05 – JavaScript

Web Technology Project (International Computer Science)
Summer semester 2025
Prof. Dr. Felix Schwägerl

Key facts

- JavaScript (JS) makes static web pages dynamic.
  - It adds (non-default) *behavior* to HTML elements and may interact with CSS, too.
  - Frequent use case: React to *browser events* and *manipulating* the *DOM*, updating the page dynamically so that it *reflects user interaction*
- Lightweight, single-threaded, *interpreted* language executed by browsers
  - Not considered here: server-side JS
- JavaScript has a lot of differences to Java (despite the name similarity)
  - JS is dynamically typed and object-based but not class-centric (“duck-typed”)
  - *Functions* play a more important role; functions are first-class *values*.
- JS is an evolving language. There are many bad examples on web pages
  - ChatGPT has been trained with outdated examples, which is reflected in its responses.
- Recommended documentation: https://developer.mozilla.org/docs/Web/JavaScript

History of JavaScript

1995: JavaScript was created by Brendan Eich at Netscape in just 10 days, originally called Mocha, later renamed to LiveScript and then JavaScript.

1996: Microsoft released JScript, their own version of JavaScript, creating early browser compatibility issues.

1997: JavaScript was standardized as *ECMAScript (ES1)* by ECMA International to ensure cross-browser compatibility.

1999–2009: Slow evolution with *ES3* (1999) being the dominant standard; *ES4* was abandoned due to complexity.

2009: *ES5* introduced significant improvements like strict mode, JSON support, and array methods.

2015: *ES6* (also called ES2015) was a major update introducing let, const, arrow functions, promises, classes, and modules.

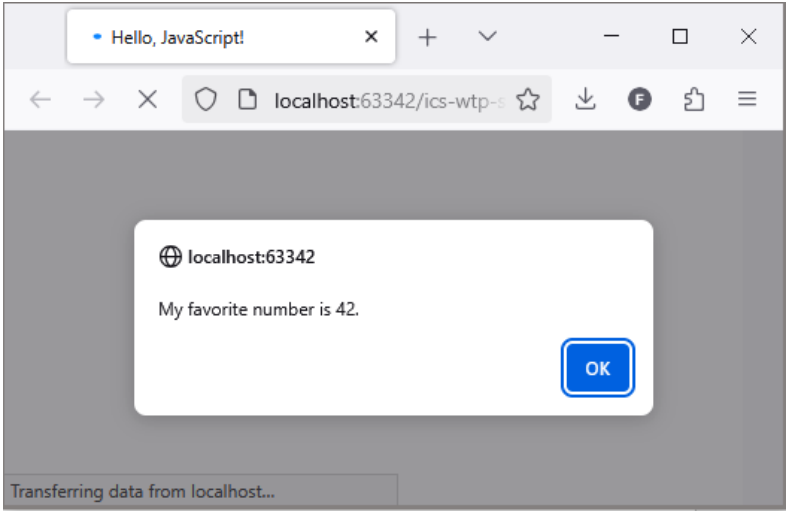
2016–present: Annual updates (*ES7, ES8*, etc.) continue to enhance language features and performance.

Today: JavaScript is a cornerstone of web development, powering interactive websites, mobile apps, and even servers (via Node.js).

[ChatGPT1]

Hello, JavaScript!

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, JavaScript!</title>
    <script>
        let x = 17 * 3 - 9;
        let s = "My favorite number is " + x + ".";
        window.alert(s);
    </script>
</head>
<body>
    <h1>Hello, JavaScript!</h1>
</body>
</html>
```



Adding JavaScript to HTML documents

Internal (not recommended)

Physically connected to current document

```
<html lang="en">
<head> …
    <script>
        let x = 17 * 3 - 9;
        let s = "My favorite number is " + x + ".";
        console.log(s);
    </script>
</head>
<body> <h1> … <h1> </body>
</html>
```

External (recommended)

Makes sure JS is executed after loading page finished

May be included from multiple documents

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, JavaScript! (external script)</title>
    <script src="hello-external.js" defer></script>
</head>
<body> <h1> … </h1> </body>
</html>
```

```
let x = 17 * 3 - 9;
let s = "My favorite number is " + x + ".";
console.log(s);
```

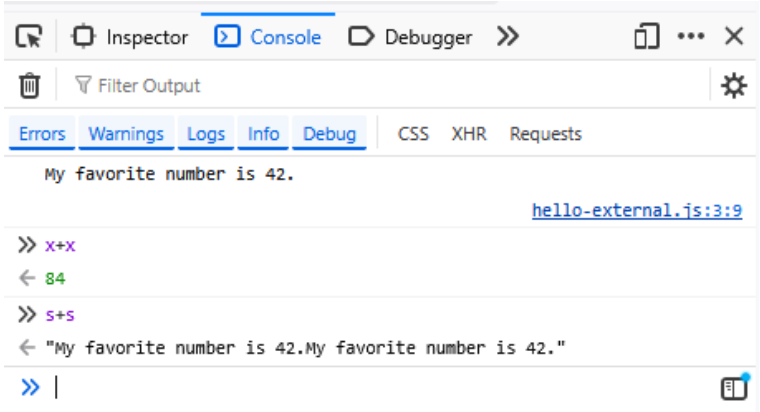
What does console.log actually do?

- JavaScript contains an internal logging facility, which is primarily used for debugging.
- We may view console messages using the Browser tools (e.g., Alt+Shift+I in Firefox)
- The browser console is also a REPL (read-eval-print loop)
  - We may input JavaScript expressions and inspect the results.

```
let x = 17 * 3 - 9;
let s = "My favorite number is "
            + x + ".";
console.log(s);
```

Output produced by console.log

Inputs and outputs produced by REPL



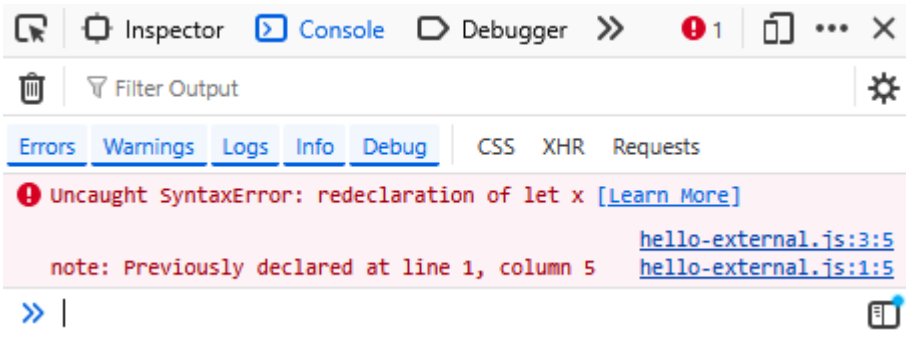
What happens if our JavaScript contains errors?

