 8
```

When you access an element above the array's actual length, the size of the array will grow, and the new element will be created.

```
"use strict";

const arr_5 = [0, 2, 4, 6, 8];
arr_5[10] = 9;
alert(arr_5); // 0,2,4,6,8,,,,,,9
alert(arr_5.length); // 11
```

Varying data types

You can store values of different data types within an array.

```
"use strict";

const arr_6 = [0, "two", 4]; // number and string
console.log(arr_6);          // [0, "two", 4]

// and even values of data type 'array' can be stored
const arr_7 = [10, 11];
arr_7[2] = arr_6;            // array in array
console.log(arr_7);          // [10, 11, [0, "two", 4]]
console.log(arr_7.length); // 3
```

Nested arrays

As shown before, an array element may be an array (which itself may contain elements of type array (which itself may contain ...)). This can occur during runtime or during initialization. To access the lower levels directly, you must specify as many bracket pairs `[ ]` as necessary to reach this level.

```
"use strict";

const arr_8 = [ [0, 1], [10, 11, 12, 13], [20, 21] ];
console.log(arr_8[2]);    // one level goes to an array of numbers: [20, 21]
console.log(arr_8[1][1]); // two levels go to a number: 11

// same with assignments ...
arr_8[2][0] = "twenty";
console.log(arr_8[2]); // ["twenty", 21]
```

... and a little more complex

```
"use strict";

const arr_9 = []; // empty
arr_9[0] = [];
arr_9[0][0] = [];
arr_9[0][0][2] = "Hallo world!";
console.log(arr_9);  // [[[undefined, undefined, "Hallo world!"]]]

arr_9[2] = "Third element of first level";
console.log(arr_9);
// [[[undefined, undefined, "Hallo world!"]], undefined, "Third element of first level"]
```

Properties and methods

`length`

`length` is a property of each array (it's not a method). It represents the number of elements in that array.

```
alert([0, 1, 2].length);  // 3
```

Please notice that array indices are zero-based. Therefore the array's length is huger than the last index.

`concat`

The `concat` method returns the combination of two or more arrays. To use it, first, you need two or more arrays to combine.

```
const arr1 = ["a","b","c"];
const arr2 = ["d","e","f"];
```

Then, make a third array and set its value to `arr1.concat(arr2)`.

```
const arr3 = arr1.concat(arr2); //arr3 now is: ["a","b","c","d","e","f"]
```

Note that in this example, the new arr3 array contains the contents of both the arr1 array and the arr2 array.

`join` and `split`

The `join` method returns a single string that contains all of the elements of an array — separated by a specified delimiter. If the delimiter is not specified, it is set to a comma.

There is also a `split` method that performs the opposite of `join`: it operates on a string, divides him into elements based on the specified delimiter, and returns an array that contains those elements. (Hint: `split` is a method of the data type *string*, not of *array*.)

To use `join`, first make an array.

```
const abc = ["a", "b", "c"];
```

Then, make a new variable and assign it to `abc.join()`.

```
const a = abc.join();  // "a,b,c"
```

You can also use a dedicated delimiter.

```
// use 'semicolon' plus 'space' as delimiter
const b = abc.join("; ");  // "a; b; c"
```

Convert it back into an array with the string's `split` method.

```
const a2 = a.split(",");  // ["a", "b", "c"]
const b2 = b.split("; "); // ["a", "b", "c"]
```

push

The `push` method adds one or more elements to the end of an array and returns the array's new length.

```
"use strict";
const arr = [0, 1, 2, 3];

alert(arr);                 // 0, 1, 2, 3
const len = arr.push(100);
alert(len);                 // 5
alert(arr);                 // 0, 1, 2, 3, 100
```

pop

The `pop` method removes the last element of an array and returns the element.

```
"use strict";
const arr = [0, 1, 2, 3];

alert(arr);                 // 0, 1, 2, 3
const elem = arr.pop();
alert(elem);                // 3
alert(arr);                 // 0, 1, 2
```

`push` and `pop` work at the end of an array; they reverse their respective effects.

unshift

The `unshift` method adds one or more new elements to the beginning of an array and returns the array's new length. It works by 'unshifting' every old element from its old index i to $i + 1$, adds the new element to index 0, and adopts the array's `length` property. It is comparable with `push` but works at the beginning of the array.

```
"use strict";
const arr = [0, 1, 2, 3];

alert(arr);                 // 0, 1, 2, 3
const len = arr.unshift(100);
alert(len);                 // 5
alert(arr);                 // 100, 0, 1, 2, 3
```

`shift`

The `shift` method removes the first element of an array and returns the removed element. It works by 'shifting' every old element from its old index i to $i - 1$, adopts the array's `length` property, and returns the old first element. It is comparable with `pop` but works at the beginning of the array.

```
"use strict";
const arr = [0, 1, 2, 3];

alert(arr);                 // 0, 1, 2, 3
const elem = arr.shift();
alert(elem);                // 0
alert(arr);                 // 1, 2, 3
```

`unshift` and `shift` work at the beginning of an array; they reverse their respective effects.

Exercises

... are available on another page (click here).

See also

- MDN: Arrays (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

Dates

In JavaScript, a *Date* is an object. Hence it must be explicitly created with the `new` operator.

Date contains a value that represents the number of milliseconds that have elapsed since January 1, 1970 UTC. It's worth mentioning what it does not contain: There is no information about a timezone within the object. Nevertheless, you can convert it to an appropriate string of any arbitrary time zone. Other methods can select parts like the month or the day of the week, or you can use the number for any computation, and more.

Constructor

The default constructor creates the *Date* object as of the current point in time **of your computer**.

```
const currentMilliSeconds = new Date(); // creates a new Date object as of 'now'
alert(currentMilliSeconds);             // implicit conversion to string
alert(currentMilliSeconds.toString());  // explicit conversion to string

alert(currentMilliSeconds.valueOf());   // the real value
```

You can pass parameters to the constructor to generate a certain *Date* object.

```
// 1000 ms = 1 second after the beginning of JavaScript time
const pointInTime_1 = new Date(1000);
alert(pointInTime_1);
// begin of last minute in last century
const pointInTime_2 = new Date(1999, 11, 31, 23, 59);
alert(pointInTime_2);
```

Methods

Some often used methods (https://www.w3schools.com/js/js_date_methods.asp) of *Date* are:

Static methods

- Date.now(): Returns the number of milliseconds since January 1, 1970, 00:00:00 **UTC** calibrated (plus/minus some hours) to the timezone of **your computer**.
- Date.UTC(<parameters>): Returns the number of milliseconds since January 1, 1970, 00:00:00 **UTC**.
- Date.parse(text): Parsing of strings with Date.parse() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date#static_methods) is strongly discouraged due to browser differences and inconsistencies.

Instance methods

- toISOString(): Returns a string in ISO 8601 format.
- getFullYear(): Returns the full 4-digit year.
- getMonth(): Returns the current month. [0 - 11]
- getDate(): Returns the day of the month. [1 - 31]

- getDay(): Returns the day of the week. [0 - 6]. Sunday is 0, with the other days of the week taking the next value.
- getHours(): Returns hours [0 - 23] based on a 24-hour clock.
- getMinutes(): Returns minutes. [0 - 59]
- getSeconds(): Returns seconds. [0 - 59]
- getTime(): Returns the time in milliseconds since January 1, 1970.
- valueOf(): Returns the time in milliseconds since January 1, 1970. Equivalent to getTime().
- getTimezoneOffset(): Returns the difference in minutes between UTC and local time.
- setFullYear(year): Stores the full 4-digit year within the Date object.
- setMonth(month, day): Sets the month, and optionally the day within the month. '0' is January, ...

'As Integer'

The *Date* can be returned as an integer by the method `valueOf()` or by prefixing the constructor with a `+` sign, e.g., to use it for "seeding" a PRNG (Pseudo Random Number Generator) or to perform calculations.

```
const dateAsInteger_1 = new Date().valueOf();
alert(dateAsInteger_1);

const dateAsInteger_2 = +new Date();
alert(dateAsInteger_2);
```

Timezones

The *Date* object purely contains a whole number (Integer). It does not know anything about timezones. As long as you don't specify timezones in any form, you work in your local timezone.

But sometimes it's necessary to consider timezones.

```
// Example 1

// Create it in UTC: 27th of January, midnight
const date_1 = new Date(Date.UTC(2020, 00, 27, 0, 0, 0));
alert(date_1);

// show it in another timezone (Jakarta is +7) ...
const jakartaTime = new Intl.DateTimeFormat('en-GB', { timeZone: 'Asia/Jakarta', dateStyle:
'full', timeStyle: 'long' }).format(date_1);
alert(jakartaTime);

// ... the original value has not changed
alert(date_1);
```

```
// Example 2

// assume we are in New York timezone (-5 against UTC)
const date_2 = new Date();
console.log(date_2.toString());

// show it in another timezone (Los Angeles is -8 against UTC)
const laTime = new Intl.DateTimeFormat('en-GB', { timeZone: 'America/Los_Angeles', dateStyle:
'full', timeStyle: 'long' }).format(date_2);
console.log(laTime);
```

```
// ... the internal value has not changed
console.log(date_2.toString());
```

New API: Temporal

The object *Date* has some inconsistencies and shortcomings, especially: weak timezone support, no support for non-Gregorian calendars, missing calendar functions, and more. Possibly they will be fixed in a new Temporal API (https://tc39.es/proposal-temporal/docs/index.html) in one of the subsequent JavaScript versions.

Exercises

... are available on another page (click here).

Regular expressions

A regular expression - **RE** or **RegExp** for short - is a pattern that may or may not correlate with a given string, i.e. (in pseudo-code): "a single digit followed by one to five letters". The string "3Stars" correlates, while "Three Stars" does not.

Regular expressions are notated in their own syntax. The above pseudo-code must be expressed as `/\d\w{1,5}/`, which is a literal delimited by a slash at the beginning and the end, and contains the formal description for the words 'digit' `\d`, 'letter' `\w`, and 'one to five' `{1,5}`.

From the perspective of JavaScript, *RegExp* is a data type that is derived from *Object*. It has its own constructors, methods, and properties. But their direct usage in the form of an object is a little counterintuitive. Because the Wikibook on hand addresses JavaScript beginners, we explain the advantage and appliance of regular expressions in the context of strings, which is the standard use case. Also, we show only simple regular expressions and central aspects. For more details on regular expressions, there is a separate Wikibook Regular Expressions. The focus here is on their usage within JavaScript.

Regular expressions most commonly appear in conjunction with the `match` and `replace` methods of strings.

String's `match` method

The `match` method accepts an RE and returns an array with the matching string(s).

```
"use strict";
const myVar = "A short text";

// show the result of the match()
alert(myVar.match(/t t/));  // note the slashes

// match() accepts the RE as String ("..." instead of /.../)
alert(myVar.match("t t"));

// explicit constructor of an RE
const re = new RegExp("t t");
alert(myVar.match(re));
```

Make decisions

If the `match` doesn't find something in the string, it returns `null`. Because JavaScript treats `null` as `false`, the result can be used in an `if` statement.

```
"use strict";
const myVar = "A short text";
if (myVar.match(/t t/)) {
  alert("Bingo.");
} else {
```

```
  alert("Bad luck.");
}
```

Meta-characters

REs are very powerful through the use of meta-characters instead of concrete characters. Meta-characters extend the meaning of REs in different ways: indicating character classes, wildcards, quantifiers, groups, back-references, and much more.

Wildcard

You can use the `.` (dot) as a wildcard for any character.

```
"use strict";
const myVar = "A short text";
alert(myVar.match(/t..t/));      // text
```

Quantifier

There are three quantifiers:

? The question mark indicates zero or one occurrence of the preceding element.
* The asterisk indicates zero or more occurrences of the preceding element.
+ The plus sign indicates one or more occurrences of the preceding element.

```
"use strict";
const myVar = "aaa bbb ccc 789";
alert(myVar.match(/bbb c?/)); // bbb c    one 'c'
alert(myVar.match(/bbb x?/)); // bbb      zero 'x'
alert(myVar.match(/bbb c*/)); // bbb ccc  multiple 'c's
alert(myVar.match(/bbb c+/)); // bbb ccc  multiple 'c's

// combination of wildcard and quantifier
alert(myVar.match(/bbb .*/)); // bbb ccc 789   multiple arbitrary characters
```

Modifier

The semantics of the RE may be modified by appending one of `g`, `i`, or `m` to the end (after the second slash) of the RE.

g (global) indicates that all occurrences of the RE shall be returned, not only the first one.
i (ignore case) indicates that the case of characters shall be ignored.
m (multiple lines) indicates that the RE shall work across line terminators.

```
"use strict";
const myVar = "A short text that contains 7 words.";
alert(myVar.match(/t t./g));   // t te, t th
```

```
alert(myVar.match(/SHORT/));   // null
alert(myVar.match(/SHORT/i));  // short
```

Character classes

Some important *character classes* (or *character types*) are identified by the following backslash-notations:

\w	An alphanumeric character or "_"
\W	An non-alphanumeric character or "_"
\d	A digit
\D	A non-digit
\s	A whitespace character (space, tab, newline, formfeed)
\S	A non-whitespace character
\b	A boundary of a word

```
"use strict";
const myVar = "a+ a a3 ";
alert(myVar.match(/a\w/));   // a3
alert(myVar.match(/a\W/));   // a+
alert(myVar.match(/\d/));    // 3
alert(myVar.match(/a\s/));   // a
alert(myVar.match(/a\b/));   // a
```

String's `replace` method

The `replace` method returns a new string where the RE is replaced by the given string.

```
"use strict";
const myString1 = "Hello world.";
const myString2 = myString1.replace(/world/, "Tim");
alert(myString2);  // Hello Tim.
```

Exercises

... are available on another page (click here).

See also

- Regular Expressions - a Wikibook dedicated to regular expressions.
- Perl Regular Expressions Reference - a chapter devoted to regular expressions in a book about the Perl programming language.

External links

- MDN: Regular Expressions in JavaScript - 1 (http://developer.mozilla.org/en/JavaScript/Guide/Regular_Expressions)
- MDN: Regular Expressions in JavaScript - 2 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp)

- W3Schools: JavaScript RegExp Reference (http://www.w3schools.com/jsref/jsref_obj_regexp.asp)
- JavaScript RexExp Tester (http://www.regular-expressions.info/javascriptexample.html) at regular-expressions.info

Operators

String concatenation

The + operator acts in two different ways depending on the type of its two operands. If one or both of them are strings, it acts as a string concatenator. If both are numeric, it acts as an arithmetic addition.

An example of string concatenation is `"one " + "world"` resulting in `"one world"`. If one of the two operands isn't a string, it is implicitly converted to a string before the + operation takes place.

```
"use strict";

// regular concatenation
const word_1 = "one";
const word_2 = "world";
let   result = word_1 + " " + word_2;
alert(result);        //  "one world"

// implicit type conversion
const arr = [41, 42, 43];
result = word_1 + arr + word_2;   // first, the array is converted to a string
alert(result);        //  "one41,42,43world"

// dito
const x = 1;
result = x + word_2;
alert(result);        //  "1world"
```

Arithmetic operators

JavaScript has the arithmetic operators `+`, `-`, `*`, `/`, `%` (remainder), and `**` (exponentiation). These operators work the way you learned in mathematics. Multiplication and division will be calculated before addition and subtraction. If you want to change such precedence, you can use parenthesizes.

Hint: In opposite to some other languages, the division operation may result in a floating point number - it's not always a pure integer.

```
const a = 12 + 5;       // 17
const b = 12 - 5;       // 7
const c = 12 * 5;       // 60
const d = 12 / 5;       // 2.4   Division results in floating point numbers
const e = 12 % 5;       // 2     Remainder of 12/5 is 2
const f = 12 - 2  * 5;  // 2     Multiplication is calculated first
const g =(12 - 2) * 5;  // 50    Parenthesis are calculated first
```

Some mathematical operations, such as dividing by zero, cause the returned variable to be one of the error values - for example, `infinity`, or `NaN` (Not a number).

The return value of the remainder operator maintains the sign of the first operand.

The `+` and `-` operators also have unary versions, where they operate only on one variable. When used in this fashion, `+` returns the number representation of the object, while `-` returns its negative counterpart.

```
"use strict";
const a = 42;
const b = +a;    // b is 42
const c = -a;    // c is -42
```

As noted above, `+` is also used as the string concatenation operator: If any of its arguments is a string or is otherwise *not* a number, all arguments are converted to strings and concatenated together. All other arithmetic operators works the other way round: They attempt to convert their arguments into numbers before evaluating.

```
"use strict";
const a = 42;
const b = "2";
const c = "3";
alert(a + b);   //  "422"  (a string)
alert(a * b);   //  84     (a number)
alert(b * c);   //  6      (a number)
```

Bitwise operators

There are seven bitwise operators: `&`, `|`, `^`, `~`, `>>`, `<<`, and `>>>`.

These operators convert their operands to integers (truncating any floating point towards 0), and perform the specified bitwise operation on them. The logical bitwise operators, `&`, `|`, and `^`, perform the *and*, *or*, and *xor* on each individual bit and provides the return value. The `~` (*not* operator) inverts all bits within an integer, and usually appears in combination with the logical bitwise operators.

Two bit shift operators, `>>`, `<<`, move the bits in one direction that has a similar effect to multiplying or dividing by a power of two. The final bit-shift operator, `>>>`, operates the same way, but does not preserve the sign bit when shifting.

These operators are kept for parity with the related programming languages, but are unlikely to be used in most JavaScript programs.

Assignment operators

The assignment operator `=` assigns a value to a variable. Primitive types, such as strings and numbers, are assigned directly. However, function and object names are just pointers to the respective function or object. In this case, the assignment operator only changes the reference to the object rather than the object itself. For example, after the following code is executed, "0, 1, 0" will be alerted, even though `array_A` was passed to the alert, but `array_B` was changed. This is because they are two references to the same object.

```
"use strict";
const array_A = [0, 1, 2];
const array_B = array_A;
array_B[2] = 0;
alert(array_A);  // 0, 1, 0
```

Similarly, after the next code snippet is executed, `x` is a pointer to an empty array.

```
"use strict";
const z = [5];
```

```
const x = z;
z.pop();
alert (x);
```

If the result of any of the above-shown arithmetic or bitwise operators shall be assigned to its first operand, you can use a shorter syntax. Combine the operator, e.g., `+`, directly with the assignment operator `=` resulting in `+=`. As an example `x = x + 5` can be abbreviated as `x += 5`.

The abbreviated operator/assignment syntax reads:

Arithmetic	Logical	Shift
+=	&=	>>=
-=	|=	<<=
*=	^=	>>>=
/=		
%=		

For example, a common usage for `+=` in a `for` loop:

```
"use strict";
const arr = [1, 2, 3];
let sum = 0;
for (let i = 0; i < arr.length; i++) {
  sum += arr[i];  // same as: sum = sum + arr[i];
}
alert(sum);
```

Increment operators

Increment and decrement operators are a particular form of arithmetic operators: `++` and `--`. `a++` increments `a` and returns the old value of `a`. `++a` increments `a` and returns the new value of `a`. The decrement operator acts similarly but reduces the variable instead.

As an example, the next operations all perform the same task:

```
"use strict";
let a = 1;
alert(a);     // 1

a = a + 1;
alert(a);     // 2

a += 1;
alert(a);     // 3

a++;
alert(a);     // 4

++a;
alert(a);     // 5

// but this SHOWS something different; see next chapter
```

```
alert(a++);   // shows 5 again
alert(a);     // nevertheless, 'a' was incremented to 6
```

Pre and post-increment operators

Increment operators may be written before or after a variable. The position decides whether they are pre-increment or post-increment operators, respectively. That affects the operation.

```
"use strict";

// increment occurs before a is assigned to b
let a = 1;
let b = ++a;  // a = 2, b = 2;

// increment occurs to c after c is assigned to d
let c = 1;
let d = c++;  // c = 2, d = 1;
```

Due to the possibly confusing nature of pre and post-increment behavior, code can be easier to read if you avoid increment operators.

```
"use strict";

// increment occurs before a is assigned to b
let a = 1;
a += 1;
let b = a;  // a = 2, b = 2;

// increment occurs to c after c is assigned to d
let c = 1;
let d = c;
c += 1;     // c = 2, d = 1;
```

Comparison operators

Comparison operators determine whether their two operands meet the given condition. They return *true* or *false*.

Concerning the 'equal' and 'not-equal' operators, you must take care. `==` is different from `===`. The first one tries to adapt the data type of the two types to each other and then compares the values. The second one compares the types as well as their values and returns only *true* if type and value are identical: `3 == "0003"` is *true* and `3 === "3"` is *false*.

Op.	Returns	Notes
==	true, if the two operands are equal	May change an operand's type (e.g. a string as an integer).
===	true, if the two operands are **strictly** equal	Does not change operands' types. Returns *true* if they are the same type **and** value.
!=	true, if the two operands are not equal	May change an operand's type (e.g. a string as an integer).
!==	true, if the two operands are not **strictly** equal	Does not change the operands' types. Returns *true* if they differ in type **or** value.
>	true, if the first operand is greater than the second one	
>=	true, if the first operand is greater than or equal to the second one	
<	true, if the first operand is less than the second one	
<=	true, if the first operand is less than or equal to the second one	

We recommend using the **strict** versions (`===` and `!==`) because the simple versions may lead to strange and non-intuitive situations (https://dorey.github.io/JavaScript-Equality-Table/), such as:

```
0 == ''            // true
0 == '0'           // true
false == 'false'   // false (''Boolean to string'')
false == '0'       // true  (''Boolean to string'')
false == undefined // false
false == null      // false (''Boolean to null'')
null == undefined  // true
```

... although you might want:

```
0 === ''            // false
0 === '0'           // false
false === 'false'   // false
false === '0'       // false
false === undefined // false
false === null      // false
null === undefined  // false
```

Logical operators

The logical operators are *and*, *or*, and *not* — written as `&&`, `||`, and `!`. The first two take two boolean operands each. The third takes one and returns its logical negation.

Operands are boolean values or expressions that evaluate to a boolean value.

```
"use strict";
const a = 0;
const b = 1;

if (a === 0 && b === 1) {  // logical 'and'
  alert ("a is 0 AND b is 1");
}

if (a === 1 || b === 1) {  // logical 'or'
```

```
  alert ("a is 1 OR b is 1");
}
```

`&&` and `||` are *short circuit* operators: if the result is guaranteed after the evaluation of the first operand, it skips the evaluation of the second operand. Due to this, the `&&` operator is also known as the guard operator, and the `||` operator is known as the default operator.

```
"use strict";

// declare 'myArray' without initialization
let myArray;

// runtime error!
if (myArray.length > 0) {
  alert("The length of the array is: " + myArray.length);
}

// no error because the part after '&&' will not be executed!
if (myArray && myArray.length > 0) {
  alert("The length of the array is: " + myArray.length);
}
```

The `!` operator determines the inverse of the given boolean value: *true* becomes *false* and *false* becomes *true*.

Note: JavaScript represents false by either a Boolean false, the number 0, NaN, an empty string, or the built-in types undefined or null. Any other value is treated as true.

Concerning the precedence of the three operators, `!` is evaluated first, followed by `&&`, and lastly `||`.

```
      a  ||   b  &&   !c

    |       |       |    |
    |       |       └ 1. ┘
    |       └────── 2. ───┘
    └────────── 3. ────────┘
```

More details on the relationship between precedence and short-circuiting are explained at MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#short-circuiting)

Other operators

`? :`

The `? :` operator (also called the "ternary" operator) is an abbreviation of the `if` statement. First, it evaluates the part before the question mark. Then, if *true*, the part between the question mark and the colon is evaluated and returned, else the part behind the colon.