- No compilation
  - Scripts are executed (and validated) on the fly by browsers
- Syntax errors are reported by the browser console, too.
  - When developing JavaScript, you should always keep the browser console open!
- The script terminates in the case of a syntax error.

```
let x = 17 * 3 - 9;
let s = "My favorite number is " + x + ".";
let x = 23;
console.log(s);
```

Not executed (as script terminated before)

Syntax error: redeclaration of an existing variable x



In JavaScript, the type of variables is inferred and may change.

- `number` (includes integer and decimal)
- `boolean`
- `string`
- Objects and arrays (→)
- `null` and `undefined`

```
let n1 = 42;
let n2 = n1 / 2;
let b1 = true;
let b2 = n2 + n2 == n1;
let s1 = "10";
let s2 = s1 + n1;
let s3 = "23" - s2;

// this variable changes its type:
let x1 = 23;
x1 = "Hello";
```

Comments like in Java: // and /*...*/

number

boolean

string

number (!)

number

type changes to string

```
» n1
← 42
» n2
← 21
» b1
← true
» b2
← true
» s1
← "10"
» s2
← "1042"
» s3
← -1019

» x1
← "Hello"
```

`null` vs. `undefined`

Every variable may have values `null` or `undefined`.

- `null` is a value representing the absence of a value, similar to `null` in Java.
- `undefined` is the value given to a variable not having been assigned a concrete value.
  - However, you may also manually set a variable's value to `undefined`.
- Depending on the type of comparison (→), `null` and `undefined` are considered different from each other.

implicitly undefined

```
let x;
console.log(x);
x = null;
console.log(x);
x = undefined;
console.log(x);
```

```
undefined                    variables.js:31:9
null                         variables.js:33:9
undefined                    variables.js:35:9
```

JavaScript distinguishes non-strict from strict equality.

- Non-strict: == and !=
  - The interpreter tries to perform type conversions before doing the comparison.
  - The underyling *coercion* is more often considered a bug than a feature.
- Strict: === and !==
  - Variables can only be equal if their type is equal.
  - Type conversions are prevented, leading to “less surprising” comparisons”

```
console.log("3" == 3);
console.log(1 == true);
console.log(null == 0);
console.log(null == undefined);
console.log("" == undefined);
```

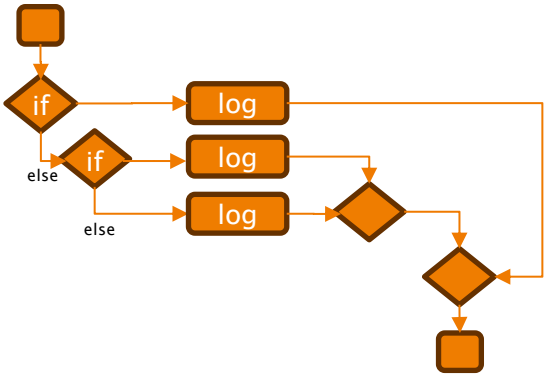
```
true                    variables.js:15:9
true                    variables.js:16:9
false                   variables.js:17:9
true                    variables.js:18:9
false                   variables.js:19:9
```

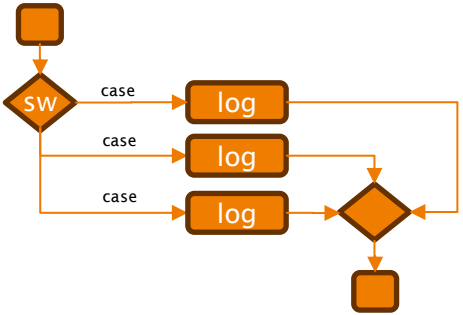
```
console.log("3" === 3);
console.log(1 === true);
console.log(null === 0);
console.log(null === undefined);
console.log("" === undefined);
```

```
false                   variables.js:22:9
false                   variables.js:23:9
false                   variables.js:24:9
false                   variables.js:25:9
false                   variables.js:26:9
```

if/else and switch/case define conditions.

```
let x = 23;
let y = 42;

if (x === y) {
    console.log("x and y are equal");
} else if (x > y) {
    console.log("x is greater than y");
} else {
    console.log("y is greater than x");
}

switch (x) {
    case 23: console.log("x is 23"); break;
    case 42: console.log("x is 42"); break;
    default: console.log("x is something else");
}
```






```
y is greater than x                                conditions.js:9:13
x is 23                                            conditions.js:13:22
```

`for` and `while` define loop statements.

```
let x = 9;
let y = 24;

let facX = 1;
for (let i = 1; i <= x; i++) {
    facX *= i;
}
console.log("The factorial of " + x + " is " + facX)
```

```
let a = x;
let b = y;
while (b !== 0) {
    let t = b;
    b = a % b;
    a = t;
}
let gcdXY = a;
console.log("The GCD of " + x + " and " + y + " is " + gcdXY)
```

```
The factorial of 9 is 362880                    loops.js:8:9
The GCD of 9 and 24 is 3                        loops.js:18:9
```

Keyword `function` defines reusable units of behavior.

- Parameter and return types are not declared explicitly.
- `console.log` is an example of a function call (of a built-in object)
- Function calls can be nested.
- Functions may be declared in any order (*hoisting*)

```
function repeatMessage(message, times) {
    let result = "";
    for (let i = 0; i < times; i++) {
        result += message + "\n";
    }
    return result;
}

function printMessage(message, times) {
    console.log(repeatMessage(message, times));
}

printMessage("Hello", 3);
```

This parameter gets interpreted as string

… and this one as number

Function returns a string (not declared)

Nested function call: Result of one function call is passed to another function

Function call from the main script

A higher-order function is a function that receives and/or produces another function as parameter or result.

- Higher-order parameters look like regular parameters, but are called like functions in the body of the higher-order function.

```
function repeatNTimes(n, action) {
    for (let i = 0; i < n; i++) {
        action(i);
    }
}

function sayHello(n) {
    console.log("Saying hello for the " + n + " time");
}

repeatNTimes(7, sayHello);
```

This parameter gets interpreted as function that receives an int and produces nothing („void“)

Calling the passed function

Regular function that is used as parameter in the function call below.

Calling the higher-order function, passing an existing function as argument for the higher-order function parameter (action)

```
Saying hello for the 0 time    functions.js:25:13
Saying hello for the 1 time    functions.js:25:13
Saying hello for the 2 time    functions.js:25:13
Saying hello for the 3 time    functions.js:25:13
Saying hello for the 4 time    functions.js:25:13
Saying hello for the 5 time    functions.js:25:13
Saying hello for the 6 time    functions.js:25:13
```

Higher-order parameters can be passed as anonymous functions (using extended or lambda notation).

```
repeatNTimes(7, function(n) {
    console.log("*".repeat(n))
})
```

Instead of referencing a global function, an anonymous function is passed here

```
<empty string>
*
**
***
****
*****
******
```

Compact notation (arrow/lambda syntax) for passing functions to functions

```
repeatNTimes(5, n => console.log("#".repeat(n)));
```

```
<empty string>
#
##
###
####
```

If the parameters are not used by the function, they should be elided with ()

```
repeatNTimes(3, () => console.log("Cheers!"));
```

Compact logging (3x same output)

```
Cheers!  3  functions.js:36:31
```

Functions can also be *returned* by higher-order functions.

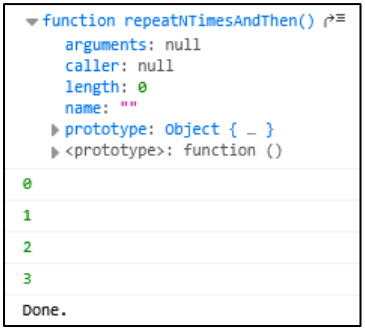
- The function below is a higher-order function for 3 reasons:
  - It defines the action higher-order parameter (like previous ones)
  - It takes a second higher-order parameter, a *callback* function called at the end.
  - It does not perform the action directly, but returns a function that allows the caller of the higher-order function to *defer* the method call.
- Callbacks and deferred functions are used often by JavaScript libraries.

```
function repeatNTimesAndThen(n, action, conclusion) {
    return function() {
        repeatNTimes(n, action);
        conclusion();
    }
}

let f = repeatNTimesAndThen(4,
        n => console.log(n),
        () => console.log("Done."));
console.log(f);
f();
```

Returning a function from a function

f stores the function without calling it

deferred function call



Scopes and constness: `var` vs. `let` vs. `const`

`var` defines function-scoped variables.

- Variable is visible in the entire function, even if it is declared in a nested block.
- Modern JavaScript should not use `var`.

`let` and `const` define block-scoped variables.

- Variable is visible in the current scope (function or nested block).
- `const` declares constants
  - (like `final` in Java)
  - Prefer `const` over `let` if possible!

```
var x = 3;
f();

function f() {
    var x = 4; // this is a different x
    x = 5; // modifies the inner x
    const w = x;

    if (x > 5) {
        var y = 7;
        let z = 8;
        const v = 42;
    }

    y++;   // ok: y is function-scoped
    z++;   // error: z is block-scoped
    w++;   // error: re-assignment to const
    x = v; // error: v is block-scoped
}
```

OK: y is function-scoped

Error: z is block-scoped

Error: re-assignment to const w

Error: v is block-scoped

JavaScript arrays are heterogeneous and resizable.

- Not being restricted to a type, they may contain a mixture of variables of different types.
  - Arrays of arrays (and arrays of objects →) are possible.
- Arrays are *non-associative* and *zero-indexed*.
- Important built-in operators: Index access (read/write), `length`, `slice`, `includes`

```
let a1 = ["Hello", 42, "World", [true, [1, 2, 3]]];
a1.push("xyz");

console.log(a1);
console.log(a1.length);
console.log(a1[0] + ", " + a1[2]);
console.log(a1[3][1])
let a2 = a1[3][1];
a2[0] = 0;
console.log(a2);
console.log(a1[3][1]);
let a3 = [...a2, 4, 5];
console.log(a3.slice(1, 3))
```

Add element to array

Current length of array

Multi-dimensional array access

Copies reference, not array: Modification affects both references

Copy operator allows to insert more elements to the copied array

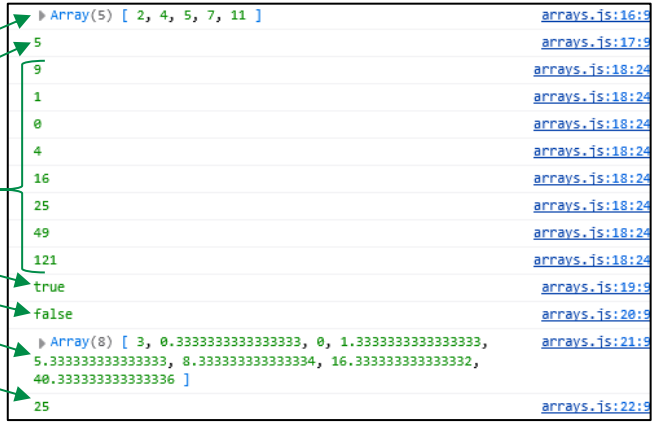
Creates a new sub-array [from, toExcl]

```
▶ Array(5) [ "Hello", 42, "World", (2) […], "xyz" ]   arrays.js:4:9
5                                                     arrays.js:5:9
Hello, World                                          arrays.js:6:9
▶ Array(3) [ 1, 2, 3 ]                                arrays.js:7:9
▶ Array(3) [ 0, 2, 3 ]                                arrays.js:10:9
▶ Array(3) [ 0, 2, 3 ]                                arrays.js:11:9
▶ Array [ 2, 3 ]                                      arrays.js:13:9
```

Arrays offer powerful higher-order functions.