```
const target = (a == b) ? c : d;
```

However, be careful when using it. Even though you can replace verbose and complex if/then/else chains with ternary operators, it may not be a good idea to do so. You can replace

```
if (p && q) {
    return a;
} else {
    if (r != s) {
        return b;
    } else {
        if (t || !v) {
            return c;
        } else {
            return d;
        }
    }
}
```

with

```
return (p && q) ? a
  : (r != s) ? b
  : (t || !v) ? c
  : d
```

The above example is a poor coding style/practice. When other people edit or maintain your code, (which could very possibly be you,) it becomes much more difficult to understand and work with the code.

Instead, it is better to make the code more understandable. Some of the excessive conditional nesting can be removed from the above example.

```
if (p && q) {
    return a;
}
if (r != s) {
    return b;
}
if (t || !v) {
    return c;
} else {
    return d;
}
```

`delete`

`delete obj.x` unbinds property `x` from object `obj`.

The delete keyword deletes a property from an object. It 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.

`new`

`new cl` creates a new object of type *cl*. The *cl* operand must be a constructor function.

`instanceof`

`o instanceof c` tests whether `o` is an object created by the constructor `c`.

typeof

`typeof x` returns a string describing the type of `x`. Following values may be returned.

Type	returns
boolean	"boolean"
number	"number"
string	"string"
function	"function"
undefined	"undefined"
null	"object"
others (array, ...)	"object"

Exercises

... are available on another page (click here).

See also

- MDN: Operators (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/)

Control structures

Most programming languages are composed of 'bricks' like tokens (keywords, variables, operators, ...), expressions like `myArray.length + 1`, statements (delimited by `;`), blocks `{...}`, functions, and modules. At first glance, the execution of the program follows the sequence of statements, top down. But in nearly all cases, it is necessary that the program does not run in the strict order of the written statements. Instead, some parts must run only if certain conditions apply, and others will be omitted and run under different conditions. Or, it may become necessary that some parts are executed repetitively. Other parts may run in parallel and get synchronized later. Or, a function of a different module must compute a value before the next statement can be executed.

In this hierarchy of 'language bricks' the term **block** is essential for the understanding of the program flow. In JavaScript, a block is a sequence of zero or more statements (or smaller blocks) that are surrounded by braces `{ // zero or more statements }`. The language constructions we discuss here invoke or repeat blocks.

if / else

The `if / else` statement (yes, it's a single statement, even though it contains other statements in its blocks) invokes the execution of one of two blocks depending on the evaluation of a condition. The evaluation returns a boolean value. If it is `true`, the first block is executed; if it is `false`, the second block is executed. The respectively other block is skipped over.

```
if ( condition ) {
  // block of statements
} else {
  // block of statements
}
```

The `else` part is optional, i.e. it's possible to use `if` without the `else` part and its block.

```
if ( condition ) {
  // block of statements
}
```

An example:

```
"use strict";
const a = 3;
const b = "3";
if (a == b) {
  alert("The two variables contains the same value, but may have different data types.");
} else {
  alert("The two variables contain different values.");
}

// an example without 'else'
const c = 6 / 2;
if (a === c) {
  alert("The two variables contains the same value and are of the same data type.");
}
```

If one of the two blocks contains exactly ONE statement, the braces can be omitted. But for the clearness of the code, we recommend the use of a unified syntax with braces.

```
// same as above; but this abbreviated syntax is not recommended.

"use strict";
const a = 3;
const b = "3";

if (a == b) alert("The two variables contains the same value, but may have different data
types.");
else alert("The two variables contain different values.");

const c = 6 / 2;
if (a === c) alert("The two variables contains the same value and are of the same data
type.");
```

In many cases, the situation demands more complex decisions than a simple true/false alternative. For example, you may want to know whether a number is negative, zero, or positive. In such cases, a solution might look like this:

```
"use strict";
const x = 3;

if (x < 0) {
  alert("The number is negative.");
} else {
  // x is equal or greater than 0
  if (x === 0) {
    alert("The number zero.");
  } else {
    alert("The number is positive.");
  }
}
```

You can shorten this code a bit without losing clarity. Because the first `else` block contains only a single statement - namely the second `if` - you can omit its braces and combine the first `else` and the second `if` within one line.

```
"use strict";
const x = 3;

if (x < 0) {
  alert("The number is negative.");
} else if (x === 0) {
  alert("The number is zero.");
} else {
  alert("The number is positive.");
}
```

This is a clear and often-used programming style. It's used in situations where you have a manageable number of choices or where you have to make decisions with multiple variables.

switch

If the number of decisions grows significantly, the code gets clearer if you use the `switch` statement instead of a long list of `else if` conditions.

The `switch` statement evaluates an expression and steers the flow of statements based on the comparison of its result with the labels behind the keyword `case`.

```
"use strict";

const myVar = "a";

// evaluation takes simple variables as well as complex expressions
switch (myVar.toUpperCase()) {
case "A":
  // …
  break;
case "B":
  // …
  break;
default:   // analog to 'else' without any further 'if'
  // …
  break;
}
```

If the result of the evaluation matches one of the labels, JavaScript executes the following statements up to the next `break` or the end of the entire `switch`. If none of the labels match, execution continues at the `default` label, or - if none is present - skips the `switch` statement entirely.

Labels are literals or expressions; e.g., `case (2 + 1).toString():` is possible.

As soon as a `break` statement is reached, the execution of the `switch` gets terminated. Normally it appears at the end of each case to prevent execution of the code of the following cases. But it can be omitted if you intentionally want to execute them in addition to the current ones. In the following example, the same code will run for `i` equal to 1, 2, or 3.

```
"use strict";

const i = 2;

switch(i) {
case 1:
case 2:
case 3:
  // …
  break;
case 4:
  // …
  break;
default:
  // …
  break;
}
```

Because the expression to be evaluated as well as the labels can be complex expressions, it's possible to build very flexible constructions.

```
"use strict";

const i = 2;

switch(true) { // in this example it's a constant value
case (i < 10):
  alert("one digit");
  break;
case (i >= 10 && i < 100):
  alert("two digits");
  break;
```

```
default:
  // …
  break;
}
```

The `continue` keyword does not apply to the `switch` statement.

try / catch / finally

If there is a possibility that a runtime error might occur, you can 'catch' that error and perform meaningful actions to handle the situation. E.g., a network connection or a database might no longer be available; a user input leads to a division by zero; ... .

```
try {
  // critical block where errors might occur
} catch (err) {
  // block to handle possible errors. Normally not executed.
} finally {
  // block that will be executed in ALL cases
}
```

```
"use strict";

const x = 15;
let average;
try {
  // block with critical statements
  x = x + 5;
  average = x / 0;
  alert("The average is: " + average);
} catch (err) {
  // block to handle possible errors
  alert("Something strange occurs. The error is: " + err);
} finally {
  // block that will be executed in ALL cases
  alert("End of program.");
}
```

If one of the statements in the *critical* block raises a runtime error, the execution of its remaining statements is omitted. Instead, the execution invokes the *catch* block. Lastly, the *finally* block is executed.

Please note that the *finally* block is executed in all cases, regardless of whether a runtime error occurs or not. That even applies if the *critical* or the *catch* block executes a `return` statement.

throw

In the above example, the JavaScript engine throws an exception by itself. In other situations, the JavaScript engine acts in one way or another, but you may want to see it treated differently. E.g., in the case of a division by zero, the engine doesn't throw an error; it assigns `Infinity` to the result and jumps to the following statement. If you want a different behavior, you can create and throw exceptions by your own program.

```
"use strict";

const x = 15;
let average;
try {
```

```
  // block with critical statements
  average = x / 0;
  // or: const z = "abc"; average = z / 0;
  if (average === Infinity || Number.isNaN(average)) {
    // Throw your own exception with any text
    throw "Error during division. The result is: " + average;
  }
  alert("The average is: " + average);
} catch (err) {
  // block to handle possible errors
  alert("Something strange occurs. The error is: " + err);
} finally {
  // block that will be executed in ALL cases
  alert("End of program.");
}
```

If an exception occurs - generated by the JavaScript engine or by your program - and is not caught by a *catch* block, the script terminates or - if it is a function - it returns control to the calling function. The error handling may be implemented there or in one of the functions which have been called it.

```
"use strict";

const answer = prompt("How old are you?");
const age = Number(answer);

if (isNaN(age)) {
  throw answer + " cannot be converted to a number.";
  // The script terminates with this message (it's not a function)
}
alert("Next year you will be " + (age + 1));
```

Exercises

... are available on another page (click here).

Loops

Loops and iterations are other cases where the sequential flow of statements is manipulated by surrounding language constructs. This is described on the next page.

See also

- MDN: Control program flow and error handling (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling)

Loops

JavaScript supports the repetitive execution of code blocks via the keywords `for` and `while`. Similar behavior is offered by the `forEach` method of data type `Array` (and similar data types).

for (;;) {}

The syntax of the `for` statement is: `for (<initial expression>; <condition>; <final expression>) { <block> }`. It is executed based on the following rules:

1. The `initial expression` is executed - exactly once for the complete `for` statement. Usually, it declares and initiates one or more variables - often numbers with integer values. Sometimes the declaration of the variables is made above the `for` statement.
2. The `condition` is evaluated. If it returns `true`, step 3 follows. If it returns`false`, the `for` statement terminates.
3. The code block is executed.
4. The `final expression` is executed. Usually, it increments or decrements a variable that is part of the `condition`.
5. The loop repeats at step 2.

```
for (let i = 0; i < 10; i++) {
  // a block of statements
}
```

An example: Show the series of even numbers multiplied by itself (power 2); show the sum over the numbers from 0 to 10.

```
"use strict";

const upperLimit = 10;
let sum = 0;
let tmpString = "";

for (let i = 0; i <= upperLimit; i++) {
  sum = sum + i;
  if (i % 2 === 0) {
    tmpString += i*i + "; "
  }
}
alert ("The sum is: " + sum + ". The square numbers are: " + tmpString);
```

Optional syntax parts

The `initial expression`, the `condition`, and the `final expression` are optional. If you omit one or more of them, your script shall perform other useful statements to control the loop. Some examples:

```
"use strict";

// an infinite loop if you do not terminate it with 'break' (see below)
for (;;) {
```

```
  // ...
  break;
}

const answer = prompt("With which start value shall the loop begin?");
let i = Number(answer);
for (; i >= 0 && i < 10; i++) {
  // the start value is computed above the loop
}

for (let i = 0; i < 10; ) {
  // ...
  if (true) { // an arbitrary other condition to control the increment
    i++;
  }
}
```

Nesting

`for` loops can be nested. You can use a second (or 'inner') loop within the block of the first (or 'outer') loop.

```
"use strict";

const maxInner = 10;
const maxOuter = 4;
let myString = "";

for (let o = 1; o <= maxOuter; o++) {
  myString = "";
  // Be careful. It's easy to confuse inner and outer loop variables.
  for (let i = 1; i <= maxInner; i++) {
    myString = myString + o*i + ", ";
  }
  alert(myString);
}
```

This nesting of loops is also possible in all the below constructs.

continue / break

Sometimes only parts or the block shall run. This can be realized by one or more `if / else` statements. If it's appropriate, such conditional statements can be shortened by the keyword `continue`. If `continue` is reached, the rest of the block is skipped, and the above step 4 `final expression` is executed.

```
"use strict";

for (let i = 0; i <= 6; i++) {
  if (i === 5) {
    continue;   // skip the rest of the block
  }
  alert(i);     // 0, 1, 2, 3, 4, 6
}
```

This example is very simple. It skips the lower part of the block for the case where `i` equals to 5. Of course, it can be expressed differently. A more realistic example would be a loop over the result of a database search which skips parts of a complex handling of rows with a specific status.

The `break` keyword is similar but more rigid than the `continue` keyword. It skips not only the rest of the block but also terminates the complete loop.

```
"use strict";

for (let i = 0; i <= 6; i++) {
  if (i === 5) {
    break;      // terminal the loop
  }
  alert(i);     // 0, 1, 2, 3, 4
}
```

A realistic scenario is a loop over the result of a database search where the connection to the database is broken in the middle of the loop.

You can use `continue` and `break` in all forms of the here-discussed variants of loops.

do {} while ()

The syntax of the statement is: `do { <block> } while (<condition>)`. It is executed based on the following rules:

1. The block is executed. Because this is the very first step, the block is executed at least 1 time. This is helpful if you want to be sure that something happens under all circumstances.
2. The `condition` is evaluated. If it returns `true`, step 1 is invoked again. If it returns`false`, the loop terminates. Please notice that there is no specific part where you can manipulate a variable that is checked in the condition. This must be done somewhere in the block among the other statements.

```
"use strict";

let counter = 100;
do {
  counter++;
  alert(counter);  // ... or some logging
} while (counter < 10);
```

while () {}

The syntax of `while (<condition>) { <block> }` is very similar to the previous `do { <block> } while (<condition>)`. The only difference is that the condition is checked before and not after the block.

for (x in Object) {}

This language construct is intended to iterate over the properties of an object. Its syntax is: `for (<variable> in <object>) { <block> }`. The variable receives the **key** of all properties - one after the next - of the object. For each of them, the block is executed once.

```
"use strict";

const myObj = {firstName: "Marilyn", familyName: "Monroe", born: 1953};
```

```
for (const key in myObj) {
  alert(key); // firstName, familyName, born
  // alert(myObj[key]);  // if you want to see the values
}
```

Arrays are specialized objects. Hence it is possible to use the `for..in` on arrays. The keys for arrays are integers starting with 0 - and that is precisely what is extracted from the array.

```
"use strict";

const myArray = ["certo", "uno", "dos", "tres"];
for (const key in myArray) {
  alert(key); // 0, 1, 2, 3
  // alert(myArray[key]);  // if you want to see the values
}
```

Arrays also accept non-numeric keys, e.g., `myArray["four"] = "cuatro";`. The `for..in` loop handles such non-numeric keys - in opposite to the below `for..of` loop, and in opposite to the traditional `for` loop at the top of this page.

See also: MDN: for..in (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in)

for (x of Array) {}

This language construct is intended to iterate over an array. Its syntax is: `for (<variable> of <iterable object>) { <block> }`. The variable receives all **values** - one after the next - of the array. For each of them, the block is executed once.

This definition sounds similar to the above definition of the `for..in`. But there are significant differences:

- It returns the **values**, not the **keys** resp. **index** of the array.
- It works only on all *iterable* objects (Array, Map, Set, ...). The data type 'object' is not iterable.
- It works only on such keys which are numeric. Non-numerical **keys** resp. **index** are silently ignored.

```
"use strict";

const myArray = ["cero", "uno", "dos", "tres"];
myArray["four"] = "cuatro";

for (const myValue of myArray) {
  alert(myValue); // cero, uno, dos, tres. No 'cuatro' because the key is a string.
}
```

See also: MDN: for..of (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of)

for..in vs. for..of

The differences between the two language constructs are summarized in an example.

```
"use strict";

const myObj = {firstName: "Marilyn", familyName: "Monroe", born: 1953};
const myArray = ["cero", "uno", "dos", "tres"];
myArray["four"] = "cuatro";

// for..of not allowed on true objects; only for..in is possible.
for (const x of myObj) {};  // error

for (const x in myArray) {
  alert(x); // 0, 1, 2, 3, four
}

for (const x of myArray) {
  alert(x); // cero, uno, dos, tres. NO cuatro!
}
```

Object.entries() method

If you are looking for a loop that processes both keys and values of an object, there is a more straightforward method than the above-shown combination of the `for..in` with `myObj[key]`. The data type *Object* offers the `entries()` method that returns an array. Each element of this array contains an array with two values: the property key and the property value. In other words: the return value of the `entries()` method is a two-dimensional array. And - as usual with arrays - you should use it in combination with `for..of`.

```
"use strict";

const myObj = {firstName: "Marilyn", familyName: "Monroe", born: 1953};
const myArray = ["cero", "uno", "dos", "tres"];
myArray["four"] = "cuatro";

for (const [key, val] of Object.entries(myObj)) {
  alert(key + ' / ' + val);
}
for (const [key, val] of Object.entries(myArray)) {
  alert(key + ' / ' + val); // 'four / cuatro' is included
}
```

Hint: The `Object.entries()` as well as the following `Array.forEach()` methods are no 'core' language elements like keywords or the `for..in` or `for..of` language constructs. They are methods of the *Object* respectively *Array* data type.

Array.forEach() method

The data type *Array* offers the method `forEach()`. It loops over the elements of the array, returning one element after the next. The interesting point is that it takes a function as its argument. This makes the difference to the divers `for` and `while` loops. Such functions are often called **callback function**.

The method invocation is very easy: `myArray.forEach(myFunction)`. The - possibly - confusing part is that function calls can be abbreviated in various ways.

The first example shows the function call explicitly. It defines the function *myFunction* which also takes a single argument. When *myFunction* is called by forEach(), the next array element is inserted as the argument to *myFunction*.

```
"use strict";

// function definition (the function is not called here!)
function myFunction(element) {
  alert("The element of the array is: " + element)
};

// a test array
const myArray = ['a', 'b', 'c'];

// iterate over the array and invoke the function once per array element
myArray.forEach(myFunction);
```

The following examples show some abbreviations of the function invocation syntax.

```
"use strict";

// the 'arrow' syntax
const myFunction =
  (element) => {
    alert("The element of the array is: " + element)
  };

// same code without line breaks:
// const myFunction = (element) => {alert("The element of the array is: " + element)};

const myArray = ['a', 'b', 'c'];
myArray.forEach(myFunction);
```

```
"use strict";
const myArray = ['a', 'b', 'c'];

// Define the function directly as the argument of the forEach(). Such
// functions are called 'anonymous' functions.
myArray.forEach((element) => {alert("The element of the array is: " + element)});
```

```
"use strict";
const myArray = ['a', 'b', 'c'];

// in simple cases, more syntactical elements are optional
myArray.forEach(element => alert("The element of the array is: " + element));
```

It depends on your preference, which syntax you use.

In many cases, the called function performs side effects like logging. To calculate values over all array elements, it is necessary to use the technique of closures. In the following example, the variable `sum` is not defined in the outer context, not in the anonymous function.

```
"use strict";

const myArray = [3, 0, -1, 2];
let sum = 0;

myArray.forEach(element => sum = sum + element);
alert(sum);
```

All in all, the following rules apply to the `forAll()` method:

- The method can be used for iterable objects like *Array*, *Map*, *Set*. The data type *Object* is not iterable.

- The method only iterates over such elements which have a numerical key - what is the usual case for arrays.

See also:

> MDN: forEach() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)
> MDN: Iterative methods (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#iterative_methods)

Exercises

> ... are available on another page (click here).

See also

- MDN: Loops and iterations (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration)

Functions

A function is a block of code that solves a dedicated problem and returns the solution to the calling statement. The function exists in its own context. Hence, functions divide massive programs into smaller 'bricks' which structure the software as well as the software development process.

```
// define a function
function <function_name> (<parameters>) {
  <function_body>
}

// call a function
<variable> = <function_name> (<arguments>);
```

JavaScript supports the software development paradigm *functional programming*. Functions are a data type derived from Object; they can be bound to variables, passed as arguments, and returned from other functions, just as any other data type.

Declaration

Functions can be constructed in three main ways. The first version can be abbreviated further; see below.

The conventional way:

```
"use strict";

// conventional declaration (or 'definition')
function duplication(p) {
  return p + "! " + p + "!";
}

// call the function
const ret = duplication("Go");
alert(ret);
```

Construction via a variable and an expression:

```
"use strict";

// assign the function to a variable
let duplication = function (p) {
  return p + "! " + p + "!";
};

const ret = duplication("Go");
alert(ret);
```

Construction via the `new` operator (this version is a little cumbersome):

```
"use strict";

// using the 'new' constructor
let duplication = new Function ("p",
  "return p + '! ' + p + '!'");
```

```
const ret = duplication("Go");
alert(ret);
```

Invocation

For the declaration of functions, we have seen 3 variants. For their invocation, there are also 3 variants. The declarations and invocations are independent of each other and you can arbitrarily combine them.

The conventional invocation variant uses the function name followed by parenthesis `( )`. Within the parenthesis, there are the function's arguments, if any exist.

```
"use strict";

function duplication(p) {
  return p + "! " + p + "!";
}

// the conventional invocation method
const ret = duplication("Go");
alert(ret);
```

If the script runs in a browser, there are two more possibilities. They use the `window` object that is provided by the browser.

```
"use strict";

function duplication(p) {
  return p + "! " + p + "!";
}

// via 'call'
let ret = duplication.call(window, "Go");
alert(ret);

// via 'apply'
ret = duplication.apply(window, ["Go"]);
alert(ret);
```

Hint: If you use the function name without the parenthesis `()`, you will receive the function itself (the script), not any result of an invocation.

```
"use strict";

function duplication(p) {
  return p + "! " + p + "!";
}

alert(duplication); // 'function duplication (p) { ... }'
```

Hoisting

Functions are subject to 'hoisting'. This mechanism transfers the declaration of a function automatically to the top of its scope. As a consequence, you can call a function from an upper place in the source code than its declaration.

```
"use strict";

// use a function above (in source code) its declaration
const ret = duplication("Go");
alert(ret);

function duplication(p) {
  return p + "! " + p + "!";
}
```

Immediately Invoked Function

So far, we have seen the two separate steps *declaration* and *invocation*. There is also a syntax variant that allows the combination of both. It is characterized by using parenthesis around the function declaration followed by ( ) to invoke that declaration.

```
"use strict";

alert(  // 'alert' to show the result
  // declaration plus invocation
  (function (p) {
    return p + "! " + p + "!";
  })("Go")   // ("Go"): invocation with the argument "Go"
);

alert(
  // the same with 'arrow' syntax
  ((p) => {
    return p + "! " + p + "!";
  })("Gooo")
);
```

This syntax is known as a Immediately Invoked Function Expression (IIEF) (https://developer.mozilla.org/en-US/docs/Glossary/IIFE).

Arguments

When functions are called, the parameters from the declaration phase are replaced by the arguments of the call. In the above declarations, we used the variable name p as a parameter name. When calling the function, we mostly used the literal "Go" as the argument. At runtime, it replaces all occurrences of p in the function. The above examples demonstarte this technique.

Call-by-value

Such substitutions are done 'by value' (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions#passing_arguments) and not 'by reference'. A **copy** of the argument's original value is passed to the function. If this copied value is changed within the function, the original value outside of the function is not changed.

```
"use strict";

// there is one parameter 'p'
function duplication(p) {
  // In this example, we change the parameter's value
  p = "NoGo";
  alert("In function: " + p);
  return p + "! " + p + "!";
```

```
};

let x = "Go";
const ret = duplication(x);

// is the modification of the argument done in the function visible here? No.
alert("Return value: " + ret + " Variable: " + x);
```

For objects (all non-primitive data types), this 'call-by-value' has a - possibly - astonishing effect. If the function modifies a property of the object, this change is also seen outside.

```
"use strict";

function duplication(p) {
  p.a = 2;       // change the property's value
  p.b = 'xyz';   // add a property
  alert("In function: " + JSON.stringify(p));
  return JSON.stringify(p) + "! " + JSON.stringify(p) + "!";
};

let x = {a: 1};
alert("Object: " + JSON.stringify(x));
const ret = duplication(x);

// is the modification of the argument done in the function visible here? Yes.
alert("Return value: " + ret + " Object: " + JSON.stringify(x));
```

When the example runs, it shows that after the invocation of `duplication`, the changes made by the function are visible not only in the return value. Also, the properties of the original object `x` have changed. Why this; is it different from the behavior concerning primitive data types? No.

The function receives a copy of the reference to the object. Hence, within the function, the same object is referenced. The object itself exists only one time, but there are two (identical) references to the object. It does not make a difference whether the object's properties are modified by one reference or the other.

Another consequence is - and this may be intuitively the same as with primitive data types (?) - that the modification of the reference itself, e.g., by the creation of a new object, will not be visible in the outer routine. The reference to the new object is stored in the **copy** of the original reference. Now we have not only two references (with different values) but also two objects.