- `filter`: Creates a new array restricted to elements matching the condition provided.
- `find`: Retrieves the first element matching the condition provided, or `undefined` if not exists.
- `forEach`: Applies a provided consumer function to every element of the array.
- `every`: Decides if a provided predicate applies to all elements of the array.
- `some`: Decides if a provided predicate applies to one or more elements of the array.
- `map`: Applies a transformer function to every element and creates a new array from the results.
- `reduce`: Reduces an array to a single value by repeatedly applying a binary operation.

```
let a = [-3, -1, 0, 2, 4, 5, 7, 11]
console.log(a.filter(x => x > 0));
console.log(a.find(x => x > 0 && x % 2 === 1))
a.forEach(x => console.log(x * x));
console.log(a.every(x => x > -4));
console.log(a.some(x => x > 10 && x % 2 === 0));
console.log(a.map(x => x * x / 3));
console.log(a.reduce((x, y) => x + y));
```



The for..of loop easily loops over array elements.

- More compact, but does not capture the index.

```
for (const x of a) {
    console.log("The square of „
            + x + " is " + x * x);
}
```

```
The square of -3 is 9                    arrays.js:25:13
The square of -1 is 1                    arrays.js:25:13
The square of 0 is 0                     arrays.js:25:13
The square of 2 is 4                     arrays.js:25:13
The square of 4 is 16                    arrays.js:25:13
The square of 5 is 25                    arrays.js:25:13
The square of 7 is 49                    arrays.js:25:13
The square of 11 is 121                  arrays.js:25:13
```

for..of also works for strings.

- The loop variable is bound to every character of the string.

```
for (const c of "Hello") {
    console.log(c);
}
```

```
H                                        arrays.js:29:13
e                                        arrays.js:29:13
l                                   2    arrays.js:29:13
o                                        arrays.js:29:13
```

for..in loops over *indexes* rather than elements (rarely needed)!

JavaScript emphasizes objects, not classes.

- JavaScript uses *duck typing*: Classes are implicitly created by objects having properties and/or functions of the same type.
  - "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."
- Many strict OOP concepts such as inheritance, interfaces, generics are not applicable to JavaScript due to its untyped / duck-typed nature.
  - JavaScript relied on *prototypes* (not considered here) to simulate typing and inheritance behavior.
  - *Classes* have been introduced in ES6 (not considered here)

How to define an object in JavaScript

- Basic structure: JSON (actually, JSON is derived from JavaScript)

```
const john = {
    firstNames: ["John", "F."],
    lastName: "Doe",
    fullName() {
        return this.firstNames.reduce(
           (s1, s2) => s1 + " " + s2) +
            " " + this.lastName;
    }
}
```

properties

method (function as property)

property access via this

```
Object { firstNames: (2) […], lastName: "Doe", fullName: fullName() ↱≡ }   objects.js:9:9
  firstNames: Array [ "John", "F." ]
  fullName: function fullName() ↱≡
  lastName: "Doe"
  <prototype>: Object { … }
John F. Doe                                                                objects.js:10:9
```

Object properties (also methods) can be dynamically queried, modified and removed.

```
console.log(john);
console.log(john.firstNames);
console.log(john["lastName"]);
console.log(john.fullName());
john.firstNames.push("W.")
console.log(john.fullName());
john.birthDate = Date.parse("1995-04-29")
john.ageYears = function(){
    return (Date.now() - this.birthDate)
    / MILLIS_PER_YEAR;
}
console.log(john.birthDate);
console.log(john.ageYears())
john.lastName = undefined;
console.log(john.fullName());
```

- Accessing a property with dot notation
- Property access with brackets (not usual)
- Method call
- Modification of a property
- Dynamic addition of a new property
- Dynamic addition of a new method (referencing existing and dynamically added properties via this)
- Call of a dynamically added method
- Removing a property (by setting it undefined)

```
▶ Object { firstNames: (2) […], lastName: "Doe", fullName: fullName() ↱≡ }   objects.js:11:9
▶ Array [ "John", "F." ]                                                     objects.js:12:9
Doe                                                                          objects.js:13:9
John F. Doe                                                                  objects.js:14:9
John F. W. Doe                                                               objects.js:16:9
799113600000                                                                 objects.js:19:9
29.968129030395584                                                           objects.js:20:9
John F. W. undefined                                                         objects.js:22:9
```

The left-hand side of an assignment may extract multiple values.

```
const john = { firstNames: ["John", "F."],
    lastName: "Doe" };

function range(from, to) {
    let result = [];
    for (let i = from; i < to; i++) {
        result.push(i);
    }
    return result;
}

const [firstName1, firstName2] = john.firstNames;
console.log(firstName1);
console.log(firstName2);

const { firstNames: [fn,], lastName: ln } = john;
console.log(fn);
console.log(ln);

const [three, four, ...fiveSixSeven] = range(3, 8);
console.log(three);
console.log(four);
console.log(fiveSixSeven);
```

```
John
F.
John
Doe
3
4
▸ Array(3) [ 5, 6, 7 ]
```

Destructuring an array with two elements into two new variables

Destructuring an object (and an array value within the object) into two new variables

Destructuring an array into two variables and an additional array containing the rest

JavaScript objects may be coverted from/to strings.