```
1  "use strict";
2
3  function duplication(p) {
4
5    // modify the reference by creating a new object
6    p = {};
7
8    p.a = 2;       // change the property's value
9    p.b = 'xyz';   // add a property
10   alert("In function: " + JSON.stringify(p));
11   return JSON.stringify(p) + "! " + JSON.stringify(p) + "!";
12 };
13
14 let x = {a: 1};
15 alert("Object: " + JSON.stringify(x));
16 const ret = duplication(x);
17
18 // is the modification of the argument done in the function visible here? No.
19 alert("Return value: " + ret + " Object: " + JSON.stringify(x));
```

Note 1: The naming of this argument-passing technique is not used consistently across different languages. Sometimes it is called 'call-by-sharing'. Wikipedia gives an overview.

Note 2: The described consequences of JavaScript's argument-passing are comparable with the consequences of using the keyword `const`, which declares a variable to be a *constant*. Such variables cannot be changed. Nevertheless, if they refer to an object, the object's properties can be changed.

Default values

If a function is invoked with fewer arguments than it contains parameters, the surplus parameters keep undefined. But you can define default values for this case by assigning a value within the function's signature. The missing parameters will get those as their default values.

```
"use strict";

// two nearly identical functions; only the signature is slightly different
function f1(a, b) {
  alert("The second parameter is: " + b)
};

function f2(a, b = 10) {
  alert("The second parameter is: " + b)
};

// identical invocations; different results
f1(5);        //   undefined
f1(5, 100);   //   100

f2(5);        //   10
f2(5, 100);   //   100
```

Variable number of arguments

For some functions, it is 'normal' that they get involved with different numbers of arguments. For example, think of a function that shows names. `firstName` and `familyName` must be given in any case, but it's also possible that an `academicTitle` or a `titleOfNobility` must be shown. JavaScript offers different possibilities to handle such situations.

Individual checks

The 'normal' parameters, as well as the additional parameters, can be checked to determine whether they contain a value or not.

```
"use strict";

function showName(firstName, familyName, academicTitle, titleOfNobility) {
  "use strict";

  // handle required parameters
  let ret = "";
  if (!firstName || !familyName) {
    return "first name and family name must be specified";
  }
  ret = firstName + ", " + familyName;

  // handle optional parameters
  if (academicTitle) {
    ret = ret + ", " + academicTitle;
  }
  if (titleOfNobility) {
    ret = ret + ", " + titleOfNobility;
  }
```

```
  return ret;
}

alert(showName("Mike", "Spencer", "Ph.D."));
alert(showName("Tom"));
```

Every single parameter that may not be given must be individually checked.

The 'rest' parameter

If the handling of the optional parameters is structurally identical, the code can be simplified by using the *rest operator* syntax - mostly in combination with a loop. The syntax of the feature consists of three dots in the function's signature - like in the *spread syntax*.

How does it work? As part of the function invocation, the JavaScript engine combines the given optional arguments into a single array. (Please note that the calling script does **not** use an array.) This array is given to the function as the last parameter.

```
"use strict";

// the three dots (...) introduces the 'rest syntax'
function showName(firstName, familyName, ...titles) {

  // handle required parameters
  let ret = "";
  if (!firstName || !familyName) {
    return "first name and family name must be specified";
  }
  ret = firstName + ", " + familyName;

  // handle optional parameters
  for (const title of titles) {
    ret = ret + ", " + title;
  }

  return ret;
}

alert(showName("Mike", "Spencer", "Ph.D.", "Duke"));
alert(showName("Tom"));
```

The third and all following arguments of the call are collected into a single array that is available in the function as the last parameter. This allows the use of a loop and simplifies the function's source code.

The 'arguments' keyword

In accordance with other members of the C-family of programming languages, JavaScript offers the keyword `arguments` within functions. It is an Array-like object that contains all given arguments of a function call. You can loop over it or use its `length` property.

Its functionality is comparable with the above *rest syntax*. The main difference is that `arguments` contains **all** arguments, whereas the *rest syntax* affects not necessarily all arguments.

```
"use strict";

function showName(firstName, familyName, academicTitles, titlesOfNobility) {

  // handle ALL parameters with a single keyword
```

```
  for (const arg of arguments) {
    alert(arg);
  }
}

showName("Mike", "Spencer", "Ph.D.", "Duke");
```

Return

The purpose of a function is to provide a solution of a dedicated problem. This solution is given back to the calling program by the `return` statement.

Its syntax is `return <expression>`, where `<expression>` is optional.

A function runs until it reaches such a `return` statement (, or an uncatched exception occurs, or behind the last statement). The `<expression>` may be a simple variable of any data type like `return 5`, or a complex expression like `return myString.length`, or is omitted entirely: `return`.

If there is no `<expression>` within the `return` statement, or if no `return` statement is reached at all, `undefined` is returned.

```
"use strict";

function duplication(p) {
  if (typeof p === 'object') {
    return;  // return value: 'undefined'
  }
  else if (typeof p === 'string') {
    return p + "! " + p + "!";
  }
  // implicit return with 'undefined'
}

let arg = ["Go", 4, {a: 1}];
for (let i = 0; i < arg.length; i++) {
  const ret = duplication(arg[i]);
  alert(ret);
}
```

Arrow functions (=>)

Arrow functions are a compact alternative to the above-shown conventional function syntax. They abbreviate some language elements, omit others, and have only a few semantic distinctions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) in comparison to the original syntax.

They are always anonymous but can be assigned to a variable.

```
"use strict";

// original conventional syntax
function duplication(p) {
    return p + "! " + p + "!";
}

// 1. remove keyword 'function' and function name
// 2. introduce '=>' instead
// 3. remove 'return'; the last value is automatically returned
(p) => {
```

```
      p + "! " + p + "!"
}

// remove { }, if only one statement
(p) =>  p + "! " + p + "!"

// Remove parameter parentheses if it is only one parameter
// ----------------------------
      p => p + "! " + p + "!"     // that's all!
// ----------------------------

alert(
  (p => p + "! " + p + "!")("Go")
);
```

Here is one more example using an array. The `forEach` method loops over the array and produces one array-element after the next. This is put to the arrow function's single argument `e`. The arrow function shows `e` together with a short text.

```
"use strict";

const myArray = ['a', 'b', 'c'];

myArray.forEach(e => alert("The element of the array is: " + e));
```

Other programming languages offer the concept of arrow functions under terms like anonymous functions or lambda expressions.

Recursive Invokations

Functions can call other functions. In real applications, this is often the case.

A particular situation occurs when they call themselves. This is called a *recursive invokation*. Of course, this implies the danger of infinite loops. You must change something in the arguments to avoid this hurdle.

Typically the need for such recursive calls arises when an application handles tree structures like bill of materials, a DOM tree, or genealogy information. Here we present the simple to implement mathematical problem of factorial computation.

The factorial is the product of all positive integers less than or equal to a certain number n, written as $n!$. For example, $4! = 4 \times 3 \times 2 \times 1 = 24$. It can be solved by a loop from 1 to n, but there is also a recursive solution. The factorial of n is the already computed factorial of $(n-1)$ multiplied with n, or in formulas: $n! = (n-1)! \times n$. This thought leads to the correspondent recursive construction of the function:

```
"use strict";

function factorial(n) {
  if (n > 0) {
    const ret = n * factorial(n-1);
    return ret;
  } else {
    // n = 0;  0! is 1
    return 1;
  }
}

const n = 4;
alert(factorial(n));
```

As long as n is greater than 0, the script calls `factorial` again, but time with $n - 1$ as the argument. Therefore the arguments converge to 0. When 0 is reached, this is the first time the `factorial` function is not called again. It returns the value of 1. This number is multiplied by the next higher number from the previous invocation of `factorial`. The result of the multiplication is returned to the previous invocation of `factorial`, ...

Exercises

... are available on another page (click here).

See also

- MDN: Functions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)
- MDN: Rest parameter (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)

Closures

A *Closure* is a technique where a function is bundled (enclosed) with its surrounding variables, the lexical environment. Typically, *Closures* are implemented in functional programming languages like JavaScript where they support Currying.

Lexical environment

First, we show the access of a function to its lexical environment. This alone is **not** a *Closure*.

```
"use strict";

function incrementByCertainValue(param) {
  return param + certainValue;
}

let certainValue = 10;
alert(incrementByCertainValue(8)); // 18

certainValue = 100;
alert(incrementByCertainValue(8)); // 108
```

The function `incrementByCertainValue` increments its parameter by the value of the variable `certainValue`. Because `incrementByCertainValue` as well as `certainValue` are defined within the same block, the function has access to the variable. Whenever the function is called, it reads the variable and uses it to compute its return value.

Closure

To extend the above example to a *Closure*, we have to combine an (inner) function with access to variables of its lexical environment - typically another function -, **and** save this state by returning this inner function.

```
1  "use strict";
2
3  function incrementByCertainValue(param) {
4    // declare a function that will perform the work. (Only declaration, currently not
   invoked.)
5    const add = function (certainValue) {
6      // access to 'param' from lexical environment and to its parameter 'certainValue'
7      return param + certainValue;
8    }
9    return add;
10 }
11
12 const incrementByFive = incrementByCertainValue(5);
13 alert(incrementByFive(8));  // 13
14
15 const incrementBySix = incrementByCertainValue(6);
16 alert(incrementBySix(8));   // 14
17
18 alert(incrementByFive(10)); // 15
```

The (outer) function `incrementByCertainValue`

- contains a parameter `param`,
- defines an (inner) function `add` that takes another parameter `certainValue`,

in addition to its parameter, the (inner) function has access - as usual - to its lexical environment where `param` is defined
returns the (inner) function.

So far, there are exclusively declarations and no running code.

When the variable `incrementByFive` is declared in line 12, it is initialized with the return value of the function `incrementByCertainValue(5)`. **This is the crucial point** where code runs and the *Closure* technique acts. Within `incrementByCertainValue` the 5 is known as the parameter/variable `param`. Next, the function `add` is created using `param` from its lexical environment. This function `add` accepts one parameter that must be given from the calling routine later on. The return statement of `incrementByCertainValue` delivers this function `add` that has bound the value '5' into its **body**. Please note that the function name `add` is arbitrary and not seen outside of `incrementByCertainValue`.

When `incrementByCertainValue` is called a second time with the argument '6', the '6' is bound to a separate, second function.

Exercises

... are available on another page (click here).

See also

- MDN: Closures (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures)
- Wiki: Closures
- Currying in JavaScript (German).

Async

Without particular keywords and techniques, the JavaScript engine executes statements one after the next in the sequence of the written code. In most cases, this is necessary because the result of a line is used in the following line.

```
"use strict";

/* Line 1 */ const firstName = "Mahatma";
/* Line 2 */ const familyName = "Gandhi";
/* Line 3 */ const completeName = firstName + " " + familyName;
```

Lines 1 and 2 must be finished entirely before line 3 can be executed. This is the usual sequential behavior.

But there are situations where it is not necessary for the following statements to wait for the end of the current one. Or, you expect that an activity will run for a long time, and you want to do something else in the meanwhile. Such parallel execution has the potential to reduce the overall response time dramatically. That becomes possible because modern computers have multiple CPUs and are able to perform multiple tasks at the same time. In the above example, lines 1 and 2 may run in parallel. Moreover, the client/server architecture delegates activities across multiple servers.

Typical situations are long-running database updates, handling of huge files, or CPU-intensive computations. But even for simple-looking things like the rendering of an HTML page, a browser typically runs multiple tasks.

Single-threaded

Natively, "JavaScript is single-threaded and all the JavaScript code executes in a single thread. This includes your program source code and third-party libraries that you include in your program. When a program makes an I/O operation to read a file or a network request, this blocks the main thread"[1].

To reach the goal of simultaneous activities anyway, browsers, service workers, libraries, and servers offer additional appropriate interfaces. Their primary use case is the unblocking execution of HTTP and database requests; a smaller proportion focuses on CPU-intensive computations.

At the level of the JavaScript language, there are three techniques to achieve asynchronism - what 'feels' like simultaneity.

- Callback function
- Promise
- Keywords `async` and `await`

The basic technique is the use of *callback functions*. This term is used for functions that are passed as an argument to another function, especially - but not only - for functions that implement an asynchronous behavior.

A *Promise* represents the final completion or failure of an asynchronous operation, including its result. It steers further processing after the end of such asynchronous running operations.

Because the evaluation of *Promises* with `.then` and `.catch` may lead to hard-to-read code - especially if they are nested -, JavaScript offers the keywords `async` and `await`. Their usage generates well-arranged code comparable with `try .. catch .. finally` of traditional JavaScript. But they don't implement additional features. Instead, under the hood they are based on *Promises*.

Strictly sequential? No.

To demonstrate that code is not always executed in strict sequential order, we use a script that contains a CPU-intensive computation within a more or less huge loop. Depending on your computer, it runs for several seconds.

```
"use strict";

// function with CPU-intensive computation
async function func_async(upper) {
  await null;                                          // (1)
  return new Promise(function(resolve, reject) {       // (2)
    console.log("Starting loop with upper limit: " + upper);
    if (upper < 0) {
      // an arbitrary test to generate a failure
      reject(upper + " is negative. Abort.");
    }
    for (let i = 0; i < upper; i++) {
       // an arbitrary math function for test purpose
       const s = Math.sin(i);                          // (3)
    }
    console.log("Finished loop for: " + upper);
    resolve("Computed: " + upper);
  })
}

const doTask = function(arr) {
  for (let i = 0; i < arr.length; i++) {
    console.log("Function invocation with number: " + arr[i]);
    func_async(array1[i])                              // (4)
      .then((msg) => console.log("Ok. " + msg))
      .catch((msg) => console.log("Error. " + msg));
    console.log("Behind invocation for number: " + arr[i]);
  }
}

const array1 = [3234567890, 10, -30];
doTask(array1);
console.log("End of program. Really?");
```

Expected output:

```
Function invocation with number: 3234567890
Behind invocation for number: 3234567890
Function invocation with number: 10
Behind invocation for number: 10
Function invocation with number: -30
Behind invocation for number: -30
End of program. Really?
Starting loop with upper limit: 3234567890
Finished loop for: 3234567890
Starting loop with upper limit: 10
Finished loop for: 10
Starting loop with upper limit: -30
Finished loop for: -30
Ok. Computed: 3234567890
Ok. Computed: 10
Error. -30 is negative. Abort.
```

- The core of the asynchronous function `func_async` is a loop where a mathematical computation is done [(3) line 14]. The loop needs more or less time depending on the given parameter.
- The return value of `func_async` is **not** a simple value but a *Promise*. [(2) line 6]
- `func_async` is invoked by `doTask` once per element of the given array. [(4) line 24]
- Because of the asynchronous nature of `asyn_func` the function `doTask` is executed **totally** before `func_async` runs! This can be observed by the program output.
- The `await null` [(1) line 5] is a dummy call. It suspends the execution of `func_async`, giving `doTask` the chance to continue. If you delete this statement, the output will be different. Conclusion: To make the function **really asynchronous** you need both keywords, `async` in the function signature and `await` in the function body.
- If you have a tool to observe your computer in detail, you can recognize that the three invocations of `func_async` doesn't run at the same time on different CPUs but run one after the next (on the same or on different CPUs).

Callback

Passing a function as a parameter to an (asynchronous) function is the original technique in JavaScript. We demonstrate its purpose and advantage with the predefined `setTimeout` function. It takes two parameters. The first one is the callback function we are speaking about. The second is a number specifying the milliseconds after which the callback function is called.

```
"use strict";

function showMessageLater() {
  setTimeout(showMessage, 3000);  // in ms
}

function showMessage() {
  alert("Good morning.");
}

showMessage();       // immediate invocation
showMessageLater();  // invocation of 'showMessage' after 3 seconds
```

If `showMessage` is invocated, it runs instantly. If `showMessageLater` is invocated, it passes `showMessage` as a callback function to `setTimeout`, which executes it after a delay of 3 seconds.

Promise

A *Promise* keeps track of whether an (asynchronous) function has been executed successfully or has terminated with an error, and it determines what happens next, invoking either `.then` or `.catch`.

The *Promise* is in one of three states:

- Pending: Initial state, before 'resolved' or 'rejected'
- Resolved: After successful completion of the function: `resolve` was called
- Rejected: After completion of the function with an error: `reject` was called

```
"use strict";

// here, we have only the definition!
function demoPromise() {
```

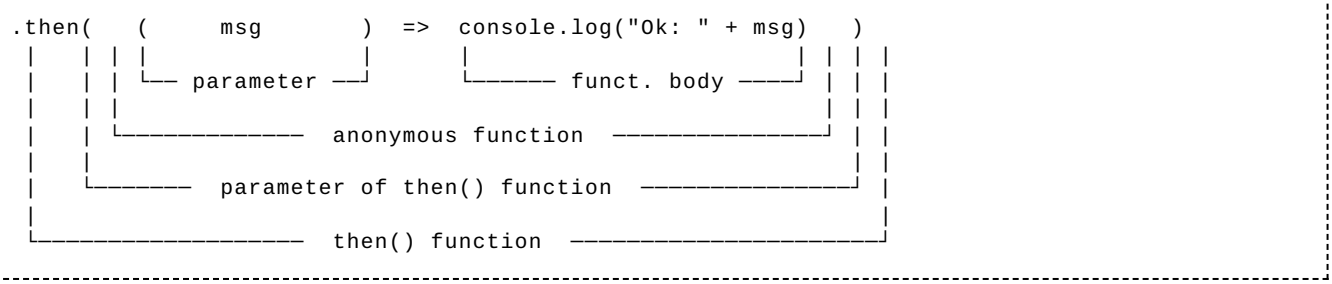
```
  // The 'return' statement is executed immediately, and the calling program
  // continues its execution. But the value of 'return' keeps undefined until
  // either 'resolve' or 'reject' is executed here.
  return new Promise(function(resolve, reject) {
    //
    // perform some time-consuming actions like an
    // access to a database
    //
    const result = true;  // for example

    if (result === true) {
      resolve("Demo worked.");
    } else {
      reject("Demo failed.");
    }
  })
}

demoPromise()  // invoke 'demoPromise'
  .then((msg) => console.log("Ok: " + msg))
  .catch((msg) => console.log("Error: " + msg));

console.log("End of script reached. But the program is still running.");
```

Expected output:

```
End of script reached. But the program is still running.
Ok: Demo worked.
```

When `demoPromise` is invoked, it creates a new *Promise* and returns it. Then, it performs the time-consuming action, and depending on the result, it invokes `resolve` or `reject`.

Next (in the meanwhile, the calling script has done other things), either the `.then()` or the `.catch()` functions behind the call to `demoPromise` are executed. The two functions accept an (anonymous) function as their parameter. The parameter given to the anonymous function is the value of the *Promise*.



Please notice that the last statement of the script has been executed **previously**.

Many interfaces to libraries, APIs, and server functions are defined and implemented as functions returning a *Promise*, similar to the above `demoPromise`. As long as it's not necessary that you create your own asynchronous functions, it's not necessary that you create *Promises*. Often it's sufficient to call an external interface and work only with `.then` or `.catch` (or the `async` or `await` of next chapter).

async / await

The keyword `await` forces the JavaScript engine to run the script named behind `await` entirely - including the `new Promise` part - before executing the following statement. Hence, - from the standpoint of the calling script - the asynchronous behavior is removed. Functions with `await` must be

flagged in their signature with the keyword `async`.

```
"use strict";

// same as above
function demoPromise() {
  return new Promise(function(resolve, reject) {
    const result = true;  // for example
    if (result === true) {
      resolve("Demo worked.");
    } else {
      reject("Demo failed.");
    }
  })
}

// a function with the call to 'demoPromise'
// the keyword 'async' is necessary to allow 'await' inside
async function start() {
  try {
    // use 'await' to wait for the end of 'demoPromise'
    // before executing the following statement
    const msg =  await demoPromise();
    // without 'await', 'msg' contains the Promise, but without
    // the success- or error-message
    console.log("Ok: " + msg);
  } catch (msg) {
    console.log("Error: " + msg);
  }
}

start();
console.log("End of script reached. End of program?");
```

The use of `async .. await` allows you to work with the traditional `try .. catch` statement instead of `.then` and `.catch`.

A realistic example

We use the freely available demo API `https://jsonplaceholder.typicode.com/`. It offers a small amount of test data in JSON format.

```
"use strict";

async function getUserData() {

  // fetch() is an asynchronous function of the Web API. It returns a Promise
  // see: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
  await fetch('https://jsonplaceholder.typicode.com/users')

    // for the use of 'response', see: https://developer.mozilla.org/en-
US/docs/Web/API/Response/json
    // '.json()' reads the response stream
    .then((response) => { return response.json() })

    // in this case, 'users' is an array of objects (JSON format)
    .then((users) => {
      console.log(users); // total data: array with 10 elements

      // loop over the ten array elements
      for (const user of users) {
        console.log(user.name + " / " + user.email);
      }
    })
    .catch((err) => console.log('Some error occurred: ' + err.message));
}
```

```
// same with 'try / catch'
async function getUserData_tc() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/users');
    const users = await response.json();
    console.log(users);
    console.log(users[0].name);
    for (const user of users) {
      console.log(user.name + " / " + user.email);
    }
  } catch (err) {
    console.log('Some error occurred: ' + err.message);
  }
}

getUserData();
getUserData_tc();
```

The steps of the example are:

- `await fetch()`: Get the data behind the URL. The `await` part guarantees that the script will not continue before the `fetch` has delivered all data.
- `json()` reads the stream which contains the resulting data.
- The resulting data is an array of 10 elements. Each element is in JSON format, e.g.:


```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```


- As an example, the script shows the complete data and some of its parts in the console.

Note: It's likely that you will run into a CORS error (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) when you use an arbitrary URL.

Exercises

... are available on another page (click here).

See also

- Processes and Threads (in Node.js) (https://www.digitalocean.com/community/tutorials/how-to-use-multithreading-in-node-js#understanding-processes-and-threads)
- Promises and 'async' (in Node.js) (https://www.geeksforgeeks.org/difference-between-promise-and-async-await-in-node-js/)
- MDN: async (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function)
- MDN: Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

References

1. *Hiden threads in Node.js (https://www.digitalocean.com/community/tutorials/how-to-use-multithreading-in-node-js#understanding-hidden-threads-in-node-js)*

Object-based programming

Object-oriented programming (OOP) is a software design paradigm that first arose in the 1960s and gained popularity in the 1990s. It strives for modularization, reusability, encapsulation and hiding of state (data) and behavior (functions), design in a hierarchy of generalization, inheritance, and more.

It allows the components to be as modular as possible. In particular, when a new object type is created, it is expected that it should work without problems when placed in a different environment or new programming project. The benefits of this approach are a shorter development time and easier debugging because you're re-using program code that has already been proven. This 'black box' approach means that data goes into the object and other data comes out of the object, but what goes on inside isn't something you need to concern yourself with.

Over time different techniques have been developed to implement OOP. The most popular ones are the class-based and the prototype-based approach.

Class-based OOP

Classes are a blueprint that defines all aspects - state as well as behavior - of a group of structurally identical objects. The blueprint is called the *class*, and the objects *instances* of that class. Popular members of the C-family languages, especially Java, C++, and C#, implement OOP with this class-based approach.