- Methods: `JSON.stringify` and `JSON.parse`

```
const john = {
    firstNames: ["John", "F."],
    lastName: "Doe",
    birthDate: "1995-04-29"
}

console.log(john);

const johnJsonString =
        JSON.stringify(john);

console.log(johnJsonString);

const janeJsonString = "{\"firstNames\": [\"Jane\"]," +
    " \"lastName\": \"Doe\", \"birthDate\": \"1996-06-11\"}";

console.log(janeJsonString);

const jane = JSON.parse(janeJsonString);

console.log(jane);
```

JavaScript supports the try/catch/finally construct.

```
function ensurePositive(x) {
    if (x > 0) return x;
    else throw {error: x + " is not positive"};
}

const values = [2, 3, -1, 7];
let i = 0;
for (let v of values) {
    try {
        let x = ensurePositive(v);
        console.log("SUCCESS: " + x + " is positive ");
    } catch (e) {
        console.log("FAILURE: " + e.error)
    } finally {
        i++;
    }
}
console.log(i + " elements were checked.")
```

Exceptions need not be declared.

An exception is an arbitrary value (here: object) thrown using the keyword throw.

Try-syntax is equivalent to Java

The catch block receives the thrown value and may access ist properties..

The finally block is optional. It is executed in both cases try and catch.

```
SUCCESS: 2 is positive                    exceptions.js:14:17
SUCCESS: 3 is positive                    exceptions.js:14:17
FAILURE: -1 is not positive               exceptions.js:16:17
SUCCESS: 7 is positive                    exceptions.js:14:17
4 elements were checked.                  exceptions.js:21:9
```

Modules are script files exporting functions to be imported.

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

Enables module imports/exports

```
import {add} from 'modules/math/addition'
import mul from 'modules/math/multiplication'
import * as console_util from 'modules/io/console_util'

console_util.println(mul(add(23, 42), -11));
```

Importing one exported function

Importing the default export renamed

Importing as renamed module

Accessing renamed module

Accessing renamed default function

```
export function add(x, y) {
    return x + y;
}
```

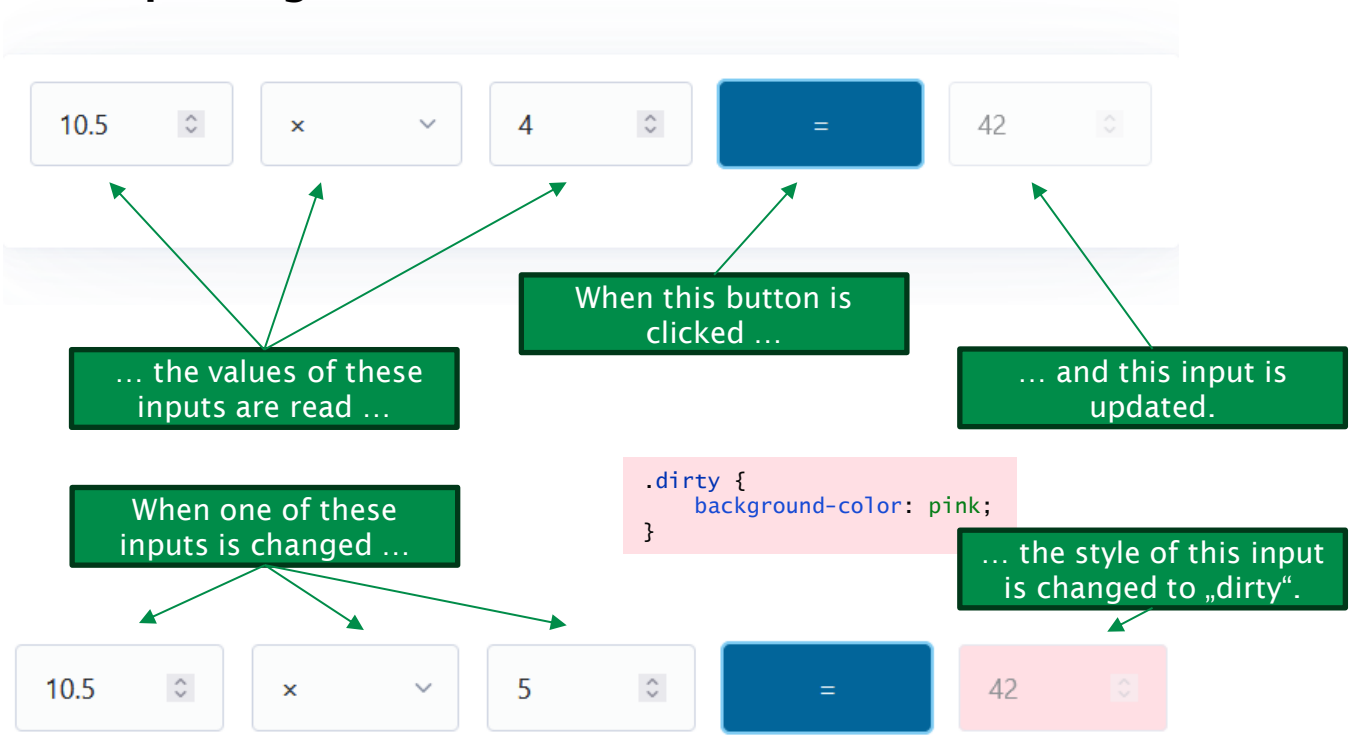
```
export function println(x) {
    console.log(x)
}
```

Exported functions can be imported by other modules.

```
export default function multiply(x, y) {
    return x * y;
}
```

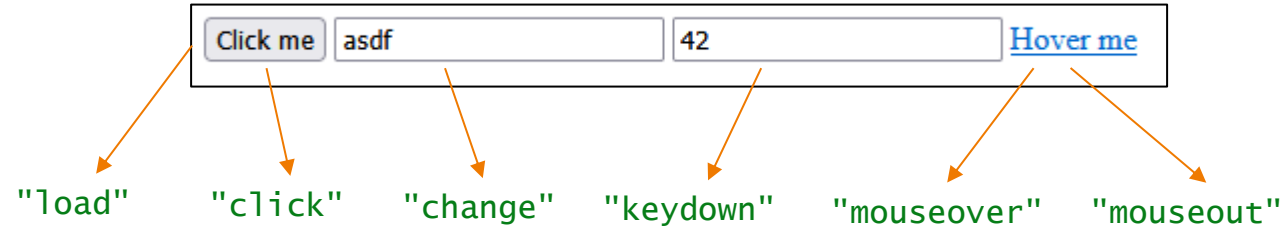
One default export per module is allowed. The function can be named differently in the importing script.