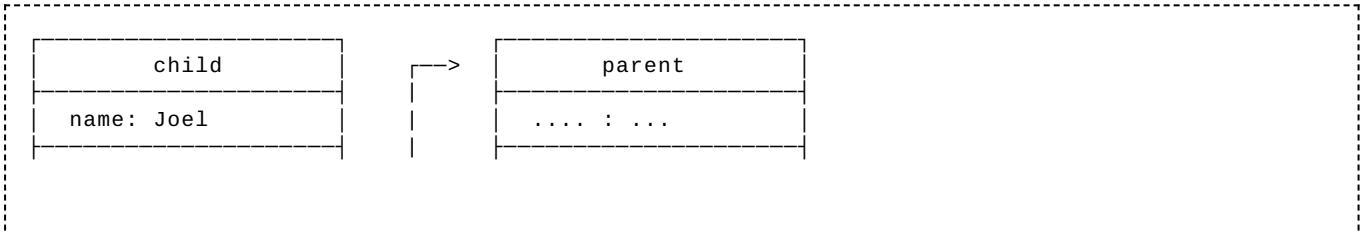
Prototype-based OOP

In the prototype-based approach, every object stores its state and behavior. In addition, it has a *prototype* (or `null` if it is on top of the hierarchy). Such a *prototype* is a pointer to another, more general object. All properties of the referenced object are also available in the referencing object. Classes don't exist in the prototype-based approach.

OOP in JavaScript "Two jackets for one body"

One of JavaScript's cornerstones is the provision of objects in accordance with the rules of the prototype-based OOP. Objects consist of properties that are key/value pairs holding data as well as methods. One of those properties is always the *prototype* property '__proto__'. It points to the 'parent' object and, by this, realizes the relationship.

```
// relationships are realized by the property '__proto__'
let parent = [];
let child = {
  name:      "Joel",
  __proto__: parent,
};
console.log(Object.getPrototypeOf(child)); // Array []
```



```
| __proto__: parent | ---┘     | __proto__: ...     | ---> ... ---> null
└-------------------┘          └--------------------┘
```

If a requested property is missed on any object, the JavaScript engine searches it in the 'parent' object, 'grandparent' object, and so on. This is called the *prototype chain*.

All of that applies to user-defined objects and system-defined objects like `Arrays` or `Date` in the same way.

Since EcmaScript 2015 (ES6), the syntax offers keywords like `class` or `extends`, which are used in class-based approaches. Even though such keywords have been introduced, the fundaments of JavaScript haven't changed: Those keywords lead to prototypes in the same way as before. They are syntactical sugar and get compiled to the conventional prototype technique.

In summary, the syntax of JavaScript offers two ways to express object-oriented features like inheritance in the source code: the 'classical' and the 'class' style. Despite the different syntax, the implementation techniques differ only slightly.

The classical syntax

Since its first days, JavaScript has defined the parent/child relation of objects with the 'prototype' mechanism. If not explicitly notated in the source code, this happens automatically. The classical syntax exposes it quite well.

To define a parent/child relation of two objects explicitly, you should use the method `setPrototypeOf` to set the prototype of an object to a dedicated other object. After the method has run, all properties - inclusive functions - of the parent object are known to the child.

```
<!DOCTYPE html>
<html>
<head>
  <script>
    function go() {
      "use strict";

      const adult = {
        familyName: "McAlister",
        showFamily: function() {return "The family name is: " + this.familyName;}
      };
      const child = {
        firstName: "Joel",
        kindergarten: "House of Dwars"
      };

      // 'familyName' and 'showFamily()' are undefined here!
      alert(child.firstName + " " + child.familyName);

      // set the intended prototype chain
      Object.setPrototypeOf(child, adult);
      // or:  child.__proto__ = adult;

      alert(child.firstName + " " + child.familyName);
      alert(child.showFamily());
    }
    </script>
  </head>

  <body id="body">
    <button onclick="go()">Run the demo</button>
```

```
  </body>
</html>
```

The 'adult' object contains the 'familyName' and a function 'showFamily'. In the first step, they are not known in the object 'child'. After `setPrototypeOf` was running, they are known because the 'child's prototype no longer points to the default 'Object' but to 'adult'.

The next script demonstrates the prototype chain. It starts with the user-defined variables `myArray` and `theObject`. `myArray` is an array with three elements. The assignment operation in line 6 sets `theObject` to the same array. A loop shows the prototype of `theObject` and assigns the next higher level of the prototype chain to it. The loop finishes when the top level of the hierarchy is reached. In this case, the prototype is `null`.

```
1   function go() {
2     "use strict";
3
4     // define an array with three elements
5     const myArray = [0, 1, 2];
6     let theObject = myArray;
7
8     do {
9       // show the object's prototype
10      console.log(Object.getPrototypeOf(theObject));  // Array[], Object{...}, null
11      // or: console.log(theObject.__proto__);
12
13      // switch to the next higher level
14      theObject = Object.getPrototypeOf(theObject);
15    } while (theObject);
16  }
```

As you know, *properties* are key/value pairs. Hence it is possible to directly use the value of the 'prototype' property to identify and manipulate prototypes. Interestingly the key's name isn't 'prototype' but '__proto__'. This is shown in line 11. Nevertheless, we recommend ignoring this technique and using API methods for prototype manipulations, such as `Object.getPrototypeOf`, `Object.setPrototypeOf`, and `Object.create` instead.

The 'class' syntax

The script defines two classes, *Adult* and *Child* with some internal properties, one of them being a method. The keyword `extends` combines the two classes hierarchically. Afterward, in line 21, an instance is created with the keyword `new`.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <script>
5    function go() {
6      "use strict";
7
8      class Adult {
9        constructor(familyName) {
10         this.familyName = familyName;
11       }
12       showFamily() {return "The family name is: " + this.familyName;}
13     }
14     class Child extends Adult {
15       constructor(firstName, familyName, kindergarten) {
16         super(familyName);
17         this.firstName = firstName;
```

```
        this.kindergarten = kindergarten;
      }
    }

    const joel = new Child("Joel", "McAlister", "House of Dwars");
    alert(joel.firstName + " " + joel.familyName);
    alert(joel.showFamily());
  }
  </script>
</head>

<body id="body">
  <button onclick="go()">Run the demo</button>
</body>
</html>
```

The property `familyName` and the method `showFamily` are defined in the *Adult* class. But they are also known in the *Child* class.

Please note again that this class-based inheritance in JavaScript is implemented on top of the prototype-based classical approach.

See also

- MDN: OOP in JavaScript (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming)
- More Details (http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/)

OOP-classical

The heart of JavaScript's object-based approach is the linked list of objects where every object acts as a prototype for its successor. It is revealed when using the classical syntax.

Construction

There are different syntactical ways to construct objects. They are not identical, but even looking behind the scene, you will see only slight differences in their semantics. All variants create a set of key/value pairs, the properties. This set of properties composes an object.

```
"use strict";

// construction via literals
const obj_1 = {};
obj_1.property_1 = "1";
alert(obj_1.property_1);

const obj_1a = {property_1a: "1a"};
alert(obj_1a.property_1a);


// construction via 'new' operator
const obj_2 = new Object();
obj_2.property_2 = "2";
alert(obj_2.property_2);

const obj_2a = new Object({property_2a: "2a"});
alert(obj_2a.property_2a);


// construction via 'Object.create' method
const obj_3 = Object.create({});
obj_3.property_3 = "3";
alert(obj_3.property_3);

const obj_3a = Object.create({property_3a: "3a"});
alert(obj_3a.property_3a);
```

The three language constructs *literal*, `new`, and `Object.create` create simple or complex objects. Such objects can be subsequently extended by assigning values to additional properties.

Functions

The 'value' part of the key/value pairs can contain not only values of primitive data types. It's also possible that they contain functions. (When functions are the value-part of a property, they are called a *method*.)

```
"use strict";

// pure literal syntax for 'func_1'
const obj_1 = {
  property_1: "1",
  func_1: function () {return "Message from func_1: " + this.property_1}
};
// add a second function 'func_2'
obj_1.property_2 = "2";
```

```
obj_1.func_2 = function () {return "Message from func_2: " + this.property_2};

// invoke the two functions
alert(obj_1.func_1());
alert(obj_1.func_2());
```

new

In the previous example, we defined objects containing a property with a value and another with a method. Both parts are accessible by the usual dot-notation. But they miss a smart syntactical feature: it's not possible to define their properties directly with the first invocation. Something like `const x = new obj_1("valueOfProperty")` or `const mike = new Person("Mike")` will not run because such a syntax misses the name of the property.

We change and extend the above example to allow this syntax, the `new` operator in combination with parameters. To do so, we define functions (which are also objects) that contain and store variables as well as ('inner') functions/methods.

```
"use strict";

function Person(name, isAlive) {
  this.name = name;
  this.isAlive = isAlive;
  // a (sub-)function to realize some functionality
  this.show = function () {return "The person's name is: " + this.name};
}

// creation via 'new'
const mike = new Person("Mike", true);
const john = new Person("John", false);

alert(mike.name + " / " + mike.show());
alert(john.name + " / " + john.show());
```

The function `Person` takes parameters as every other function. The first letter of its name is written in uppercase, but this is only a convention and not mandatory. If the function is invoked with the `new` operator, in the first step, a new object is constructed (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new). Within the function, you can refer to the new object with the keyword 'this', e.g., to store the given parameters. If, in addition, the function shall offer some functionality in the form of ('inner') functions, you define them and store a reference to them in 'this' under an arbitrary name - in the example, it is the 'show' function.

After the function is defined, you can use them via the `new` operator to create individual objects (instances). Such individual objects store the given arguments and offer the internally defined functions.

Please note that the `new Person(...)` statement is different from the usual invocation of a function without the `new` operator: `Person(...)`. The `new` is necessary to indicate that the construction of an object must be done before the body of the function can run. Without `new`, the JavaScript engine doesn't create an object, and the use of 'this' will fail.

Predefined data types

Many predefined data types (Date, Array, ...) are defined in the above way. Therefore you can use the `new` operator for their creation: `const arr = new Array([0, 1, 2])`. The arguments are stored somewhere in the newly created object. Here, it's only a single one, a literally expressed array. It gets decomposed, and the array elements are stored. Some derived properties are computed, e.g., the `length`, and a lot of methods are provided, e.g., `push` and `pop`. All in all, the internal structure often differs from the externally visible form.

In addition to this unified syntax with `new` there are some syntax variants that are special according to the intended data type. E.g., for `Array` you can use `const arr = [0, 1, 2]`. But it's just an abbreviation for: `const arr = new Array([0, 1, 2])`.

Inheritance

This chapter shows different syntactical possibilities for how to arrange objects in a hierarchy.

setPrototypeOf

If you have defined independent objects, you can subsequently link them together so that they build a parent/child relationship afterward. The crucial function is `setPrototypeOf`. As its name suggests, the function sets one object as the prototype of another object. By this, the parent's properties, including functions, are accessible to the child.

```
1  "use strict";
2
3  // two objects which are independent of each other (in the beginning)
4  const parent = {property_1: "1"};
5  const child  = {property_2: "2"};
6
7  // alert(child.property_1);  // undefined in the beginning
8
9  // link them together
10 Object.setPrototypeOf(child, parent);
11
12 alert(child.property_1);    // '1'
```

After `setPrototypeOf` is successfully executed, the *child* object 'extends' the *parent* object. It gets access to all properties of the *parent* object, as shown in line 12.

How does it work? Every single object contains a property named '__proto__', even if it is not mentioned anywhere in the source code. Its value refers to the object which acts as its 'parent'. Also, the 'parent' contains such a '__proto__' property, and so on. At the highest level, the value is `null` to flag the end of the hierarchy. All in all, it's a 'linked list' of objects. It is called the *prototype chain*. It is the heart of JavaScript's implementation of OOP: 'parents' act as 'prototypes' for the referencing objects - for all system objects as well as for all user-defined objects.

The JavaScript engine uses the *prototype chain* whenever it searches for any property. When the engine doesn't find it, it switches to the next higher level and repeats the search.

This applies in the same way to the case that a function is searched.

```
1  "use strict";
2
3  const parent = {
```

```
4    property_1: "1",
5    func_1: function () {return "Message from func_1: " + this.property_1}
6  };
7  const child = {
8    property_2: "2",
9    func_2: function () {return "Message from func_2: " + this.property_2}
10 };
11
12 // alert(child.func_1());  // not possible at the beginning
13 Object.setPrototypeOf(child, parent);
14
15 alert(child.func_1());  // '1'
```

After line 13, the method `func_1` can be invoked by the *child* object, although it is defined by the *parent*.

new

Suppose you know in advance that one object shall act as a child of another object. In that case, the `new` operator offers the possibility to define the dependency from the beginning. The already existing object can be given as a parameter to the creation process. The JavaScript engine will combine this existing object with the newly creating object by the exact same mechanism, the '__proto__' property.

```
"use strict";

const parent = {property_1: "1"}

// inheritance via 'new' operator
const child = new Object(parent);
alert(child.property_1);
```

Object.create

This pre-known hierarchical relation can also be realized with the `Object.create` method.

```
"use strict";

const parent = {property_1: "1"}

// construction via 'Object.create'
const child = Object.create(parent);
alert(child.property_1);
```

A distinction to class-based approaches

There are some distinctions between JavaScript's prototype-based approach and class-based approaches. One of them regarding inheritance is shown here.

After creating a prototype hierarchy and instances with one of the above methods, you can modify the 'parent' instance to manipulate all 'child' instances at once.

```
1  "use strict";
2
3  // construction of a small hierarchy
4  const parent   = {property_1: "1"}
5  const child_11 = {property_11: "11"}
6  const child_12 = {property_12: "12"}
```

```
 7
 8  Object.setPrototypeOf(child_11, parent);
 9  Object.setPrototypeOf(child_12, parent);
10
11  // show that none of the instances contains a property 'property_2'
12  alert(parent.property_2);    // undefined
13  alert(child_11.property_2);  // undefined
14  alert(child_12.property_2);  // undefined
15
16  // a single statement adds 'property_2' to all three instances:
17  parent.property_2 = "2";
18  alert(parent.property_2);    // 2
19  alert(child_11.property_2);  // 2
20  alert(child_12.property_2);  // 2
```

The statement in line 17 adds the property 'property_2' - virtually - to all instances at once. Whenever 'property_2' is acquired by a subsequent statement, the JavaScript engine will follow the prototype chain. First, in the 'child' instances, it will not find 'property_2'. But following the prototype chain, it will find it in the 'parent' instance. For the 'child' instances, it doesn't make a difference whether the property is in its own space or in its parent space.

The distinction to a class-based approach is that not only the value of the new property is added. Also, the structure of all instances is expanded: the added property hasn't existed at all before line 17.

Check object hierarchy

There are different ways to check the hierarchy of data types of any variable or value.

getPrototypeOf

The `getPrototypeOf` method gives you a chance to inspect the hierarchy. It returns the parent object itself, not its data type. If you are interested in the data type of the parent, you must check the parent's data type with one of the other operators.

```
 1  "use strict";
 2
 3  const parent = {property_1: "1"}
 4  const child_1 = Object.create(parent);
 5
 6  // use 'console.log'; it's more meaningful than 'alert'
 7  console.log(Object.getPrototypeOf(child_1));  // {property_1: "1"}
 8
 9  const arr = [0, 1, 2];
10  const child_2 = Object.create(arr);
11  console.log(Object.getPrototypeOf(child_2)); // [0, 1, 2]
12  console.log(Object.getPrototypeOf(arr));     // []
```

Or, follow the prototype chain in a flexible loop:

```
 1  "use strict";
 2
 3  // define an array with three elements
 4  const myArray = [0, 1, 2];
 5  let theObject = myArray;
 6
 7  do {
 8    // show the object's prototype
 9    console.log(Object.getPrototypeOf(theObject));  // Array[], Object{...}, null
10
11    // switch to the next higher level
```

```
  theObject = Object.getPrototypeOf(theObject);
} while (theObject);
```

instanceof

The `instanceof (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof)` operator tests whether the prototype chain of a variable contains the given data type. It returns a boolean value.

```
"use strict";

// define an array with three elements
const myArray = [0, 1, 2];
alert (myArray instanceof Array);   // true
alert (myArray instanceof Object);  // true
alert (myArray instanceof Number);  // false
```

typeof

The `typeof (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof)` operator returns a string showing the data type of its operand. But it is limited to detect only certain data types respectively their parent object. Possible return values are: "undefined", "object", "boolean", "number", "bigint", "string", "symbol", "function".

```
"use strict";

// define an array with three elements
const myArray = [0, 1, 2];
let theObject = myArray;

do {
  // show the object's prototype
  console.log(typeof theObject);  // object, object, object

  // switch to the next higher level
  theObject = Object.getPrototypeOf(theObject);
} while (theObject);
```

Exercises

... are available on another page (click here).

See also

- MDN: Prototypes (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain)

OOP-classes

The popularity of class-based programming languages inspires the JavaScript community to cover its prototype-based implementation of OOP with a syntax that mimics a class-based approach. EcmaScript 2015 (ES6) gets expanded by corresponding key words like 'class' or 'extend'.

Classes are templates or 'blueprints' to create objects. They encapsulate their data and contain functions to work on it.

Creation

```
1  class Person {
2    // Class body is always implicitly in "use strict" mode
3    constructor(name) {
4      // Data. Declarations like 'let x = 0' are not necessary.
5      this.name = name;
6    }
7    // functionality
8    showName() {
9      return "My name is: " + this.name;
10   }
11 }
12
13 const ada = new Person("Lovelace");
14 alert(ada.showName());
```

The keyword `class` introduces the class definition. In the example, *Person* is the class name. It is followed by the class *body* that is enclosed in curly brackets `{ }`, lines 1 - 11. Within the body, there is a special method `constructor`. This function is invoked during class creation. In the example, it takes one argument, the name of a person. Within `constructor` this argument is stored internally by using the keyword 'this'. The class offers only one functionality: the `showName` method.

Static properties and methods

The above syntax shows how to handle properties and methods of individual objects (instances - like 'ada' in the above example). It's also possible to define properties and methods that are not available at the level of individual objects but at the class level - 'Person' in the above example. They are introduced by the keyword `static`.

```
1  class Person {
2    constructor(name) {
3      // data
4      this.name = name;
5    }
6
7    static className = "The PERSON Class";
8    static showClassName() {return "The name of this class is: " + this.className};
9
10   showName() {
11     return "My name is: " + this.name;
12   }
13 }
14
15 const ada = new Person("Lovelace");
```

```
// alert(ada.showClassName()); // Error!
alert(Person.showClassName());
```

Lines 7 and 8 use the 'static' keyword. Therefore the property and method are NOT available for instances, only for the class altogether.

get

Class methods can be offered as properties. This frees the programmer to distinguish between access to methods - via parenthesis `()` - and properties. The keyword `get` introduces the feature.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  // getter
  get showTheName() {
    return this.showName();
  }
  // 'regular' method
  showName() {
    return "My name is: " + this.name;
  }
}

const ada = new Person("Lovelace");
// NO parenthesis ()
alert(ada.showTheName);
```

Inheritance

Next, we define a hierarchy of classes. This is done with the keyword `extends`. In the example, `Employee` is a sub-class of `Person` and has access to all its properties and methods.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  // method
  showName() {
    return "My name is: " + this.name;
  }
}
class Employee extends Person {
  constructor(name, company) {
    super(name);
    this.company = company;
  }
  // method
  showCompany() {
    return "I, " + this.name + ", work at the company " + this.company;
  }
}

const henry = new Employee("Henry Miller", "ACME Inc.");
alert(henry.showCompany());
alert(henry.showName());     // method of the parent class
```

Line 12 invokes the constructor of the parent class. This is necessary because the parent's `constructor` creates 'this'.

Access control

By default, class properties and methods are accessible. You can hide them by using a hash # as the first character of their name.

```
class Person {

  // two hidden properties (sometimes called 'private fields')
  #firstName;
  #lastName;

  constructor(firstName, lastName) {
    this.#firstName = firstName;
    this.#lastName = lastName;
    // one public property
    this.name = lastName + ", " + firstName;
  }
  #showName() {  // hidden method
    alert("My name is " + this.name);
  }
}

const ada = new Person("Ada", "Lovelace");
alert(ada.name);        // ok
alert(ada.firstName);   // undefined
alert(ada.#firstName);  // undeclared private field

alert(ada.#showName()); // undeclared private method
```

Polymorphism

If a method name is used in a 'parent' class as well as in a 'child' class, the JavaScript engine invokes that one of the correlating class.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  // method name is used also in 'child'
  showName() {
    return "My name is: " + this.name;
  }
}
class Employee extends Person {
  constructor(name, company) {
    super(name);
    this.company = company;
  }
  // same method name as in 'parent'
  showName() {
    return "My name is: " + this.name + ". I'm working at the company " + this.company;
  }
}

const henry = new Employee("Henry Miller", "ACME Inc.");
alert(henry.showName());       // from Employee

const nextPerson = new Person("John");
alert(nextPerson.showName());  // from Person
```

The example defines and uses two different methods `showName`.

this

`this` is not a variable or an object; it's a keyword. Depending on the context, it refers to different things. In the context of class definitions, it refers to the class itself, e.g., `this.city = "Nairobi"` refers to the property 'city' of the current class.

When `this` is used at the top level of a file (in other words: outside of any function or object), it refers to the *global object*. In a function, in strict mode, this is `undefined`. In a DOM event, it refers to the element that received the event.

Exercises

> ... are available on another page (click here).

See also

- MDN: Classes (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)
- W3Schools: OOP in JavaScript (https://www.w3schools.com/js/js_class_intro.asp)

Modules

In the early days of JavaScript, scripts were relatively small. In many cases, the complete functionality resides in a single JavaScript file. Over time requirements and solutions grow significantly. Mainly, often-used functionalities are shifted into separate files. With this growth of complexity, the danger of unwanted side effects grows as well, and the need for **modularisation** of source code gets obvious.

No modules

The original JavaScript syntax - which is valid until today - does not know borders between source code written in different files of scripts. Everything is known everywhere, regardless of the file organization. The following example shows that the two functions are known from within HTML and from each other. One can call the other.

```
<!DOCTYPE html>
<html>
<head>
  <script>
  alert("In script 1");
  function function_1 () {
    "use strict";
    alert("In function_1");
    function_2();
  }
  </script>
  <script>
  alert("In script 2");
  function function_2 () {
    "use strict";
    alert("In function_2");
  }
  </script>
</head>
<body>
  <button onclick="function_1()">Click</button>
</body>
</html>
```

When the HTML page is loaded, both `script` parts are read by the browser; hence two `alert` messages are shown. After clicking the button, `function_1` is called, which invokes `function_2`.

The same behavior occurs when you transfer the scripts to external files and refer them via `<script src="./function_1.js"></script>`

To avoid the possibility of **unwanted** side effects from one function to another or from one file to another, diverse forms of modularisations have been developed.

ECMAScript modules (*ES modules*)

Since ECMAScript 2015 (ES6), the standard defines a syntax for modules and their behavior. Most browsers support it natively without any additional library.

Concerning HTML, the syntax changes slightly. The `<script>` element must be extended by the *type* attribute `<script type="module">`. This declares a file or an inline script to be a module. Afterward, its internal classes, functions, variables, ... are no longer visible to other inline scripts, files, or HTML. In the following example, a click on the button results in an error message because the inline script with its function *function_1* is treated as a module.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <script type="module">
5    alert("In script 1");
6    function function_1 () {
7      "use strict";
8      alert("In function_1");
9    }
10   </script>
11 </head>
12 <body>
13   <button onclick="function_1()">Click</button>
14 </body>
15 </html>
```

Due to security reasons, it's complicated to create an example with pure inline scripts. We use an example with an external file. When testing, create this file as shown in the following inline script.

- When the HTML page is loaded, it shows an alert message (line 7).
- The external file *function_1.js* publishes its function *function_1* to the public (line 16). All other functionality keeps hidden (in this simple example, there is no other functionality).
- The function *function_1* gets imported to the inline script (line 5).
- We add the event listener via `addEventListener` to the button (line 10). The event listener consists of an anonymous function that calls `function_1` (in the external file). In line 10, the event listener is only declared; at this moment, it is not called.
- The "use script" statement gets superfluous because modules always act in *strict* mode.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <script type="module">
5    import {function_1} from "./function_1.js";
6
7    alert("In script 1");
8
9    const btn1 = document.getElementById("btn1");
10   btn1.addEventListener("click", () => function_1());
11
12   /* create a file 'function_1.js' with the following content:
13   function function_1() {
14     alert("In function_1");
15   }
16   export { function_1 };
17   */
18
19   </script>
20 </head>
21 <body>
22   <button id="btn1">Click</button>
23 </body>
24 </html>
```

In essence, the *ES6 module syntax (http://w:https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export#syntax)* consists of the two statements `export` and `import`. `export` is used in the publishing module to make some of its classes, functions, or variables publically available. `import` is used in the calling script to get access to those objects.

Nodejs modules (*CommonJS*)

Web servers are not steered by HTML elements. Therefore they use to have a different technique to make the decision about which JS scripts they have to tread as modules and which not.

The popular Web server *node.js* supports the *export/import* syntax of *ES modules*. But its default module system is different. It's called *CommonJS*.

To tell *node.js* which syntax you use, an additional line must be added to the project's `package.json` file.

```
{
  ..
  "type": "module",
  ..
```

leads to the *ES module* syntax. An alternative way is the use of the extension '.mjs' instead of '.js' for file names. A `"type": "commonjs"` line (respective no definition) leads to the node.js specific syntax *CommonJS*.

Within *CommonJS* exports are done with `modules.exports` (please note the additional 's') and imports with the `require` statement. An example:

```
// export in a file 'logger.js'
...
function doLogging() { ... };
module.exports = {doLogging};

// import in a file 'main.js'
const doLogging = require('./logger.js')
...
```

See also

- MDN: Modules (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)
- Node.js modules (https://nodejs.org/api/modules.html)

Generators

The term *generator* denotes a technique to generate a sequence of 'anything': numbers, strings, rows of database queries, elements of an array, nodes of a DOM tree, ...

It's used to avoid any memory overloading, by splitting the data in small chunks.

To use them, you must define a generator function[1] first. Such functions are notated with an asterisk `*` directly behind the keyword `function`, e.g., `function* myGenerator(){...}`.

When called, the function will not run instantly. It only runs just **before** the first occurrence of a `yield` statement and returns a 'Generator object'. This 'Generator object' offers a `next` method. Calling `next` again and again returns the sequence elements one after the next. The elements arise by each `yield` that is reached within the generator function.

Examples

- The script generates a sequence of 4 integers.

```
function* fourInts() {
  let int = 0;
  while (int < 4) {
    yield int;  // each .next() receives the current value of 'int'
    int++;
  }
}

const gen = fourInts();   // creation
alert(gen.next().value);  // 0
alert(gen.next().value);  // 1
alert(gen.next().value);  // 2
alert(gen.next().value);  // 3
alert(gen.next().value);  // undefined
```

- Every `next()` call returns not only a `value`; there is also a boolean `done`. Hence you can call the generator in loops.

```
function* fourInts() {
  let int = 0;
  while (int < 4) {
    yield int;
    int++;
  }
}

const gen = fourInts();   // creation
do {
  const tmp = gen.next();
  if (tmp.done) {
    break;
  } else {
    alert(tmp.value);  // 0, 1, 2, 3
  }
} while (true)
```

- Generation out of an array, a database request, or a tree traversal.

```
function* arrayElements() {
  // for simplicity, we use an array; database queries or tree traversals
  // are more realistic.
  const myArray = ["yellow", "green", "blue"];
  for (const elem of myArray) {
    yield elem;
  }
}

const sequence = arrayElements();   // creation
do {
  const tmp = sequence.next();
  if (tmp.done) {
    break;
  } else {
    alert(tmp.value);  // "yellow", "green", "blue"
  }
} while (true)
```

- Creation of the infinite sequence of all even numbers.

```
function* evenNumbers() {
  for (let i = 0; true; i = i + 2) {
    yield i;
  }
}

const sequence = evenNumbers();   // creation
let i = 0;
while (i < 20) {
  i = sequence.next().value;
  alert(i);  // 0, 2, 4, ...
}
```

Parameters

The generator function may receive parameters. In this example, the 'pageSize' defines the number of array elements to be returned.

```
function* pagination(arr, pageSize) {
  for (let i = 0; i < arr.length; i = i + pageSize) {
    yield arr.slice(i, i + pageSize);
  }
}

const arr = [1, 2, 3, 4, 5, 6, 7]
const page = pagination(arr, 3);

alert (page.next().value);   // { value: [1, 2, 3], done: false }
alert (page.next().value);   // { value: [4, 5, 6], done: false }
alert (page.next().value);   // { value: [7], done: false }
alert (page.next().value);   // { value: undefined, done: true }
```

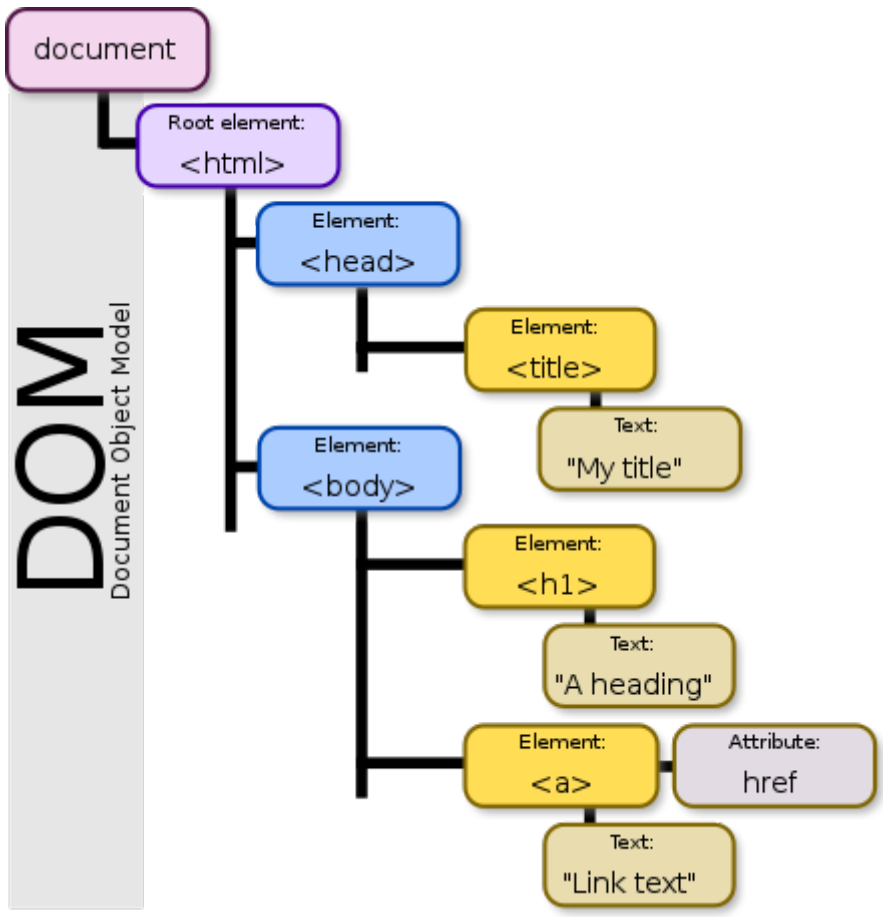
References

1. MDN: Generator function (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*)

Introduction to the Document Object Model (DOM)

HTML pages are internally implemented by a tree that contains the HTML elements (and CSS styles) as its nodes. This tree is called *Document-Object Model*, or DOM. JavaScript has full access to the DOM. It can navigate through the tree, modify the tree and nodes, ranging from simply adding new nodes to rearranging several areas on the page.

A hint about the terminology: Because of its closeness to XML, HTML uses the term *element*; and because of its structure as a tree, DOM uses the term *node*.



DOM-model

Nodes

When loading into the browser, the HTML document is broken down into a *tree* of *nodes*. For example, take a look at the following HTML snippet:

```
<div id="exampleDiv">This is an<br>example HTML snippet.</div>
```

Through DOM, JavaScript sees this snippet as four nodes.

- The `div`, from its start tag through its end tag, is one node. This `div` happens to have a *property* assigned inside its start tag. This property is named "id" and has the *value* "exampleDiv".

  The three other nodes in this example are inside the `div`. They are called *child nodes* of the `div`, because the `div` contains them. Conversely, the `div` is their *parent node*.

- The first child of the `div` is a *text node*, with the value "This is an". Text nodes contain only text; they never contain tags, which is why the tree stops here.
- The `br` tag is another node.
- The rest of the text is another text node.

Since the text nodes and the `br` tag all share the same parent, they are said to be *sibling nodes*.

Accessing nodes

You can access nodes of the DOM tree by various methods. One of them is `getElementById`.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      function go() {
        "use strict";
        const p = document.getElementById("p2");
        alert(p.innerHTML);
      }
    </script>
  </head>

  <body id="body">
    <p id="p1" style="background: aqua">one</p>
    <p id="p2" style="background: green">two</p>
    <p id="p3" style="background: red">three</p>
    <button onclick="go()">Show the second paragraph</button>
  </body>
</html>
```

When clicking on the button, the function `go` is called. It accesses the element with the id 'p2' and shows its content.

Accessing content

If you want to get access to the content of a node, you can use different properties of different classes: `Node.textContent` (https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent), `HTMLElement.innerText` (https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/innerText), or `Element.innerHTML` (https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML). But they are not equal; please consider the differences (https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent). To keep our examples clear and easy, we use `Element.innerHTML` whenever possible because it is very close to the HTML source code.

```
const exampleContent = document.getElementById("example").innerHTML;
```

Changing content

After accessing a node, you can change its content by assigning a new value to its content.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      function go() {
        "use strict";
        const p = document.getElementById("p2");
        p.innerHTML = "Another text";
      }
    </script>
  </head>
```

```
  <body id="body">
    <p id="p1" style="background: aqua">one</p>
    <p id="p2" style="background: green">two</p>
    <p id="p3" style="background: red">three</p>
    <button onclick="go()">Change the second paragraph</button>
  </body>
</html>
```

When clicking on the button, the function `go` is called. Again, it accesses the element with the id 'p2' and changes its content.

Modifying the tree structure

JavaScript can manipulate the structure of the DOM tree.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      function go() {
        "use strict";

        // read 'body' node
        const b = document.getElementById("body");
        // read second 'p' node
        const p = document.getElementById("p2");
        // moves it to the end of 'body'
        b.appendChild(p);
      }
    </script>
  </head>

  <body id="body">
    <p id="p1" style="background: aqua">one</p>
    <p id="p2" style="background: green">two</p>
    <p id="p3" style="background: red">three</p>
    <button onclick="go()">Move the second paragraph</button>
  </body>
</html>
```

When clicking on the button, the function `go` is called. It accesses the elements 'body' and 'p2', then, it moves the 'p' element to the end of the 'body'.

See also

- MDN DOM (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

Finding elements

To work with nodes of the DOM tree, you need to locate them directly or navigate to them beginning at a starting point. DOM's Document (https://developer.mozilla.org/en-US/docs/Web/API/Document) interface serves as an entry point into the web page's content. It offers a rich set of properties and methods to reach particular nodes. The methods return single nodes or an array of nodes.

We use the following HTML page to demonstrate the most important methods.

```
<!DOCTYPE html>
<html>
<head>
  <script>
    function show() {
      "use strict";
      // ...
    }
  </script>
  <style>
    .head_2 {
      display: flex;
      justify-content: center;
    }
    .text {
      padding-left: 1em;
      font-size: 1.4em;
    }
    .button {
      height:1.4em;
      width: 4em;
      margin-top: 1em;
      font-size: 1.2em;
      background-color: Aqua;
    }
  </style>
</head>

<body>
  <h1>An HTML header</h1>

  <h2 class="head_2">An HTML sub-header</h2>
  <div id="div_1">
    <p id="p1" class="text">Paragraph 1</p>
    <p id="p2" class="text">Paragraph 2</p>
    <p id="p3" class="text">Paragraph 3</p>
  </div>

  <div id="div_2">
    <h2>Another HTML sub-header</h2>
    <div id="div_3">
      <p>Another paragraph 1</p>
      <p>Another paragraph 2</p>
      <p>Another paragraph 3</p>
    </div>
  </div>

  <button class="button" onclick="show()">Go</button>

</body>
</html>
```

Clicking on the `button` invokes the function `show`. The examples should be included there.

Using ID

An easy-to-use, fast, and exact method to locate an individual element is to mark the element with the `id` property in the HTML, e.g., `<p id="p2">`, and use this `id` as the parameter to `getElementById()`. The following code snippet will locate the element and displays its content.

```
function show() {
  "use strict";
  const elem = document.getElementById("p2");
  alert(elem.innerHTML);
}
```

The `getElementById` method returns one single element (the first with this `id` if the `id` is not unique).

That is also true if the element is not a text node but a node with child nodes. The return value is a single element with all its child elements included.

```
function show() {
  "use strict";
  const elem = document.getElementById("div_3");
  alert(elem.innerHTML);
}
// expected output:
// <p>Another paragraph 1</p>
// <p>Another paragraph 2</p>
// <p>Another paragraph 3</p>
```

Using tag name

Another way to find elements on an HTML page is the `getElementsByTagName` method. It accepts a tag name, e.g., 'h1', 'div', or 'p', and returns **all** such elements in an array.

Here, we use the method to retrieve an array of all 'div' elements.

```
function show() {
  "use strict";

  // if you want to search in the complete document, you must specify 'document'
  let elemArray = document.getElementsByTagName("div");
  // loop over all array elements
  for (let i = 0; i < elemArray.length; i++) {
    alert(elemArray[i].innerHTML);
  }

  alert("Part 2");

  // if you want to search only a sub-tree, you must previously locate
  // the root of this sub-tree
  const elem = document.getElementById("div_2");
  elemArray = elem.getElementsByTagName("div");
  for (let i = 0; i < elemArray.length; i++) {
    alert(elemArray[i].innerHTML);
```

```
  }
}
```

Using class name

Next, elements can be located by an associated CSS class selector. Class selectors can have a complex syntax. Here, we use only the simple form of class names.

The example retrieves all elements that use the CSS class *text* - what is done by the 3 paragraphs of the first `div`. Please note, that the other paragraphs are not retrieved.

```
function show() {
  "use strict";

  let elemArray = document.getElementsByClassName("text");
  // loop over all array elements
  for (let i = 0; i < elemArray.length; i++) {
    alert(elemArray[i].innerHTML);
  }
}
```

Using a query selector

The shown locating methods use specialized semantics to locate elements. But there is also a general method that combines all of that - plus more.

Query selectors (https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Locating_DOM_elements_using_selectors) use a complex syntax consisting of HTML element ids, HTML element names, HTML attributes, CSS classes, positions, and more. They locate single elements or a list of elements. To retrieve the first element which satisfies the selector, use the `querySelector` method. If you want to retrieve all matching elements, use `querySelectorAll`.

```
function show() {
  "use strict";

  // '#' locates via ID, '.' locates via CSS class
  let elemArray = document.querySelectorAll("h1, #p2, .head_2");
  // loop over all array elements
  for (let i = 0; i < elemArray.length; i++) {
    alert(elemArray[i].innerHTML);
  }
}
```

Navigating the DOM tree

You can navigate in the DOM tree in the direction from the root to the leaves. This is done by locating an element and using this node as the new root for the following navigation steps.

```
function show() {
  "use strict";

  // start at 'div_2'
  const elem_1 = document.getElementById("div_2");
  // use this element as the new root for further selections
  const elemArray = elem_1.getElementsByTagName("h2");
  for (let i = 0; i < elemArray.length; i++) {
```

```
    alert(elemArray[i].innerHTML);
  }
  // only the child-'h2' is selected! The first 'h2' is ignored.
}
```

See also

- DOM methods (https://developer.mozilla.org/en-US/docs/Web/API/Document#instance_methods)

Exercises

... are available on another page (click here).

Changing elements

On this page, we show how to change two different things of an HTML element, respectively, DOM node.

- Its **content** (there is only one - or none)
- Any of its **attributes** (there may be many)

Please take note of this distinction between content and attributes.

```
<!-- in general: -->
<element_name attribute_name="attribute_value">content of the element</element_name>
<!-- a concrete example. 'href' is an attribute. 'Visit IANA...' is the content. -->
<a href="https://www.example.com">Visit IANA's example domain.</a>
```

Example page

We use the following example HTML page to demonstrate the possibilities.

```
<!DOCTYPE html>
<html>
<head>
  <script>
    function show() {
      "use strict";
      // ...
    }
  </script>
</head>

<body id="body" style="margin:2em">
  <p id="p1" style="background: aqua">A blue paragraph</p>

  <svg id="svgSrc" width="100" height="100" viewBox="0 0 100 100">
    <circle cx="50" cy="50" r="25" fill="green"/>
  </svg>

  <p />
  <a id="refToSomewhere" href="https://www.example.com">Visit IANA's example domain.</a>

  <p />
  <button id="buttonShow" onclick="show()">Start</button>
</body>
</html>
```

Clicking on the `button` invokes the function `show`. The examples should be included there.

Change the content

We use the example of a paragraph `p`. To change its content, the text, just assign the new value to its `innerHTML`.

```
function show() {
  "use strict";
  const elem = document.getElementById("p1");
  elem.innerHTML = "New text in the paragraph.";
```

```
}
```

Or, to do the same with a different HTML element, we change the SVG graphic.

```
function show() {
  "use strict";
  const elem = document.getElementById("svgSrc");
  elem.innerHTML = "<rect width='80' height='40' fill='blue'/>";
}
```

Because the new text is HTML code, you can 'misuse' this approach to add child nodes.

```
function show() {
  "use strict";
  const elem = document.getElementById("p1");
  elem.innerHTML = "New text in the paragraph.<p>next P</p><p>and even one more P</p>";
}
```

The script inserts two more paragraphs, but not behind the first one. They are within the first one.

```
<p id="p1">New text in the paragraph
  <p>next P</p>
  <p>and even one more P</p>
</p>
```

Change an attribute

In general, the syntax to change attributes is as follows:

```
element_name.attribute_name = "new value";
// or:
element_name.setAttribute("attribute_name", "new value");
```

The HTML element `a` knows a `href` attribute: `<a id="refToSomewhere" href="https://www.example.com">...</a>`. This `href` attribute can be changed:

```
function show() {
  "use strict";
  const elem = document.getElementById("refToSomewhere");
  elem.href = "https://en.wikibooks.org";
  elem.innerHTML = "Link changed";
}
```

First, the element is located. Second, the function assigns a new value to its attribute 'href' (and to the innerHTML).

The following example changes the `src` attribute of `img` element and the `value` attribute of `button` element

```
// The HTML
<img id="imgOne" src="myPicture.jpg">
<input id="buttonOne" value="I'm a button!">

// The JavaScript
document.getElementById("imgOne").src = "otherPicture.jpg";
```

```
const b = document.getElementById("buttonOne");
b.value = "I'm a changed button";
```

setAttribute()

The modification of attributes can also be done via the function `setAttribute`.

```
function show() {
  "use strict";
  const elem = document.getElementById("refToSomewhere");
  elem.setAttribute("href", "https://en.wikibooks.org");
  elem.innerHTML = "Link changed";
}
```

See also

- MDN setAttribute (https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute)

Exercises

... are available on another page (click here).

Adding elements

DOM's Document (https://developer.mozilla.org/en-US/docs/Web/API/Document) interface offers - among other things - functions that create new elements, including their attributes and content, and joins them together or into an existing DOM.

`createElement()` creates an element. `createAttribute()` creates an attribute that can be assigned to this new or an already existing element. `setAttribute()` creates an attribute and links it to an existing element. `appendChild()` integrate an element into another.

Creating elements

```
// an <p> element
const p = document.createElement("p");
// its content
p.innerHTML = "The new paragraph.";
```

Now, the element and its content are created. But until here, they are not part of a DOM. They exist only in the memory of the JavaScript engine.

To integrate them into the page, we retrieve the body or any other element of an existing page and append the new element as its last element.

```
const body = document.getElementById("body");
body.appendChild(p);
```

All in all, the HTML plus JavaScript looks like this:

```
<!DOCTYPE html>
<html>
<head>
  <script>
    function show() {
      "use strict";

      // an create an <p> element
      const p = document.createElement("p");
      // create its content
      p.innerHTML = "The new paragraph.";

      // integrate it into the body
      const body = document.getElementById("body");
      body.appendChild(p);
    }
  </script>
</head>

<body id="body" style="margin:2em">
  <button id="buttonShow" onclick="show()">Start</button>
</body>
</html>
```

The original page does not contain a paragraph. But after you click on the button, the paragraph is integrated into the page and is visible. Btw: You can click more than once on the button. What will happen?

Creating attributes

Attributes are created with either the `createAttribute()` or the `setAttribute()` function. The first of the two acts like the above shown `createElement()` function. It creates the new attribute only in memory without a connection to other elements. Because `setAttribute()` integrates the new attribute directly into an element, we use this variant.

The example uses the `a` element with its `href` attribute.

```
// an <a> element
const anchor = document.createElement("a");
// its content
anchor.innerHTML = "The IANA example daomain.";
// its 'href' attribute
anchor.setAttribute("href", "https://www.example.com");
```

Now, the element, a single attribute, and the element's content are created. Again, we integrate them into the page as we have done above.

All in all, the HTML plus JavaScript looks like this:

```
<!DOCTYPE html>
<html>
<head>
  <script>
    function show() {
      "use strict";

      // an create an <a> element
      const anchor = document.createElement("a");
      // create its content
      anchor.innerHTML = "The IANA example domain.";
      // create its 'href' attribute
      anchor.setAttribute("href", "https://www.example.com");
        // see below: anchor.href = "https://www.example.com";

      // integrate the element inclusive its attribute into the body
      const body = document.getElementById("body");
      body.appendChild(anchor);
      /* now, the body looks like this:
        <button id="buttonShow" onclick="show()">Start</button>
        <a href="https://www.example.com">The IANA example domain.</a>
      */
    }
  </script>
</head>

<body id="body" style="margin:2em">
  <button id="buttonShow" onclick="show()">Start</button>
</body>
</html>
```

The original page does not contain a link. But after you click on the button, the link to the IANA example page is integrated into the page and is usable.

Alternative syntax

One of the previous pages has explained how to change attributes with a different syntax.

```
element_name.attribute_name = "new value";
```

Just use the element plus its attribute name and assign the attribute value to it. If you change the previous example to this syntax, you will reach the same behavior of adding the link.

```
  anchor.href = "https://www.example.com";
  // instead of the above:
  // anchor.setAttribute("href", "https://www.example.com");
```

Join the puzzles pieces

The shown functions create elements and attributes. Such new objects can be joined together to create huger parts - of course in a nested way. And they can be joined to an already existing HTML page, respectively, the DOM tree.