JavaScript is actually used for handling events and updating the document content.



Different HTML elements support individual event types.

```
<button id="button1" type="button">Click me</button>
<input id="text1" type="text"/>
<input id="text2" type="text"/>
<a id="a1" href="#">Hover me</a>
```

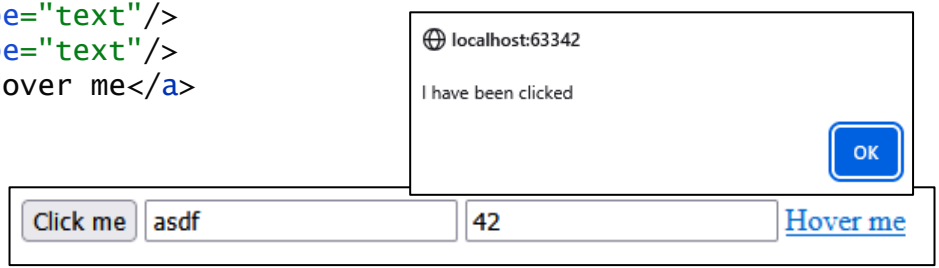


- Events can be connected to behavior using *event listeners.*
  - Design pattern *Observer* (→ Global Software Engineering)
- Registering an event listener involves three steps:
  - Selecting the *target* using a *DOM query*: `document.getElementById("text1")`
  - Specifying the event type: `"change"`
  - Implementing the event handler: `e => alert('Changed: ' + e.target.value)`

```
document.getElementById("text1").addEventListener("change",
                e => alert('Changed: ' + e.target.value));
```

Adding event listeners

```
<head>
    <script src="events.js" defer></script>
</head>
<body>
    <button id="button1" type="button">Click me</button>
    <input id="text1" type="text"/>
    <input id="text2" type="text"/>
    <a id="a1" href="#">Hover me</a>
</body>
```




```
window.addEventListener("load", () => alert('Body finished loading'));
document.getElementById("button1").addEventListener("click", () =>
                                        alert('I have been clicked'));
document.getElementById("text1").addEventListener("change", e =>
                                        alert('Changed: ' + e.target.value));
document.getElementById("text2").addEventListener("keydown", e =>
                                        alert('Pressed: ' + e.target.value));
document.getElementById("a1").addEventListener("mouseover", () =>
                                        alert('I have been hovered'));
document.getElementById("a1").addEventListener("mouseout", () =>
                                        alert('I have been unhovered'));
```

Event listeners may also be registered inline

- In HTML:

```
<body onload="alert('Body finished loading')">
    <button type="button" onclick="alert('I have been clicked')">Click me</button>
    <input type="text" onchange="alert('I have been changed')"/>
    <input type="text" onkeydown="alert('A key was pressed')"/>
    <a href="#" onmouseover="alert('I have been hovered')"
                onmouseout="alert('I have been unhovered')">Hover me</a>
</body>
```

- Or externally in JavaScript:

```
document.getElementById("button2").onclick = () => alert('I have been clicked');
```

- Inline event listeners should not be used in modern JavaScript!
  - You cannot register more than one listener of the same type to an element.
  - Code becomes unmanageable quickly. Separate HTML and JavaScript if possible!

What's in a browser event?

This is what you get when you log an event.

- Events carry all the information about the trigger and the context. E.g.:
  - x and y coordinates of a click event
  - device (e.g., mouse)
  - *Target*: Reference to the HTML element on which the event occurred

- Some types of events are connected to a *default handler* even though not explicitly defined in custom JavaScript.
  - Example 1: clicking a checkbox, the selection state gets toggled.
  - Example 2: clicking a button of a form, the data gets submitted to the server.
  - The default handler of an event e can be disabled by calling `e.preventDefault();`

Query selectors allow more advanced element selection.

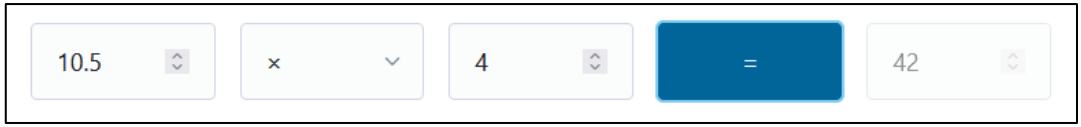
- Considered so far: `document.getElementById(...)`
- Query selectors allow to specify CSS queries to select elements:
  - `document.querySelector(...)`
    returns the first element matching the CSS query
  - `document.querySelectorAll(...)`
    returns an array of all elements matching the CSS query

```
document.querySelector("#button1").addEventListener("click",
    () => console.log("selected by id."));
document.querySelector("button.nice").addEventListener("click",
    () => console.log("selected by class."));

document.querySelectorAll("*").forEach(
    s => s.addEventListener("click",
        () => console.log("selected by wildcard.")));
document.querySelectorAll("button").forEach(
    s => s.addEventListener("click",
        () => console.log("selected by element type.")));
```