```
const div = document.getElementById("div_1");
const anchor = document.createElement("a");
div.appendChild(anchor);
```

'Misusing' innerHTML

The content of an element can be changed by assigning a new value to its property `innerHTML`. If this new value contains the string representation of an HTML fragment, the assignment creates child nodes within the element. That's possible but not the intended way of using `innerHTML`.

```
const elem = document.getElementById("p1");
elem.innerHTML = "New text in the paragraph.<p>next P</p><p>and even one more P</p>";
```

.. leads to ..

```
<p id="p1">New text in the paragraph
  <p>next P</p>
  <p>and even one more P</p>
</p>
```

The JavaScript fragment inserts two more paragraphs, but not behind the first one. They exist within the first one.

write()

The antiquated function `document.write()` was able to insert new elements into an HTML page. Its usage is strongly discouraged (https://developer.mozilla.org/en-US/docs/Web/API/Document/write) nowadays.

See also

- MDN Create Element (https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement)
- MDN Set Attribute (https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute)
- MDN Append Child (https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)

Exercises

... are available on another page (click here).

Removing elements

HTML pages and DOM objects are hierarchically structured. Every element and attribute belongs to exactly **one** parent. To delete an element or attribute, first, you must locate the parent element. The remove operation can be done on this object.

Remove elements

Elements are removed with the `removeChild` function. To delete the `<p>` element from the `<div>` element in the following example

```
<div id="parent">
  <p id="child">I'm a child!</p>
</div>
```

the JavaScript code is ...

```
// get elements
const parent = document.getElementById("parent");
const child = document.getElementById("child");

// delete child
parent.removeChild(child);
```

... and the remaining HTML structure will be

```
<div id="parent"></div>
```

Children of children

If an element is removed, all of its children are removed as well. By this, you can remove huge parts of the DOM with one command if they have a common root. E.g., remove a complete list:

```
<div id="div_1">
  <ul id="listOfNames">
    <li>Albert</li>
    <li>Betty</li>
    <li>Charles</li>
  </ul>
</div>
```

The JavaScript fragment removes the `<ul>` element as well as all `<li>` elements.

```
const parent = document.getElementById("div_1");
const child = document.getElementById("listOfNames");
```

```
parent.removeChild(child);
```

parentNode

To remove an element, you need to know its parent element. If you can locate only the child, but for some reason, not the parent, the child's property `parentNode` shows you the way.

```
// get the child element
const child = document.getElementById("child");

// retrieve the parent
const parent = child.parentNode; // no parenthesis ()

// remove the child element from the document
parent.removeChild(child);
```

Remove attributes

Attributes are removed with the `removeAttribute` function. To delete the `href` attribute from the `<a>` element in the following example

```
<a id="anchor" href="https://en.wikibooks.org">Wikibook</a>
```

the JavaScript code is:

```
// get element
const anchor = document.getElementById("anchor");
// remove attribute
anchor.removeAttribute("href");
```

The element itself, including the text of the link, keeps alive, but you cannot navigate anymore.

See also

- MDN Remove Child (https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild)
- MDN Remove Attribute (https://developer.mozilla.org/en-US/docs/Web/API/Element/removeAttribute)

Restructure DOM

Besides adding and removing nodes, a common activity on trees is the rearranging of nodes respectively of sub-trees. In some of the previous examples, we have seen that `appendChild` inserts a node as the last child of a parent node. Of course, this is not delimited to the case that the child is currently created. The same operation is possible for an existing node. Hence it is an appropriate function to perform rearrangements.

Example page

We use the following example HTML page to demonstrate the possibilities.

```
<!DOCTYPE html>
<html>
<head>
  <script>
    function show() {
      "use strict";
      // ...
    }
  </script>
</head>

<body id="body" style="margin:2em">
  <h1>The magic manipulator</h1>
  <button id="buttonShow" onclick="show()" style="margin:1em 0em 1em 0em">Start</button>

  <div id="div_1">Our sweet products
    <ul id="ul_1">
      <li id="li_11">Ice creme</li>
      <li id="li_12">Chocolate</li>
      <li id="li_13">Coffee</li>
    </ul>
  </div>

  <div id="div_2">Additional offers
    <ul id="ul_2">
      <li id="li_21">Creme</li>
      <li id="li_22">Sugar</li>
      <li id="li_23">Cakes</li>
    </ul>
  </div>

</body>
</html>
```

Move an element to the end of siblings

We identify two elements and move them to the end of another node. This other node is not necessarily the old parent. But after the movement, it becomes the new parent.

```
function show() {
  "use strict";

  // select the first <ul> as the new parent
  const ul_1 = document.getElementById("ul_1");

  // select two <li> elements to get moved
  const li_11 = document.getElementById("li_11");
  const li_23 = document.getElementById("li_23"); // The 'cakes'
```

```
  // move the two children
  ul_1.appendChild(li_11);
  ul_1.appendChild(li_23);
}
```

As you see, the 'Ice creme' is moved to the end (temporarily) of its product group. After that, the 'cakes', which were a child of the second product group, were moved to the end of the first product group.

Move an element before a sibling

You can move an element to any position within a group of siblings. ('Siblings' are nodes with a common parent node.) First, you must locate the parent. Next, you locate the child, which shall become the new successor of the element. When both nodes are known, the function `insertBefore` inserts the element to this position.

As an example, we move the 'Cakes' to the first place of the first product group.

```
function show() {
  "use strict";

  // select the first <ul> element; it acts as the parent
  const ul_1 = document.getElementById("ul_1");
  // select the first li element; we need it for positioning
  const li_11 = document.getElementById("li_11");
  // select 'Cakes' li element
  const li_23 = document.getElementById("li_23");

  // move the 'Cakes' to the first place of the first product group
  ul_1.insertBefore(li_23, li_11);
}
```

Here, the 'Cakes' become the first element of the first product group. With the same commands you can move them to any position in the sequence of siblings. Just locate the sibling that shall be his new successor instead of 'li_11'.

Attributes

The sequence of attributes within an element is in no way relevant. Hence, there is no need and no function to rearrange them. When a DOM gets serialized, programs may do it in different ways (original sequence, alphabetically, ...).

Exercises

... are available on another page (click here).

See also

- MDN Append Child (https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)
- MDN Insert Before (https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore)

Changing element styles

As you have seen in previous chapters, the attributes of an element can be modified by JavaScript. Two attributes, the `class` and `style`, influences the visual representation of an element. They contain CSS code.

```
<!DOCTYPE html>
<html>
<head>
  <script>
  function toggle() {
    "use strict";
    // ...
  }
  </script>
<style>
  .divClassGreen {
    background-color: green;
    padding: 2em;
    margin: 1em
  }
  .divClassRed {
    background-color: red;
    padding: 2em;
    margin: 1em
  }
</style>
</head>

<body id="body">
  <div id="div_1" class="divClassGreen">A DIV element</div>
  <div id="div_2" style="background-color: blue; padding: 2em; margin: 1em">Another DIV
element</div>
  <button id="buttonToggle" onclick="toggle()" style="margin:1em 0em 1em 0em">Start</button>
</body>
</html>
```

The `class` attribute identifies a CSS class that is created in the `style` element of HTML. The `style` attribute defines CSS rules inline (locally).

To modify them, handle them like any other attribute. They do not have special rules or exceptions.

An example

We use the above HTML file; only the JavaScript function is changed. When the button is clicked, the function assigns the CSS class 'divClassRed' to 'div_1' and it changes the inline 'style' attribute of 'div_2' to a different value.

```
  function toggle() {
    "use strict";

    // locate the elements to be changed
    const div_1 = document.getElementById("div_1");
    const div_2 = document.getElementById("div_2");

    // modify its 'class' attribute
    div_1.setAttribute("class", "divClassRed");

    // an 'inline' modification
    div_2.setAttribute("style", "background-color: silver; padding: 4em; margin: 2em");
```

```
    //or: div_2.style = "background-color: silver; padding: 4em; margin: 2em";
}
```

The 'style' attribute stores the CSS properties like 'color' or 'padding' in its own properties. This correlates with the general JavaScript object rules. Therefore the following syntax is equivalent to the previous `div_2.setAttribute` call.

```
div_2.style.backgroundColor = "silver"; // see below: camel-case
div_2.style.padding = "4em";
div_2.style.margin = "2em";
```

Properties of 'style'

In CSS, some properties are defined with a hyphen in their name, e.g., 'background-color' or 'font-size'. When you use them in JavaScript in the syntax of a **property of style**, the names change slightly. The character after the hyphen must be written in upper-case, and the hyphen disappears: 'style.backgroundColor' or 'style.fontSize'. This is called *camel-case*.

```
div_1.style.fontSize = "2em"; // the font's size as property of 'style'
/*
The next line would run into a syntax error because the hyphen
would be interpreted as a minus operation.
div_1.style.font-size = "2em";
*/
```

All other places where such names appear in CSS do not change. Especially the shown syntax with hyphens inside the HTML `<style>` element, as well as the use in the form of an inline definition, keeps unchanged.

Exercises

... are available on another page (click here).

See also

- MDN: CSS basics (https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics)

Handling DOM events

Applications with a user interface - and other application types - are predominantly driven by *events*. Here we focus on DOM events (https://w3c.github.io/uievents/). They originate (or *fire*) from specific actions or situations in a browser or a native app, e.g., a user clicks with the mouse, types a keyboard key, or works on a touch screen; a visual object is 'dragged & dropped', 'copied & pasted', or resized; the HTML page is loaded or shall be unloaded; the browser or a visual object gets or loses the focus; and much more. It's also possible that the application creates events programmatically (*dispatch*).

An event is related to its originating object and with a JavaScript statement; that is, in most cases, a call to a function that is denoted as the *event handler*. The JavaScript part is invoked after the event arises. Common actions are communication with a server, validation of user input, or modification of DOM or graphics.

Create and invoke events

Embedded in HTML

A short example shows how events are defined in an HTML page, fire, and execute. This syntax version is denoted as *inline JavaScript*.

```
<!DOCTYPE html>
<html>
<head>
  <script>
  function handleEvent(evt) {
    "use strict";
    alert("Perform some actions via a JavaScript function.");
  }
  </script>
</head>

<body id="body" style="margin:2em">
  <!-- inline all statements; unusual -->
  <button onclick="const msg='Direct invocation of small actions.'; alert(msg);">First
button</button>
  <!-- call a function (the event handler) -->
  <button onclick="handleEvent(event)">Second button</button>
</body>
</html>
```

When a user clicks on one of the buttons, the browser reads the button's attribute `onclick`, creates a JavaScript object that contains all properties of the event, and executes the JavaScript statement(s). Mostly, the JavaScript part is a call to a function. That function is denoted the *event handler*. It receives the JavaScript object as its first parameter. Only in very simple cases, the complete JavaScript script is inlined.

Of course, the called function must exist somewhere. Some standard functions like `alert()` are predefined and provided by the browser. Your self-written functions exist within the HTML tag `<script>`, or the tag refers to a file or URL where they exist.

Hint: Embedding event definitions directly into HTML is called *inline JavaScript*. It's the earliest method of registering event handlers, but it tends to make the source hard to read for non-trivial applications. Therefore it can be seen as being a less desirable technique (https://developer.mozilla.org/en-US/docs/Lear

n/JavaScript/Building_blocks/Events#inline_event_handlers_%E2%80%94_dont_use_these) than other unobtrusive techniques; see next chapter. The use of *inline JavaScript* can be considered to be similar in nature to that of using *inline CSS*, where HTML is styled by putting CSS in `style` attributes.

Nevertheless, the Wikibook on hand will use *inline JavaScript* often in its demonstration pages because the syntax is short and the concept is easy to understand.

Programmatically in JavaScript

JavaScript knows two ways to register an event handler for an HTML element. First, the event handler function can be directly assigned to the element's properties `onxxx` (onclick, onkeydown, onload, onfocus, ...). Their name starts with 'on' and ends with the value of the event type. Second, the function `addEventListener` registers the event type and the event handler.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <script>
5    function register() {
6      "use strict";
7      const p1 = document.getElementById("p1");
8
9      // do it this way ...
10     p1.onclick = showMessage; // no parenthesis ()
11     // ... or this way (prefered)
12     //p1.addEventListener('click', showMessage);
13     alert("The event handler is assigned to the paragraph.");
14   }
15   function showMessage(evt) {
16     "use strict";
17     // the parameter 'evt' contains many information about
18     // the event, e.g., the position from where it originates
19     alert("A click-event to the paragraph. At position " +
20           evt.clientX + " / " + evt.clientY + ". Event type is: " + evt.type);
21   }
22   </script>
23 </head>
24
25 <body>
26   <h1>Register an event</h1>
27   <p id="p1" style="margin:2em; background-color:aqua">
28      A small paragraph. First without, then with an event handler.
29   </p>
30   <button onclick="register()" id="button_1">A button</button>
31 </body>
32 </html>
```

The onclick event handler 'register' of button 'button_1' is registered with the above *inline JavaScript* syntax. When the page loads, only this event handler is known. Clicks to the paragraph 'p1' don't trigger any action because it does not have any event handler so far. But when the button gets pressed, the handler of button 'button_1' registers the second event handler, the function 'showMessage'. Now, after a click on the paragraph, the alert "A click-event to the paragraph..." occurs.

The registration is done in line 10 `p1.onclick = showMessage`. The noticeable difference to the *inline JavaScript* is that there are no parenthesizes. The *inline JavaScript* calls the function `showMessage` and hence needs to use parenthesizes. The function `register` does NOT call `showMessage`. It uses only its name for the registration process.

The alternative to assigning the function to the paragraph's 'onclick' property is the use of the function `addEventListener`. It acts on the element 'p1' and takes two parameters. The first one is the event type (click, keydown, ...). Such event types correlate with the *onxxx* names in that they miss the first two characters 'on'. The second parameter is the name of the function that acts as the event handler.

You can test the example by commenting out either line 10 or line 12. Line 12 with the `addEventListener` function is the preferred version.

Event types

Different kinds of events exist depending on the kind of the originating element. The complete list of event types is incredibly huge (MDN) (https://developer.mozilla.org/en-US/docs/Web/Events#event_index) (W3schools) (https://www.w3schools.com/jsref/dom_obj_event.asp). We show some important types and examples.

Name	Description
`blur`	An input element loses focus
`change`	An element gets modified
`click`	An element gets clicked
`dblclick`	An element gets double-clicked
`error`	An error occurred loading an element
`focus`	An input element received focus
`keydown`	A key was pressed when an element had focus
`keyup`	A key was released when the element had focus
`load`	An element was loaded
`mouseenter`	The mouse pointer was moved into the element
`mousemove`	The mouse pointer moves while inside the element
`mousedown`	The mouse button was pressed on the element
`mouseup`	The mouse button was released on the element
`mouseleave`	The mouse pointer was moved out of the element
`reset`	The form's reset button was clicked
`resize`	The containing window or frame was resized
`select`	Some text within the element was selected
`submit`	A form is being submitted
`unload`	The content is being unloaded (e.g., window being closed)

The following example demonstrates some different event types.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <script>
5    function registerAllEvents() {
6      "use strict";
7      // register different event types
```

```
 8      document.getElementById("p1").addEventListener("click", handleAllEvents);
 9      document.getElementById("p2").addEventListener("dblclick", handleAllEvents);
10      document.getElementById("p3").addEventListener("mouseover", handleAllEvents);
11      document.getElementById("t1").addEventListener("keydown", handleAllEvents);
12      document.getElementById("t2").addEventListener("select", handleAllEvents);
13      document.getElementById("button_1").addEventListener("mouseover", handleAllEvents);
14    }
15    function handleAllEvents(evt) {
16      "use strict";
17      alert("An event occurred from element: " +
18            evt.target.id + ". Event type is: " + evt.type);
19    }
20    </script>
21  </head>
22
23  <body onload="registerAllEvents()" style="margin:1em">
24    <h1 id="h1" style="background-color:aqua">Check Events</h1>
25    <p  id="p1" style="background-color:aqua">A small paragraph. (click)</p>
26    <p  id="p2" style="background-color:aqua">A small paragraph. (double click)</p>
27    <p  id="p3" style="background-color:aqua">A small paragraph. (mouse over)</p>
28    <p  style="background-color:aqua">
29      <textarea id="t1" rows="1" cols="50">(key down)</textarea>
30    </p>
31    <p  style="background-color:aqua">
32      <textarea id="t2" rows="1" cols="50">(select)</textarea>
33    </p>
34    <button id="button_1">A button (mouse over)</button>
35  </body>
36  </html>
```

When the page is loaded, the *onload* event of the `body` is fired. Please notice that here the 'on' prefix is necessary because it's the *inline JavaScript* syntax (line 23). The called function *registerAllEvents* locates diverse HTML elements and registers event handlers of different types (lines 8 - 13). Often you will register different functions, but to keep things easy, we register in this example the same function *handleAllEvents* to all elements. This function reports the event type and the originating HTML element.

Event properties

The event is always passed to the event handler as its first parameter in the form of a JavaScript object. JavaScript objects consist of properties; properties are key/value pairs. In all cases, one of the keys is 'type'. It contains the event's type; some of its possible values are shown in the above table. Depending on the event type, a lot of other properties are available.

Here are some examples of more or less important properties.

Name	Description
`button`	Returns which mouse button was clicked
`clientX`	Returns the horizontal coordinate of the mouse pointer within the local coordinates: scrolled-out parts don't count
`clientY`	Returns the vertical coordinate of the mouse pointer within the local coordinates: scrolled-out parts don't count
`code`	Returns a textual representation of the pressed key, e.g., "ShiftLeft" or "KeyE"
`key`	Returns the value of the pressed key, e.g., "E"
`offsetX`	Returns the horizontal coordinate of the mouse pointer within the target DOM element
`offsetY`	Returns the vertical coordinate of the mouse pointer within the target DOM element
`pageX`	Returns the horizontal coordinate of the mouse pointer within the page coordinates - including scrolled-out parts
`pageY`	Returns the vertical coordinate of the mouse pointer within the page coordinates - including scrolled-out parts
`screenX`	Returns the horizontal coordinate of the mouse pointer within the complete monitor coordinates
`screenY`	Returns the vertical coordinate of the mouse pointer within the complete monitor coordinates
`target`	Returns the element that triggered the event
`timeStamp`	Returns the number of milliseconds between element creation and event creation
`type`	Returns the type of the element that triggered the event
`x`	An alias for clientX
`y`	An alias for clientY

An example of accessing a property is given in the following script.

```
<!DOCTYPE html>
<html>
<head>
  <script>
  function changeTitle(evt) {
    "use strict";
    const xPos = evt.x;
    const yPos = evt.y;
    document.title = [xPos, yPos];
  }
  </script>
</head>

<body onmousemove="changeTitle(event)" style="margin:1em; border-width: 1px; border-
style: solid;">
  <h1 id="h1" style="background-color:aqua">Check Events</h1>
  <p id="p1" style="margin:2em; background-color:aqua">A small paragraph.</p>
  <p id="p2" style="margin:2em; background-color:aqua">Another small paragraph.</p>
  <button id="button_1">Button</button>
</body>
</html>
```

A mouse-move event is registered for the `body`. Whenever the mouse moves across the body, the event is triggered. The event handler reads the x/y properties out of the JavaScript object and shows them in the title of the browser's active tab.

removeEventListener

Similar to `addEventListener` the function `removeEventListener` removes an event listener from an element.

Synthetic events

The system offers the above-shown rich set of event types. Additionally, you can create your own events and trigger them from your application.

First, you create a function with one parameter, the event handler. Next, you register this event handler for an element. So far, everything is the same as with predefined event types.

```
function register() {
  "use strict";

  // ...

  // choose an arbitrary event type (first parameter of 'addEventListener')
  // and register function on element
  document.getElementById("p1").addEventListener("syntheticEvent", f);
}
// the event handler for the non-system event
function f(evt) {
  alert("Invocation of the synthetic event on: " + evt.target.id +
        " The event type is: " + evt.type);
}
```

Now you can trigger this event in any part of your application. To do so, you create a new event of precisely the chosen type and fire it with a call to `dispatchEvent`.

```
function triggerEvent(evt) {
  "use strict";
  // create a new event with the appropriate type
  const newEvent = new Event("syntheticEvent");
  // trigger this event on element 'p1'
  document.getElementById("p1").dispatchEvent(newEvent);
}
```

For test purposes, we bind this functionality to the button. The complete page now looks like this:

```
<!DOCTYPE html>
<html>
<head>
  <script>
  function register() {
    "use strict";
    document.getElementById("p1").addEventListener("click", showMessage);
    document.getElementById("p2").addEventListener("click", showMessage);
    document.getElementById("button_1").addEventListener("click", triggerEvent);

    // choose an arbitrary event type (first parameter of 'addEventListener')
    // and register function on element
    document.getElementById("p1").addEventListener("syntheticEvent", f);
  }
```

```
  // the event handler for the non-system event
  function f(evt) {
    "use strict";
    alert("Invocation of the synthetic event on: " + evt.target.id +
          " The event type is: " + evt.type);
  }

  function showMessage(evt) {
    "use strict";
    // the parameter 'evt' contains many information about
    // the event, e.g., the position from where it originates
    alert("A click event to element: " + evt.target.id +
          " The event type is: " + evt.type);
  }

  function triggerEvent(evt) {
    "use strict";
    // create a new event with the appropriate type
    const newEvent = new Event("syntheticEvent");
    // trigger this event on element 'p1'
    document.getElementById("p1").dispatchEvent(newEvent);
  }

  </script>
</head>

<body onload="register()">
  <h1>Create an event programmatically.</h1>
  <p id="p1" style="margin:2em; background-color:aqua">A small paragraph.</p>
  <p id="p2" style="margin:2em; background-color:aqua">Another small paragraph.</p>
  <button id="button_1">Button</button>
</body>
</html>
```

At the beginning, the button listens to click events, and 'p1' listens to events of type 'click' as well as of type 'syntheticEvent'. When the button is clicked, his event handler 'triggerEvent' creates a new event of type 'syntheticEvent' and fires it on 'p1' (what is the primary purpose of this example). The event handler `showMessage` shows a message without 'p1' being clicked. In other words: The event on 'p1' occurs without a click on 'p1'.

Possibly you need in the event handler some data from the calling function, e.g., the text of an error-message, the data of an HTTP response, ... You can pass such data by using the `CustonEvent` and its property 'detail':

```
const newEvent = new CustomEvent("syntheticEvent", {detail: "A short message."});
```

Access the data in the event handle:

```
function f(evt) {
  "use strict";
  alert("Invocation of the synthetic event on: " + evt.target.id +
        " The event type is: " + evt.type + ". " + evt.detail);
}
```

(A)synchronous behaviour

Most events are handled synchronously, e.g., 'click', 'key', or 'mouse'. But there are a few exceptions that are handled asynchronously, e.g., 'load' [1] (https://w3c.github.io/uievents/#sync-async) [2] (https://w3c.github.io/uievents/#event-types-list). 'Synchronous' means that the sequence of invocations is the same as the

sequence of their creations. Clicking on Button A, B, and then C leads to the invocation of A's, B's, and then C's event handler in exactly this sequence. In contrast, 'asynchronous' events can lead to the invocation of the correlated handlers in an arbitrary sequence.

You must distinguish this question, the invocation of event handlers, from the implementation of their bodies. Every implementation may act strictly sequential or may contain asynchronous calls - it depends on your intention. Typical asynchronous calls are HTTP requests or database queries. Of course, they can be part of the handler of a click event.

Exercises

... are available on another page (click here).

See also

- MDN: Event handlers introduction (https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Event_handlers)
- MDN: Event reference and types (https://developer.mozilla.org/en-US/docs/Web/Events)
- W3Schools: Events (https://www.w3schools.com/jsref/dom_obj_event.asp)
- W3Schools: EventListener (https://www.w3schools.com/js/js_htmldom_eventlistener.asp)

Exercises

You can solidify your JavaScript knowledge with some additional exercises and games. The pages don't offer additional information; they use the already shown language aspects plus access to the DOM tree.

To keep everything manageable within a WikiBook, we put the source code always into one file. You may divide it into different files on your computer, e.g., separate HTML-, CSS-, JS-, and image-files. For the same reason, the examples don't use any games engine, framework, library, or server access. They are standalone scripts that need nothing but a browser.

The examples are divided into two groups. The first group doesn't use a graphical context. They rely purely on events and pure HTML elements like text, buttons, links, or colors. The second group additionally contains a `canvas`. Within such a `canvas`, you can use graphical elements like circles, rectangles, lines, ...

Non-graphical examples

- Guess a number
- Check your reaction time
- TicTacToe

Examples with graphic

- Introduction
- Speed; Detect collisions
- Moving Walls

Debugging

JavaScript Debuggers

Firebug

- Firebug (http://www.getfirebug.com/) is a powerful extension for Firefox that has many development and debugging tools including JavaScript debugger and profiler.

Venkman JavaScript Debugger

- Venkman JavaScript Debugger (http://www.hacksrus.com/~ginda/venkman/) (for Mozilla based browsers such as Netscape 7.x, Firefox/Phoenix/Firebird and Mozilla Suite 1.x)
- Introduction to Venkman (http://web.archive.org/web/20040704044520/devedge.netscape.com/viewsource/2002/venkman/01/)
- Using Breakpoints in Venkman (http://web.archive.org/web/20040603085323/devedge.netscape.com/viewsource/2003/venkman/01/)

Internet Explorer debugging

- Microsoft Script Debugger (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sdbug/Html/sdbug_1.asp) (for Internet Explorer) The script debugger is from the Windows 98 and NT era. It has been succeeded by the Developer Toolbar
- Internet Explorer Developer Toolbar (http://www.microsoft.com/downloads/details.aspx?FamilyID=2f465be0-94fd-4569-b3c4-dffdf19ccd99&displaylang=en)
- Microsofts Visual Web Developer Express (http://www.microsoft.com/express/vwd/) is Microsofts free version of the Visual Studio IDE. It comes with a JS debugger. For a quick summary of its capabilities see [3] (http://www.berniecode.com/blog/2007/03/08/how-to-debug-javascript-with-visual-web-developer-express/)
- Internet Explorer 8 (http://www.microsoft.com/windows/Internet-explorer/) has a firebug-like Web development tool by default (no add-on) which can be accessed by pressing F12. The Web development tool also provides the ability to switch between the IE8 and IE7 rendering engines.

Safari debugging

Safari includes a powerful set of tools that make it easy to debug, tweak, and optimize a website for peak performance and compatibility. To access them, turn on the Develop menu in Safari preferences. These include Web Inspector, Error Console, disabling functions, and other developer features. The Web Inspector gives you quick and easy access to the richest set of development tools ever included in a browser. From viewing the structure of a page to debugging JavaScript to optimizing performance, the Web Inspector presents its tools in a clean window designed to make developing web applications more efficient. To activate it, choose Show Web Inspector from the Develop menu. The Scripts pane features the powerful JavaScript Debugger in Safari. To use it, choose the Scripts pane in the Web Inspector and click

Enable Debugging. The debugger cycles through your page's JavaScript, stopping when it encounters exceptions or erroneous syntax. The Scripts pane also lets you pause the JavaScript, set breakpoints, and evaluate local variables.[1]

JTF: JavaScript Unit Testing Farm

- JTF (http://jtf.ploki.info) is a collaborative website that enables you to create test cases that will be tested by all browsers. It's the best way to do TDD and to be sure that your code will work well on all browsers.

jsUnit

- jsUnit (http://www.jsunit.net/)

built-in debugging tools

Some people prefer to send debugging messages to a "debugging console" rather than use the alert() function[4] (http://osteele.com/archives/2006/03/inline-console)[5] (http://www.webreference.com/js/column108/5.html)[6] (http://www.experts-exchange.com/Web/Web_Languages/JavaScript/Q_21380186.html). Following is a brief list of popular browsers and how to access their respective consoles/debugging tools.

- Firefox: `Ctrl+Shift+K` opens an error console.
- Opera (9.5+): `Tools >> Advanced >> Developer Tools` opens Dragonfly.
- Chrome: `Ctrl+Shift+J` opens chrome's "Developer Tools" window, focused on the "console" tab.
- Internet Explorer: `F12` opens a firebug-like Web development tool that has various features including the ability to switch between the IE8 and IE7 rendering engines.
- Safari: `Cmd+Alt+C` opens the WebKit inspector for Safari.

Common Mistakes

- Carefully read your code for typos.
- Be sure that every "(" is closed by a ")" and every "{" is closed by a "}".
- Trailing commas in Array and Object declarations will throw an error in Microsoft Internet Explorer but not in Gecko-based browsers such as Firefox.

```
// Object
var obj = {
  'foo'   : 'bar',
  'color' : 'red', //trailing comma
};

// Array
var arr = [
  'foo',
  'bar', //trailing comma
];
```

- Remember that JavaScript is case sensitive. Look for case related errors.
- Don't use Reserved Words as variable names, function names or loop labels.

- Escape quotes in strings with a "\" or the JavaScript interpreter will think a new string is being started, i.e:

  ~~`alert('He's eating food');`~~ should be
  `alert('He\'s eating food');` or
  `alert("He's eating food");`

- When converting strings to numbers using the parseInt function, remember that "08" and "09" (e.g. in datetimes) indicate an octal number, because of the prefix zero. Using parseInt using a radix of 10 prevents wrong conversion. `var n = parseInt('09',10);`
- Remember that JavaScript is platform independent, but is not browser independent. Because there are no properly enforced standards, there are functions, properties and even objects that may be available in one browser, but not available in another, e.g. Mozilla / Gecko Arrays have an indexOf() function; Microsoft Internet Explorer does not.

Debugging Methods

Debugging in JavaScript doesn't differ very much from debugging in most other programming languages. See the article at Computer Programming Principles/Maintaining/Debugging.

Following Variables as a Script is Running

The most basic way to inspect variables while running is a simple alert() call. However some development environments allow you to step through your code, inspecting variables as you go. These kind of environments may allow you to change variables while the program is paused.

Browser Bugs

Sometimes the browser is buggy, not your script. This means you must find a workaround.

Browser bug reports (http://www.quirksmode.org/bugreports/)

browser-dependent code

Some advanced features of JavaScript don't work in some browsers.

Too often our first reaction is: Detect which browser the user is using, then do something the cool way if the user's browser is one of the ones that support it. Otherwise skip it.

Instead of using a "browser detect", a much better approach is to write "object detection" JavaScript to detect if the user's browser supports the particular object (method, array or property) we want to use.[7] (http://www.quirksmode.org/js/support.html) [8] (http://pageresource.com/jscript/jobdet.htm)

To find out if a method, property, or other object exists, and run code if it does, we write code like this:

```
var el = null;
if (document.getElementById) {
  // modern technique
  el = document.getElementById(id);
} else if (document.all) {
  // older Internet Explorer technique
  el = document.all[id];
```

```
} else if (document.layers) {
  // older Netscape Web browser technique
  el = document.layers[id];
}
```

Further reading

- "JavaScript Debugging" (http://www.mozilla.org/docs/web-developer/js/debugging/) by Ben Bucksch

References

1. "Safari - The best way to see the sites." (in English) (HTML). Apple. http://www.apple.com/safari/features.html#developer. Retrieved 2015-03-09.

Optimization

JavaScript optimization

Optimization Techniques

- High Level Optimization
  - Algorithmic Optimization (Mathematical Analysis)
  - Simplification
- Low Level Optimization
  - Loop Unrolling
  - Strength Reduction
  - Duff's Device
  - Clean Loops
- External Tools & Libraries for speeding/optimizing/compressing JavaScript code

Common Mistakes and Misconceptions

String concatenation

Strings in JavaScript are immutable objects. This means that once you create a string object, to modify it, another string object must theoretically be created.

Now, suppose you want to perform a ROT-13 on all the characters in a long string. Supposing you have a rot13() function, the obvious way to do this might be:

```
var s1 = "the original string";
var s2 = "";

for (i = 0; i < s1.length; i++) {
  s2 += rot13(s1.charAt(i));
}
```

Especially in older browsers like Internet Explorer 6, this will be very slow. This is because, at each iteration, the entire string must be copied before the new letter is appended.

One way to make this script faster might be to create an array of characters, then join it:

```
var s1 = "the original string";
var a2 = new Array(s1.length);
var s2 = "";

for (i = 0; i < s1.length; i++) {
  a2[i] = rot13(s1.charAt(i));
```

```
}
s2 = a2.join('');
```

Internet Explorer 6 will run this code faster. However, since the original code is so obvious and easy to write, most modern browsers have improved the handling of such concatenations. On some browsers the original code may be faster than this code.

A second way to improve the speed of this code is to break up the string being written to. For instance, if this is normal text, a space might make a good separator:

```
var s1 = "the original string";
var c;
var st = "";
var s2 = "";

for (i = 0; i < s1.length; i++) {
  c = rot13(s1.charAt(i));
  st += c;
  if (c == " ") {
    s2 += st;
    st = "";
  }
}
s2 += st;
```

This way the bulk of the new string is copied much less often, because individual characters are added to a smaller temporary string.

A third way to really improve the speed in a for loop, is to move the [array].length statement outside the condition statement. In face, every occurrence, the [array].length will be re-calculate For a two occurrences loop, the result will not be visible, but (for example) in a five thousand occurrence loop, you'll see the difference. It can be explained with a simple calculation :

```
// we assume that myArray.length is 5000
for (x = 0;x < myArray.length;x++){
// doing some stuff
}
```

"x = 0" is evaluated only one time, so it's only one operation.

"x < myArray.length" is evaluated 5000 times, so it is 10,000 operations (`myArray.length` is an operation and compare `myArray.length` with x, is another operation).

"x++" is evaluated 5000 times, so it's 5000 operations.

There is a total of 15 001 operation.

```
// we assume that myArray.length is 5000
for (x = 0, l = myArray.length; x < l; x++){
// doing some stuff
}
```

"x = 0" is evaluated only one time, so it's only one operation.

"l = myArray.length" is evaluated only one time, so it's only one operation.

"x < l" is evaluated 5000 times, so it is 5000 operations (l with x, is one operation).

"x++" is evaluated 5000 times, so it's 5000 operations.

There is a total of 10002 operation.

So, in order to optimize your for loop, you need to make code like this :

```
var s1 = "the original string";
var c;
var st = "";
var s2 = "";

for (i = 0, l = s1.length; i < l; i++) {
  c = rot13(s1.charAt(i));
  st += c;
  if (c == " ") {
    s2 += st;
    st = "";
  }
}
s2 += st;
```

Shell

You can play around with JavaScript in one of multiple shells, in interactive batch mode. This means that you can enter JavaScript one line at a time and have it immediately executed; and if you enter a statement that returns a value without assigning the value anywhere, the value is displayed.

For a **list of shells**, see the mozilla.org list referenced from Enternal links.

Standalone

Mozilla Firefox uses **SpiderMonkey** JavaScript engine, which is available as a standalone interactive shell for multiple platforms. You can download it from:

- https://archive.mozilla.org/pub/firefox/nightly/latest-mozilla-central/
  - Look for jsshell-*.zip

Unzip the file, and run "js" from the command line. A prompt appears:

```
js>
```

You can enter statements, one at a time:

```
js> function incr(i) { return i+1; }
js> incr(1)
2
js> function plus2(i) {
return i+2;
}
js> plus2(1)
3
js> incr
function incr(i) { return i+1; }
js> print ("1+1:"+incr(1))
1+1:2
js> console.log("Yep.") // Console is available
Yep.
```

Multi-line function definitions can be entered one line at a time, pressing enter after each line.

To run JavaScript snippets that use alert function--since they are intended for web browsers, you can define your own alert function:

```
js> function alert(message) { print ("Alert: "+message); }
```

From browser

You can have an interactive mode, entering JavaScript one line at a time and have it immediately executed, directly from your web browser.

In many versions of Firefox, press Control + Shift + K to get a web console window. At the bottom of the console window, there is a separate one-line field into which you can enter JavaScript lines and have them run by pressing Enter. Even multi-line function definitions can be entered, but not by pressing Enter but rather Shift + Enter and pressing Enter only after the whole definition was entered.

External links

- Introduction to the JavaScript shell (https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Introduction_to_the_JavaScript_shell), developer.mozilla.org
- JavaScript shells (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Shells), developer.mozilla.org
- JavaScript Engine Speeds (http://ejohn.org/blog/javascript-engine-speeds/), ejohn.org
- JavaScript interactive shell with completion (http://stackoverflow.com/questions/41207/javascript-interactive-shell-with-completion), stackoverflow.com

Forms

Many users are familiar with filling out forms on a web page and then hitting the "submit" button. There are at least two ways JavaScript can improve this process:

- JavaScript can be used to check the data before it is sent to the server.
  - JavaScript can pre-validate the data, to catch common errors and suggest improvements immediately while the user is filling out the form, before the user clicks the "submit" button.
  - JavaScript can take text typed into a text area and pre-render it in a separate area of the page, so people can see how it will be rendered and formatted before clicking the "preview" button.[1][2][3]
  - JavaScript can give a live wordcount or character count of text typed into a text area.[4][5][6][7]
- Sometimes a web site does a "online" calculation involving only a small amount of data and returns the result to the user. In this case, JavaScript can intercept the "submit" button, do the entire calculation locally in the browser. The user gets the results more or less immediately, rather than waiting for the data he typed in to be sent to the server, waiting for the server to get around to processing that data, and waiting for the data to come back from the server.

Many people recommend keeping all the content accessible to people with disabilities and to people who have JavaScript turned off. One way to do that is to start with standard HTML forms, and then add unobtrusive JavaScript to improve the user experience to people who have JavaScript turned on. The system should degrade gracefully, keeping all the content available (and validating user data, if necessary), whether or not any or all of the JavaScript runs successfully.

Further reading

- JavaScript/Working With Files mentions one use of HTML forms
- HyperText Markup Language/Forms explains how to write "plain" forms without JavaScript.

References

1. "How can I display the HTML content of a Text Area within a div as HTML content and not text?" (http://stackoverflow.com/questions/1323812/how-can-you-render-html-as-it-is-typed-into-a-textarea)
2. Andrey Fedoseev. "jQuery plugin to add realtime preview panel to text areas, similar to StackOverflow edit interface" (http://andreyfedoseev.name/en/blog/post/34/jquery-textareapreview)
3. Guillaume DE LA RUE. "A Markdown live editor in JS" (http://www.js2node.com/markdown/a-markdown-live-editor-in-js).
4. Sacha Schmid. JavaScript word count (http://radlikewhoa.github.io/Countable/)
5. Drew Schrauf. "JavaScript Wordcount That Works" (http://drewschrauf.com/blog/2012/06/13/javascript-wordcount-that-works/).
6. "JavaScript word-count for any given DOM element" (http://stackoverflow.com/questions/765419/javascript-word-count-for-any-given-dom-element)

7. Jake Rocheleau. "Building a Live Text area Character Count Limit with CSS3 and jQuery" (http://spyrestudios.com/building-a-live-textarea-character-count-limit-with-css3-and-jquery/)

Bookmarklets

Bookmarklets are one line scripts stored in the URL field of a bookmark. Bookmarklets have been around for a long time so they will work in older browsers.

JavaScript URI scheme

You should be familiar with URL that start with schemes like `http` and `ftp`, e.g. http://en.wikibooks.org/. There is also the `JavaScript` scheme, which is used to start every bookmarklet.

```
JavaScript:alert('Hello, World!');
```

Example uses

Media controls

The values in these examples can be adapted as desired. One may replace `video` with `audio` where applicable, meaning where an `<audio>` tag is embedded.

Loop the video

```
javascript:document.getElementsByTagName("video")[0].loop=1;
javascript:document.getElementsByTagName("video")[0].loop=true; // also works
```

Can be switched off using `0` or `false`.

Jump to ten minutes (using multiplication)

```
javascript:document.getElementsByTagName("video")[0].currentTime=60*10;
```

Jump forward by one minute (sixty seconds)

```
javascript:document.getElementsByTagName("video")[0].currentTime+=60;
```

Jump back by half a minute (using division)

```
javascript:document.getElementsByTagName("video")[0].currentTime-=60/2;
```

Get duration of a video on the page in console

```
javascript:document.getElementsByTagName("video")[0].duration
```

Alert the duration

```
javascript:alert('This video is '+document.getElementsByTagName("video")[0].duration+'
seconds long.')
```

Alert the playback time

```
javascript:alert('The current position of the video is at
'+document.getElementsByTagName("video")[0].currentTime+' seconds.')
```

Set audio volume to 50%

```
javascript:document.getElementsByTagName("video")[0].volume=50/100
```

Mute audio

```
javascript:document.getElementsByTagName("video")[0].muted=1 // "true" works as well
```

Unmute using `0` or `false`.

Double the playback speed

```
javascript:document.getElementsByTagName("video")[0].playbackRate=2
```

Ask for playback speed

```
javascript:document.getElementsByTagName("video")[0].playbackRate= parseFloat( prompt("How
fast should it play?") );
```

`parseFloat` is necessary to prevent setting the value to zero if the dialogue window is closed without user input.

Ask for playback position in seconds

```
javascript:document.getElementsByTagName("video")[0].currentTime=parseFloat( prompt("Jump to
playback position in seconds:") );
```

Ask for playback position in minutes

```
javascript:document.getElementsByTagName("video")[0].currentTime=60*parseFloat( prompt("Jump
to playback position in minutes:") );
```

Ask for playback position in percentage (0 to 100)

```
javascript:document.getElementsByTagName("video")
[0].currentTime=document.getElementsByTagName("video")[0].duration/100*parseFloat(
prompt("Jump to playback position in percents:") );
```

Using multiple lines of code

Since you cannot have line breaks in bookmarklets you must use a semicolon at the end of each code statement instead.

```
JavaScript:name=prompt('What is your name?'); alert('Hello, ' + name);
```

The `JavaScript` Protocol in Links

The `JavaScript` protocol can be used in links. This may be considered bad practice, as it prevents access for or confuses users who have disabled JavaScript. See Best Practices.

```
<a href="JavaScript:document.bgColor='#0000FF'">blue background</a>
```

Examples

A large quantity of links may be found on bookmarklets.com (http://www.bookmarklets.com/), which show a variety of features that can be performed within JavaScript.

Working with files

With pure HTML4 and pure JavaScript, there's really only one thing you can do with the users files:

The server sends a Web page that includes a form something like this:[1]

```
<form action="/upload_handler" method="post">
  <input type="file" />
</form>
```

and then, the browser allows the user to select one file, and the browser uploads it -- without any JavaScript on the client being able to see any of that data or cancel the transmission or even show a progress bar.

If you want JavaScript to know anything about the file before it is transmitted (for example, to immediately cancel the transmission of a large file rather than wait an hour for the file to be transmitted, *then* tell the user "File too large"; or to show a progress bar), you'll have to use something other than pure JavaScript on pure HTML4.

Some popular options are:[2][3][4][5]

- use a modern Web browser that supports the HTML5 File API.
- use Flash (perhaps a tiny flash utility like Gmail uses to draw a little progress bar)
- use a Java applet
- use an ActiveX control

References

1. RFC 1867, RFC 2388
2. Getting upload file size before upload (http://stackoverflow.com/questions/4190934/getting-upload-file-size-before-upload)
3. Is it possible to use Ajax to do file upload? (http://stackoverflow.com/questions/543926/is-it-possible-to-use-ajax-to-do-file-upload)
4. File API: W3C Last Call Working Draft 12 September 2013 (http://www.w3.org/TR/FileAPI/)
5. Reading files in JavaScript using the File APIs (http://www.html5rocks.com/en/tutorials/file/dndfiles/)

Handling XML

Simple function to open an XML file

This function first tries for Microsoft Internet Explorer, then for Firefox and others:

```
function loadXMLDoc(xmlfilename) {
  var event = new Error();
  // Internet Explorer
  try {
    var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
  } catch(event) {
    // Firefox, Mozilla, Opera, others
      try {
      xmlDoc = document.implementation.createDocument("","",null);
    } catch(event) {
      throw(event.message);
    }
  }

  try {
    xmlDoc.async = false;

    xmlDoc.load(xmlfilename);
    return(xmlDoc);
  } catch(event) {
    throw(event.message);
  }
  return(null);
}
```

Usage

```
var objXML = loadXMLDoc("filename.xml");
var oNodes = objXML.getElementsByTagName("AnyTagYouWish");
```

Now you can do any DOM operations on oNodes.

XML modifications can't be saved in JavaScript, as this is clientside…

Handling JSON

Native JSON

Modern JSON Handling

Handling JSON may require adding a supporting library, which creates the global JSON object. This object is present natively only in new browsers (e.g. FF 3.5, IE8). Such a library can be found here (http://www.json.org/js.html):

```
//Parsing JSON:
var myObject = JSON.parse(myJSONtext)

//Creating JSON:
var myJSONText = JSON.stringify(myObject);
```

Old way

In old browsers you could use the following syntax, but this raises issues of security, such as XSS.

```
var myObject = eval("(" + myJSONtext + ")")
```

JSONP

Given browser restrictions on cross-domain Ajax (allowed only by configuration in some earlier browsers, by non-standard means in IE8, and with server headers in HTML5), one way to circumvent such restrictions (while still requiring some server-side script coordination) is for sites to insert an HTML script tag dynamically into their code, whereby the cross-domain script they target (typically) supplies JSON, but wrapped inside a function call (the function name being supplied according to the value of a "callback" parameter supplied by the requestor) or some other executable code.

In PHP, one might serve such JSONP in as simple a fashion as this:

```
<?php
if (isset($_GET['callback'])) {
  header('Content-Type: application/javascript');
  $our_site_data = ... // Set to an array or object containing data to be supplied for use at
other sites
  print $_GET['callback'] . '(' . json_encode($our_site_data) . ')';
}
?>
```

jQuery and other frameworks have their own means of generating JSONP requests, but we'll use the following custom code.

Note: It is important to bear in mind that the following code should not be used, if the targeted site or the data supplied by the target site, may come from a non-trustworthy source, since it is possible for such scripts to run with the privileges of the using site (e.g., to read user cookies and pass them

on to another site) and thereby execute a Cross-site scripting attack.