Runtime error: No matching element

Is also triggered for parent elements of, e.g., button

DOM queries and event listeners of the calculator example



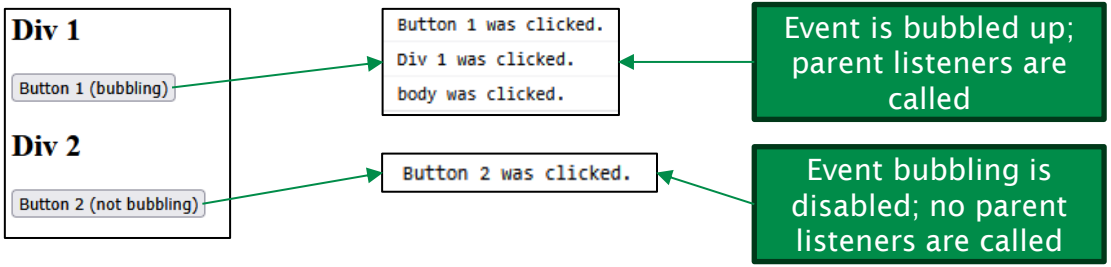

```
let operand1 = parseFloat(document.getElementById("operand1").value);
let operand2 = parseFloat(document.getElementById("operand2").value);
let operator = document.getElementById("operator").value;
let result = parseFloat(document.getElementById("result").value);

document.getElementById("operand1").addEventListener("change", e => {
    operand1 = parseFloat(e.target.value);
    setResultDirty();
});
document.getElementById("operand2").addEventListener("change", e => {
    operand2 = parseFloat(e.target.value);
    setResultDirty();
});
document.getElementById("operator").addEventListener("change", e => {
    operator = e.target.value;
    setResultDirty();
});
document.getElementById("calculate").addEventListener("click", e => {
    calculate();
    updateResult();
    setResultClean();
});
```

Events are propagated (= *bubbled*) from DOM leaf to root.

- The bubbling mechanism is enabled by default for all events.
- An event listener may *stop* this *propagation* process, so the event does not get „bubbled up“ to parent elements.

```
document.getElementById("button1").addEventListener("click",
    () => console.log("Button 1 was clicked."));
document.getElementById("button2").addEventListener("click",
    e => { e.stopPropagation(); console.log("Button 2 was clicked.");});
document.getElementById("div1").addEventListener("click",
    () => console.log("Div 1 was clicked."));
document.getElementById("div2").addEventListener("click",
    () => console.log("Div 2 was clicked."));
document.body.addEventListener("click",
    () => console.log("body was clicked."));
```

Disable event bubbling for button2

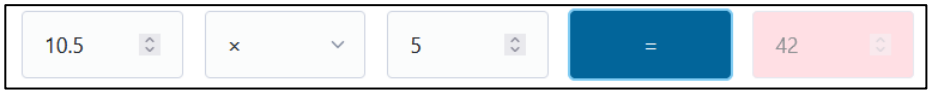


Websites are dynamically updated by manipulating the DOM.

- Initially, the DOM (document object model) reflects the HTML structure (→ chapter 03).
- DOM elements may be created, updated, and deleted using JavaScript.
- Examples:
  - Create a new element (e.g., a paragraph): `document.createElement("p")`
  - Create a new text node (character data): `document.createTextNode("asdf")`
  - Add element/node as next child to an existing parent:
    `parent.appendChild(child)`
  - Add element/node before/after an existing element to the same parent:
    `sibling.before(child)`
    `sibling.after(child)`
  - Set an attribute of a DOM element:
    `element.attribute = "value";`
    `element.setAttribute("attribute", "value");`
  - Add a CSS class: `element.classList.add("myclass");`
  - Remove a CSS class: `element.classList.remove("myclass");`
  - Remove an element (and its children) from the DOM: `element.remove();`

DOM manipulation: Calculator example continued

```
document.getElementById("calculate").addEventListener("click", e => {
    calculate();
    updateResult();
    setResultClean();
});

function calculate() {
    switch (operator) {
        case "plus": result = operand1 + operand2; break;
        case "minus": result = operand1 - operand2; break;
        case "times": result = operand1 * operand2; break;
        case "divide": result = operand1 / operand2; break;
    }
}

function updateResult() {
    document.getElementById("result").value = result;
}

function setResultDirty() {
    document.getElementById("result").classList.add("dirty");
}

function setResultClean() {
    document.getElementById("result").classList.remove("dirty");
}
```



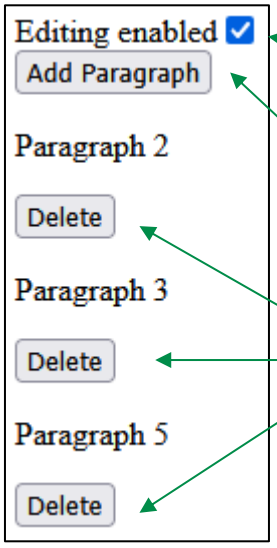
Set value of an input field

Add/remove CSS class

DOM manipulation: Paragraph example (1)

Initial content (defined in HTML)

Dynamic content (managed by JavaScript)



If this checkbox gets checked/unchecked, all buttons on the page must be enabled/disabled .

If this button is clicked, a new paragraph must be added to the page, including a button for deleting it again.

If any delete button is clicked, both the corresponding paragraph and the button itself must be removed from the page.

```
<body>
    <div>
        <label for="editing-enabled">…</label>
        <input id="editing-enabled" type="checkbox" value="true"/><br/>
        <button id="add-paragraph">Add Paragraph</button><br/>
    </div>
    <div id="paragraph-container"></div>
</body>
```

DOM manipulation: Paragraph example (2)

```
let n = 1;

function setButtonsEnabled(state) {
    console.log(state);
    for (const button of document.querySelectorAll("button")) {
        button.disabled = !state;
    }
}

function editingEnabled() {
    return document.getElementById("editing-enabled").checked;
}

document.getElementById("editing-enabled").addEventListener("click",
                                    () => setButtonsEnabled(editingEnabled()))

document.getElementById("add-paragraph").addEventListener("click", () => {
    if (editingEnabled()) {
        addParagraph();
    }
});
```

Global counter variable for labelling the next paragraph

Iterates over all buttons on the page

Sets enablement as specified by parameter

Returns whether the checkbox is checked

Update button enablement according to checkbox state when checkbox is clicked

Add paragraph when button is clicked and if editing is enabled

DOM manipulation: Paragraph example (3)

Insert the following structure at the end of paragraph-container:

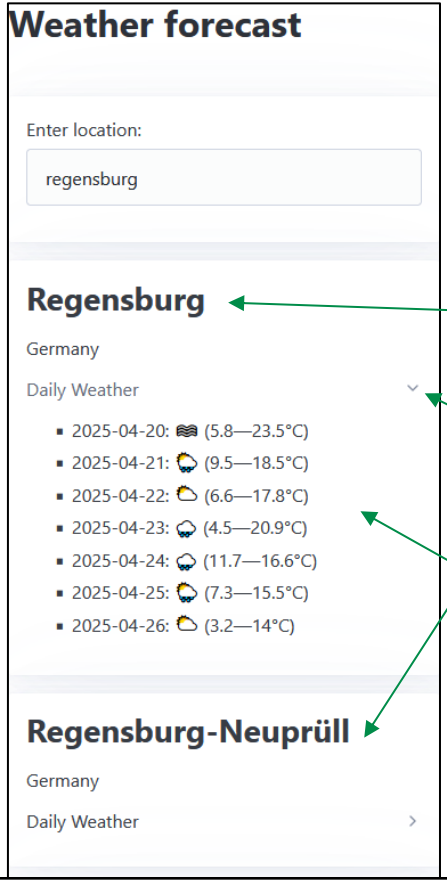
```
<div>
    <p>Paragraph {n}</p>
    <button>Delete</button>
</div>
```

```
function addParagraph() {
    const container = document.getElementById("paragraph-container");
    const div = document.createElement("div")
    const paragraph = document.createElement("p");
    paragraph.appendChild(document.createTextNode("Paragraph " + n));
    n++;
    div.appendChild(paragraph);
    const deleteButton = document.createElement("button");
    deleteButton.appendChild(document.createTextNode("Delete"));
    div.appendChild(deleteButton);
    deleteButton.addEventListener("click", e => deleteParagraph(e));
    container.appendChild(div);
}

function deleteParagraph(e) {
    e.target.parentNode.remove();
}
```

Connect the delete button to function deleteParagraph

Remove the parent element of the event target (here: the div inserted above) from the DOM

Example: Connecting to REST APIs



When the user enters a text here …

… the list of locations matching the search string must be fetched from the server and the view must be updated.

When the user expands the daily weather accordion of a specific location …

… the location data must be fetched from the server and the view must be updated.

The fetch function offers HTTP requests of all types.

- Minimal parametrization: *fetch*("http://my.url")
  - Considered by the Weather example in this chapter.
  - By default, a GET request is made without any additional headers, cookies or body.
- A second argument allows for more advanced parametrization.
  - May configure method, headers, body, and cookies (= credentials).
  - Considered by the Mensa example in chapter 06.

```
fetch("http://my.url", {
    method: "POST",
    headers: {"Accept": "application/json"},
    body: JSON.stringify(myObject),
    credentials: "same-origin"})
```

- The return value of a fetch request is a *promise* (see next slide).

Fetch requests are executed *asynchronously*.

- The execution of the current script/function is *not blocked* after starting the request.
  - The response is not available immediately, but at an unknown time in the future.
- To extract future response values, *promises* have to be used.
  - Promises can be connected callback functions, which are executed when the result becomes available.
  - They are often *chained* with other promises, e.g., parsing JSON from the body.
  - Modern JavaScript also offers the `async`/`await` syntax as an alternative to promises (not considered here)
- Pattern used here:

```
fetch(url)
    .then(response => {
        if (response.ok) return response.json();
        else throw { "error": response.statusText };
    })
    .then(result => updatePage(result));
```

The response object allows to check, e.g., status code or header values.

The body (here JSON) is parsed asynchronosly by a chained promise.

The end of the chain processes the result (here: parsed JSON), e.g., by updating the page.

Example: Consuming the OpenMeteo API (1)

```
locationInput.addEventListener("keyup", e => {
    getLocationsAndUpdateList(e.target.value);
});

function getLocationsAndUpdateList(input) {
    const url = "https://geocoding-api.open-meteo.com/v1/search?name=" + input;
    fetch(url)
        .then( response => {
            if (response.ok) return response.json();
            else throw { "error": response.statusText };
        })
        .then(locations => updateList(locations.results));
}

function updateList(locations) {
    …
    for (const location of locations) {
        …
        summary.appendChild(document.createTextNode("Daily Weather"))
        summary.addEventListener("click", e => {
            getDailyWeatherAndUpdateSummary(summary, location.latitude,
                                                     location.longitude);
        });
    }
}
```

Register event listener for search field input

This function terminates before the result is available!

Once the response arrives, the result is parsed from it and another callback function is called with it

Updates the page using DOM manipulation

Register event listener for the newly created accordion

Context information needs to be passed to the listener

Example: Consuming the OpenMeteo API (2)

```
function getDailyWeatherAndUpdateSummary(summary, latitude, longitude) {
    const url = "https://api.open-meteo.com/v1/forecast?latitude="
        + latitude + "&longitude=" + longitude +
        "&daily=weather_code,temperature_2m_max,temperature_2m_min";
    fetch(url)
        .then( response => {
            if (response.ok) return response.json();
            else throw { "error": response.statusText };
        })
        .then(weather => updateDailyWeather(summary, weather.daily));
}

function updateDailyWeather(summary, dailyWeather) { …
    const ul = document.createElement("ul");
    for (let i = 0; i < dailyWeather.time.length; i++) {
        const li = document.createElement("li");
        const weatherText =  dailyWeather.time[i] + ": " +
            weatherEmoticon(dailyWeather.weather_code[i]) +
            " (" + dailyWeather.temperature_2m_min[i] + "–"
            + dailyWeather.temperature_2m_max[i] + "°C)";
        li.appendChild(document.createTextNode(weatherText))
        ul.appendChild(li);
    }
    summary.parentNode.appendChild(ul);
}
```

Another async HTTP request

Another result callback

Iterate over weather data and construct DOM elements

References

- [Hinkula 2022] Juha Hinkula: Full Stack Development with Spring Boot 3 and React, Packt, 2022
- [Mozilla 2025] Mozilla Developer Network (MDN): HTTP web docs, https://developer.mozilla.org/en-US/docs/Web/HTTP
- [ChatGPT1] ChatGPT (https://chatgpt.com/) with prompt: “Generate a 8 bullet point summary about the history of JavaScript. It should fit one PowerPoint slide nicely.”