```
<script>
var JSONP = function(global) { // Introduces only one global
  // MIT Style license, adapted from http://devpro.it/code/209.html
  function JSONP(uri, callback) {
    function JSONPResponse() {
      // Reduce memory by deleting callbacks when finished
      try { delete JSONP[src] } catch(e) { JSONP[src] = null; }
      documentElement.removeChild(script);
      // Execute the user's callback with the arguments supplied by the server's JSONP call
      if (typeof callback === 'string') { // Assumes only one return argument and that it is
an HTML string
        document.getElementById(callback).innerHTML = arguments[0];
      } else {
        callback.apply(this, arguments);
      }
    }
    // Ensure a unique callback
    var src = '_' + id++,
    script = document.createElement("script");
    // Add our callback as a property of this JSONP
    // function to avoid introducing more globals
    JSONP[src] = JSONPResponse;
    // We include "callback", as it is typically used in
    // JSONP (e.g., on Wikibooks' API) to specify the callback
    documentElement.insertBefore(
      script,
      documentElement.lastChild
    ).src = uri + (uri.indexOf('?') === -1 ? '?' : '&') + "callback=JSONP." + src;
  }
  var id = 0, documentElement = document.documentElement;
  return JSONP;
}(this);

// Get the parsed HTML of this page you are reading now
// using the Mediawiki API (See http://en.wikibooks.org/w/api.php
// for Wikibooks, but it also applies at other Mediawiki wikis) that
// allows for such cross-domain calls
JSONP('http://en.wikibooks.org/w/api.php?
action=parse&format=json&page=JavaScript/Handling_JSON',
  function (data) {
    alert(data.parse.text['*']);
  }
);
</script>
```

More information

- Using native JSON in Firefox (https://developer.mozilla.org/En/Using_native_JSON)
- Using native JSON in IE8 (http://blogs.msdn.com/ie/archive/2008/09/10/native-json-in-ie8.aspx)
- Web Application Security Guide/XML, JSON and general API security

CS Communication

In many cases, the communication between clients and servers is programmed in *JavaScript*. Those JS-scripts use core and extended aspects of the language, especially its asynchronous features. To realize the communication, there is no need to introduce any special or additional features to the language itself.

But there are some terms, libraries, and APIs that are special to that communication, namely: the term *Ajax*, the *XMLHttpRequest* object, libraries like *jQuery* or *Axios*, and the *Fetch API*. Because of their importance, there are separate Wikibooks and Wiki pages that describe their behavior and application. The Wikibook on hand gives only a survey about their significance, short summaries, and links to appropriate pages for further readings.

XMLHttpRequest

The dominating protocol for client/server communication is http(s) (https://developer.mozilla.org/en-US/docs/Web/HTTP). It offers commands to read data from a server (GET) and to change the server's data (POST, PUT, PATCH, DELETE). Such commands are transferred within an object called *XMLHttpRequest*. It also contains the content (data) for both directions: to the server and from the server. Nowadays, the data is formatted chiefly in JSON - despite its name ***XML**HttpRequest*, which stands for its originally used XML format.

How to work with the *XMLHttpRequest* object is described in:

- MDN: XMLHttpRequest (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)
- Wikipedia: XMLHttpRequest
- W3Schools: XMLHttpRequest (https://www.w3schools.com/xml/xml_http.asp)

The response (data) returned by *XMLHttpRequest* may contain HTML fragments or other information that is used to create HTML fragments locally, e.g., data from a database that shall be shown in an HTML table. Therefore it is - among other things - a cornerstone to realize single-page applications (SPA).

Directly working with *XMLHttpRequest* is not outdated but a legacy technique where you have to consider many details. Libraries based on *XMLHttpRequest* and other APIs can make your work easier.

Ajax

The asynchronous behavior of the HTTP protocol is so important that it found its way into one of the central terms: "**Asynchronous JavaScript and XML**, or **Ajax**, is not a technology in itself, but rather 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." [1].

You can find short examples at the following sites:

- MDN: Ajax Introduction (https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started)
- Wikipedia: Ajax
- Wikibook: Programming Ajax

- W3Schools: Ajax examples (https://www.w3schools.com/js/js_ajax_php.asp)

Libraries

jQuery is a JavaScript library that simplifies commonly used activities like DOM traversal and manipulation, event handling, and client/server communication. You can find more information and examples at:

- jQuery Website (https://jquery.com/)
- Wikipedia: jQuery
- W3Schools: jQuery (https://www.w3schools.com/jquery/)

Axios is a JavaScript library. It realizes a client for browsers and for the Webserver *node.js*. Axios supports promises.

- Axios Website (https://axios-http.com/)
- Digital Ocean: Axios Tutorial (https://www.digitalocean.com/community/tutorials/js-axios-vanilla-js)

Fetch API

The *fetch API* supports the same features as the legacy *XMLHttpRequest* object and its interface. It is an entirely new implementation (with some minor differences from *XMLHttpRequest*) and is available in all modern browsers; there is no need for any additional library or framework. It is a member of the Web API family (https://developer.mozilla.org/en-US/docs/Web/API) (service worker, DOM API, cache API, ...).

Despite of its name *fetch* it does not only read data from a server. HTTP commands like PUT, POST, or DELETE are also supported.

You can find more information at:

- MDN: Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)
- W3Schools: Fetch API (https://www.w3schools.com/js/js_api_fetch.asp)

A complete example is given in the Wikibook on hand. Here is the structure of the script:

```
fetch('https://jsonplaceholder.typicode.com/users') // an example page
  .then((response) => { return response.json() })   // pick the json part
  .then((users) => { console.log(users) })          // show result
  .catch((err) => console.log('Some error occurred: ' + err.message));
```

The `fetch` command requests an example page. This page always returns an array of ten addresses in json format. The first `then` picks the json-part out of the result, and the second `then` shows it in the console. If an error occurs (network, typo in URL, ...), the `catch`-part executes and shows an error message.

References

1. MDN: Ajax (https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX)

Glossary

We try to keep consistency within the Wikibook when using technical terms. Therefore here is a short glossary.

object	An object is an associative array with key-value pairs. The pairs are called properties. All data types are derived from object - with the exception of primitive data types.
property	A key-value pair that is part of an object. The key part is a string or a symbol, the value part is a value of any type.
dot notation	The syntax to identify a property of an object using a dot: `myObject.propertyKey`
bracket notation	The syntax to identify a property of an object using brackets: `myObject["propertyKey"]`
curly braces notation	The syntax to express an object 'literaly' using {}: `const myObject = {age: 39}`
function	A function is a block of code introduced by the keyword `function`, an optional name, open parenthesis, optional parameters, and closing parenthesis. `function greeting(person) {` `  return "Hello " + person;` `};` The above function is a *named function*. If you omit the function name, we call it a *anonymous function*. In this case, there is an alternative syntax by using the *arrow syntax* =>. `function (person) { // no function name` `  return "Hello " + person;` `};` `// 'arrow' syntax. Here is only the definition of the function.` `// It is not called, hence not running.` `(person) => {` `  return "Hello " + person;` `};` `// or:` `(person) => {return "Hello " + person};` `// assign the definition to a variable to be able to call it.` `let x = (person) => {return "Hello " + person};` `// ... and execute the anonymous function by calling the variable` `// that points to it: x(...)` `alert(  x("Mike")  );` `// interesting:` `alert(x);` `// execute an anonymous function directly` `((person) => {` `  alert("Hello " + person);` `})("Mike");`
method	A method is a function stored as a key-value pair of an object. The key represents the method name, and the value the method body. A method can be defined and accessed with the following syntax: `let person = {` `  firstName: "Tarek",` `  city : "Kairo",` `  show : function() {` `  return this.firstName +` `  " lives in " +` `  this.city;` `  },`

	`` }; alert(person.show()); ```
callback function	A function that is passed as an argument to another function.
console	Every browser contains a window where internal information is shown. It's called the *console* and normally not opened. In most browsers, you can open the console via the function key `F12`.
item	A single value in an array. Same as 'element'.
element	A single value in an array. Same as 'item'.
parameter	When defining a function, variables can be declared in its signature, e.g.,: `function f(param1, param2)`. Those variables are called *parameters*.
argument	When calling a function, variables can be stated to be processed in the function, e.g.,: `f(arg1, arg2)`. Those variables are called *arguments*. The arguments replace the parameters which were used during the function definition.

Index

A

B

C

D

E

F

H

I

L

M

N

O

R

S

T

U

V

Links

Featured weblinks:

- JavaScript portal (http://developer.mozilla.org/en-US/docs/JavaScript) at developer.mozilla.org
- JavaScript Reference (http://developer.mozilla.org/en-US/docs/JavaScript/Reference) at developer.mozilla.org
- JavaScript Guide (http://developer.mozilla.org/en-US/docs/JavaScript/Guide) at developer.mozilla.org
- Gecko DOM Reference (http://mozilla.org/docs/dom/domref/dom_shortTOC.html) at developer.mozilla.org

---

- JavaScript Reference (http://msdn.microsoft.com/en-us/library/ie/yek4tbz0%28v=vs.94%29.aspx) at msdn.microsoft.com

---

- Wikipedia:JavaScript
- Wikipedia:ECMAScript
- Wikipedia:JavaScript engine

- Wikiversity: Topic:JavaScript
- Wikiversity: Advanced JavaScript

- JavaScript Tutorial (http://www.w3schools.com/js/) at w3schools.com
- JavaScript Reference (http://www.w3schools.com/jsref/default.asp) at w3schools.com
- About: Focus on JavaScript (http://javascript.about.com/) from Stephen Chapman at javascript.about.com

---

- ecmascript.org (http://www.ecmascript.org/)
- ecma-international.org (http://www.ecma-international.org/)
- http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Discussion forums, bulletin boards:

- HTML, CSS and JavaScript (http://www.coderanch.com/forums/f-20/HTML-CSS-JavaScript) at coderanch.com
- JavaScript Workshop forums (http://jsworkshop.com/bb/) at jsworkshop.com
- Forum: Client-side technologies (http://www.webxpertz.net/forums/forumdisplay.php/119-Client-side-technologies) at webxpertz.net

More Web sites:

- JavaScript tutorials (http://webreference.com/programming/javascript/) at webreference.com
- Videos (https://www.youtube.com/results?hl=en&q=crockford%20javascript) from w:Douglas Crockford on JavaScript (http://javascript.crockford.com/)

- JavaScript (http://www.epanorama.net/links/pc_programming.html#javascript) at epanorama.net
- JavaScript Tutorials (http://www.pickatutorial.com/tutorials/javascript_1.htm) at pickatutorial.com
- JavaScript Essentials (http://www.techotopia.com/index.php/JavaScript_Essentials) at techotopia.com - An online JavaScript book designed to provide Web developers with everything they need to know to create rich, interactive and dynamic Web pages using JavaScript.
- JavaScript Tutorials (http://www.yourhtmlsource.com/javascript/) at yourhtmlsource.com
- www.quirksmode.org (http://www.quirksmode.org) - over 150 useful pages for CSS and Javascript tips & cross-browser compatibility matrices.
- Wiki: I wanna Learn JavaScript (http://c2.com/cgi/wiki?IwannaLearnJavaScript) at c2.com - A list of links to Web resources on JavaScript
- Unobtrusive JavaScript (http://www.onlinetools.org/articles/unobtrusivejavascript/chapter1.html) at onlinetools.org - a guide on how to write JavaScript so that your site degrades gracefully (i.e., if the browser does not support or has turned off JavaScript, your site is still usable).

Useful software tools

A list of useful tools for JS programmers.

Editors / IDEs

- Adobe Brackets: Another browser-based editor by Adobe
- Eclipse: The Eclipse IDE includes an editor and debugger for JS
- Notepad++ (https://notepad-plus-plus.org/): A great tool for editing any kind of code. It includes syntax highlighting for many programming languages.
- Programmers' Notepad (https://www.pnotepad.org/): A general tool for programming many languages.
- Scripted (https://github.com/scripted-editor/scripted): An open-source browser-based editor by Spring Source
- Sublime Text: One of the most used editors for HTML/CSS/JS editing
- Web Storm or IntelliJ IDEA: both IDEs include an editor and debugger for JS, IDEA also includes a Java development platform

Engines and other tools

- JSLint: static code analysis for JS
- jq (http://stedolan.github.io/jq/) - '' 'jq' is like sed for JSON data ''
- List of ECMAScript engines

Best practices

This chapter will describe current JavaScript community standards and best practices that every programmer should know.

document.write

This is deprecated. Use `innerHTML` or DOM manipulation methods instead.

In XHTML, `document.write` does not work, but you can achieve the same effects with DOM manipulation methods[9] (http://w3.org/MarkUp/2004/xhtml-faq#docwrite).

JavaScript protocol

Try to avoid links that exist solely for executing JavaScript code.

```
<a href="javascript:resumeFancyVersion()">See my résumé!</a>
```

Instead consider:

```
<a href="resumePlainVersion.html" onclick="return !resumeFancyVersion()">See my résumé!</a>
```

Users with JavaScript enabled will be provided with the JavaScript-embellished version of content (perhaps using DHTML), whereas those without will be redirected to a XHTML page that provides it statically. This is more accessible than having an <a> element lacking a normal href attribute value. First, it uses proper language semantics. Second, it provides access to your content for users without JavaScript. Third, it detects whether the function execution succeeded and redirects JS-enabled readers too in case of a failure.

The onclick expression evaluates to a Boolean value. If the function succeeds to perform the desired effect and returns true, the onclick will return a failure and hyperlink not execute. When the function fails for whatever reason, the false, null or undefined value will evaluate to true and not prevent the link from being executed. Alternatively, if you do not wish to provide a static equivalent, you may embed the onclick attribute within an element with less demanding semantics:

```
<strong onclick="resumeFancyVersion()">See my résumé!</strong>
```

Thus no user agent will be confused upon reading a reduced <a> element.

Email validation

Many people use JavaScript functions to immediately catch the most common sorts of errors in form entry. For example, some web forms ask people to enter the same thing twice. Other web forms ask people to enter an email address. Then they do a quick check to see if what was entered looks at least vaguely like an email address:

```
function isValidEmail(string) {
  // These comments use the following terms from RFC2822:
  // local-part, domain, domain-literal and dot-atom.
  // Does the address contain a local-part followed an @ followed by a domain?
  // Note the use of lastIndexOf to find the last @ in the address
  // since a valid email address may have a quoted @ in the local-part.
  // Does the domain name have at least two parts, i.e. at least one dot,
  // after the @? If not, is it a domain-literal?
  // This will accept some invalid email addresses
  // BUT it doesn't reject valid ones.
  var atSym = string.lastIndexOf("@");
  if (atSym < 1) { return false; } // no local-part
  if (atSym == string.length - 1) { return false; } // no domain
  if (atSym > 64) { return false; } // there may only be 64 octets in the local-part
  if (string.length - atSym > 255) { return false; } // there may only be 255 octets in the
domain

  // Is the domain plausible?
  var lastDot = string.lastIndexOf(".");
  // Check if it is a dot-atom such as example.com
  if (lastDot > atSym + 1 && lastDot < string.length - 1) { return true; }
  // Check if could be a domain-literal.
  if (string.charAt(atSym + 1) == '[' && string.charAt(string.length - 1) == ']') { return
true; }
  return false;
}
```

Unfortunately, some other "email validation" JavaScript functions reject perfectly valid email addresses. For example, some incorrectly reject (http://www.javascriptsource.com/forms/email-validation---basic.html) valid addresses that include "+" signs.

The original email address syntax (RFC 821) did allow "+" signs. RFC 822, published in the same month (August 1982), also allowed them. The current version of the syntax is given in RFC2821 (http://www.faqs.org/rfcs/rfc2821.html)/RFC2822 (http://www.faqs.org/rfcs/rfc2822.html). Section 3 of RFC3696 (http://www.faqs.org/rfcs/rfc3696.html) gives useful examples of unusual valid email addresses.

After validation, many JavaScript programmers encode the email address using `encodeURIComponent()` to work-around certain client-side languages that can't seem to handle plus signs properly.[1][2]

The complexity of the quoting rules used for email addresses makes it impractical to test the local-part of an address or a domain literal completely. Given that there isn't necessarily a real mailbox corresponding to a valid local-part how much extra download time is worth spending on a complex validation script?

Examples valid according to RFC2822

- `me@example.com`
- `a.nonymous@example.com`
- `name+tag@example.com`
- `a.name+tag@example.com`
- `me.example@com`
- `"spaces must be quoted"@example.com`
- `!#$%&'*+-/=.?^_`{|}~@[1.0.0.127]`
- `!#$%&'*+-/=.?^_`{|}~@[IPv6:0123:4567:89AB:CDEF:0123:4567:89AB:CDEF]`

- `me(this is a comment)@example.com` – comments are discouraged but not prohibited by RFC2822.

Examples invalid according to RFC2822s

- `me@`
- `@example.com`
- `me.@example.com`
- `.me@example.com`
- `me@example..com`
- `me\@example.com`
- `spaces\ must\ be\ within\ quotes\ even\ when\ escaped@example.com`
- `a\@mustbeinquotes@example.com`

Test page

COMMENT: This code has been incorrectly designed to reject certain emails that are actually valid. If changes to valid and invalid emails are accepted, the following code should also be reviewed.

The following test page can be used to test an email validation function. Save the three files in the same directory and then open *validEmail.htm* in a web browser.

validEmail.js

```
function isValidEmail(string) {
  // These comments use the following terms from RFC2822:
  // local-part, domain, domain-literal and dot-atom.
  // Does the address contain a local-part followed an @ followed by a domain?
  // Note the use of lastIndexOf to find the last @ in the address
  // since a valid email address may have a quoted @ in the local-part.
  // Does the domain name have at least two parts, i.e. at least one dot,
  // after the @? If not, is it a domain-literal?
  // This will accept some invalid email addresses
  // BUT it doesn't reject valid ones.
  var atSym = string.lastIndexOf("@");
  if (atSym < 1) { return false; } // no local-part
  if (atSym == string.length - 1) { return false; } // no domain
  if (atSym > 64) { return false; } // there may only be 64 octets in the local-part
  if (string.length - atSym > 255) { return false; } // there may only be 255 octets in the
domain

  // Is the domain plausible?
  var lastDot = string.lastIndexOf(".");
  // Check if it is a dot-atom such as example.com
  if (lastDot > atSym + 1 && lastDot < string.length - 1) { return true; }
  //  Check if could be a domain-literal.
  if (string.charAt(atSym + 1) == '[' && string.charAt(string.length - 1) == ']') { return
true; }
  return false;
}

function testIsValidEmail(string) {
  alert("'" + string + "' is " + (isValidEmail(string) ? "" : "NOT ") + " a valid email
address.");
}

function checkSamples() {
  var validAddress = [
    'me@example.com',
    'a.nonymous@example.com',
```

```
    'name+tag@example.com',
    'name\\@tag@example.com',
    'spaces\\ are\\ allowed@example.com',
    '"spaces may be quoted"@example.com',
    '!#$%&\'*+-/=.?^_`{|}~@[1.0.0.127]',
    '!#$%&\'*+-/=.?^_`{|}~@[IPv6:0123:4567:89AB:CDEF:0123:4567:89AB:CDEF]',
    'me(this is a comment)@example.com'
  ];
  var invalidAddress = [
    'me@',
    '@example.com',
    'me.@example.com',
    '.me@example.com',
    'me@example..com',
    'me.example@com',
    'me\\@example.com'
  ];

  var results = new StringBuffer();

  var handlesValidAddressesCorrectly = true;
  results.append('<table border="1">');
  results.append('<caption>Does the function accept all the valid sample addresses?
</caption>');
  results.append('<tr><th>Valid address</th><th>Function returns</th></tr>');
  for (var i = 0; i < validAddress.length; i++) {
    results.append('<tr><td>');
    results.append(validAddress[i]);
    results.append('</td><td>');
    if (isValidEmail(validAddress[i])) {
      results.append('<span class="good">true</span>');
    } else {
      handlesValidAddressesCorrectly = false;
      results.append('<strong class="fail">false</strong>');
    }
    results.append('</td></tr>');
  }
  results.append('</table>');

  var handlesInvalidAddressesCorrectly = true;
  results.append('<table border="1">');
  results.append('<caption>Does the function reject all the invalid sample addresses?
</caption>');
  results.append('<tr><th>Valid address</th><th>Function returns</th></tr>');
  for (var i = 0; i < invalidAddress.length; i++) {
    results.append('<tr><td>');
    results.append(invalidAddress[i]);
    results.append('</td><td>');
    if (!isValidEmail(invalidAddress[i])) {
      results.append('<span class="good">false</span>');
    } else {
      handlesInvalidAddressesCorrectly = false;
      results.append('<em class="warning">true</em>');
    }
    results.append('</td></tr>');
  }
  results.append('</table>');

  var conclusion;
  if (handlesValidAddressesCorrectly) {
    if (handlesInvalidAddressesCorrectly) {
      conclusion = '<p><strong class="good">The function works correctly with all the sample
addresses.</strong></p>';
    } else {
      conclusion = '<p><em class="warning">The function incorrectly accepts some invalid
addresses.</em></p>';
    }
  } else {
    conclusion = '<p><strong class="fail">The function incorrectly rejects some valid
addresses.</strong></p>';
  }

  document.getElementById('testResults').innerHTML = conclusion + results.toString();
}
```

```
function StringBuffer() {
  this.buffer = "";
}

StringBuffer.prototype.append = function(string) {
  this.buffer += string;
  return this;
};

StringBuffer.prototype.toString = function() {
  return this.buffer;
};
```

validEmail.css

```
body {
  background-color: #fff;
  color: #000
}

table {
  margin-bottom: 2em
}

caption {
  margin-top: 2em
}

.good {
  background-color: #0f0;
  color: #000
}

.warning {
  background-color: #fc9;
  color: #000
}

.fail {
  background-color: #f00;
  color: #fff
}
```

validEmail.htm

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Valid email test</title>
    <link rel="stylesheet" href="validEmail.css">
    <script src="validEmail.js"></script>
  </head>
  <body onload="checkSamples()">
    <h1>Unit test for email address validation functions</h1>
    <h2>Interactive test</h2>
    <form action="#">
      <fieldset>
        <legend>Email address</legend>
        <label for="emailAddress">Email</label>
        <input type="text" size="40" value="" name="email" id="emailAddress">
        <input type="button" name="validate" value="Check address"
          onclick="testIsValidEmail(this.form.email.value)">
      </fieldset>
    </form>
    <h2>Selected samples</h2>
    <p>This section shows the results of testing the function with sample strings.
      The sample includes both valid strings and invalid strings
      according to <a href="http://www.faqs.org/rfcs/rfc2822.html">RFC2822</a>.
    </p>
    <div id="testResults">You need to enable JavaScript for this unit test to work.</div>
```

```
  </body>
</html>
```

use strict

Many JavaScript programmers recommend enabling ECMAScript 5's strict mode by putting the exact statement `"use strict";` before any other statements:[3][4][5]

```
"use strict";
function …
```

For further reading

JavaScript best practices:

- Coding Cookbook/Validate Email Address
- "Six JavaScript features we do not need any longer" by Christian Heilmann (http://wait-till-i.com/index.php?p=104).
- source code for Apple's recommended JavaScript validation functions: checkEmail(), etc. (http://developer.apple.com/internet/webcontent/validation.html)
- JavaScript form validation - doing it right (http://www.xs4all.nl/~sbpoley/webmatters/formval.html)
- "FORM submission and the ENTER key?" (http://ppewww.physics.gla.ac.uk/~flavell/www/formquestion.html) discusses forms that submit when you press Enter; forms that don't submit when you press Enter; and how to make a form work the other way.
- "JavaScript Best Practices" by Matt Kruse (http://www.mattkruse.com/javascript/bestpractices/)
- "Comparing E-mail Address Validating Regular Expressions" (http://fightingforalostcause.net/misc/2006/compare-email-regex.php) has a list of valid and invalid email addresses and a variety of regular expressions and how well each one works on that list.

Best practices in other languages

- Computer programming/Standards and Best Practices

References

1. Jan Wolter. "JavaScript Madness: Query String Parsing" (http://unixpapa.com/js/querystring.html) 2011.
2. PHP bug #39078: Plus sign in URL arg received as space (https://bugs.php.net/bug.php?id=39078).
3. Nicholas C. Zakas. "It's time to start using JavaScript strict mode" (http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/). 2012.
4. "What does "use strict" do in JavaScript, and what is the reasoning behind it?" (http://stackoverflow.com/questions/1335851/what-does-use-strict-do-in-javascript-and-what-is-the-reasoning-behind-it)
5. Mozilla Developer Network: "Strict mode" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode).