

# Chapter 3: Browser based programming

K. Jünemann  
Department Informations- und Elektrotechnik  
HAW Hamburg

# Content

## Chapter 3: Browser based programming

- Webpages

- The basics of Javascript

- Functions

- Objects

- Arrays

- Loading of libraries

# Webpages

3 elements of a webpage:

- ▶ HTML: structure and contents of the page
- ▶ CSS: appearance of the page
- ▶ Javascript: dynamics of the page

This course:

- ▶ just a tiny bit of HTML
- ▶ no CSS
- ▶ lots of Javascript

# HTML: Tags

HTML is a structure based on *tags*.

Syntax:

```
<tagname>Text or more tags</tagname>
```

- ▶ An opening tag `<tagname>` *must* be accompanied by a closing tag `</tagname>`
- ▶ HTML standard: only certain tags are allowed, e.g. `<h1>`, `<p>`, `<img>`, `<div>`, etc.  
(more general: XML).
- ▶ Comments:

```
<!-- <p>commented out</p> -->
```

- ▶ Tags can be nested in a tree structure:

```
<div>  
  <p>Some text</p>  
</div>
```

# HTML: Attributes

- ▶ Opening tags can have attributes:

```
<tagname attr="abc"> ... </tagname>
```

- ▶ Most common attributes:

- ▶ class
- ▶ id

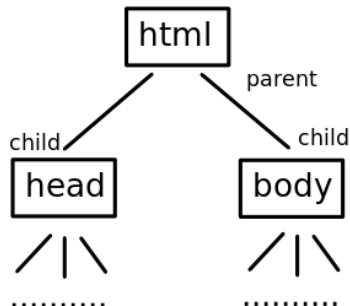
used by CSS and Javascript to identify tags.

# HTML: DOM

Mandatory structure of an html document: *DOM-tree*

(DOM: document object model)

```
<html>  
<head>  
...  
</head>  
<body>  
...  
</body>  
</html>
```



- The DOM-tree can be accessed and manipulated using Javascript.

# HTML: Loading scripts

Loading of Javascript code: the `<script>` tag

- ▶ Option 1: Inline (code *inside* html document)

```
<script type="text/javascript">  
  javascript goes here  
</script>
```

- ▶ Option 2: Code in separate file

```
<script type="text/javascript" src="filename.js">  
</script>
```

Javascript file names either relative to html document or absolut.

- ▶ Option 3: *Modules*
  - ▶ important recent Javascript feature
  - ▶ see material below

# Javascript

## Main features:

- ▶ C-like syntax:
  - ▶ control structures `for`, `while`, `if`, `else`, **etc.**
  - ▶ curly braces define scopes
- ▶ Scripting language (like Matlab):
  - ▶ No compilation and binary files
  - ▶ Access through command prompt (e.g. inside browser)
- ▶ Dynamic typing:
  - ▶ No type declaration of variables
  - ▶ Types are assigned by Javascript engine



# Javascript: A little history

- ▶ Invented around 1995 by Brendan Eich for use in the Netscape browser.
- ▶ Considered to be a nice little language with some quirks.
- ▶ Standardized under the Name of *ECMAScript (ES)*
- ▶ **Major** revision in 2015: ES6 -> Most features now supported by modern browsers.
- ▶ Huge amount of pre-ES6 code still out there!

# Web applications

HTML content can be manipulated with Javascript:

- ▶ DOM-API: standardized method to do this
- ▶ provided by `document` object

Example:

```
<script type="text/javascript">  
  const myItem = document.getElementById('myItem');  
  myItem.innerText = "Hallo";  
</script>
```

- ▶ Real world web apps use frameworks wrapping the DOM-API.
- ▶ This is *not* a course about web apps.

# Defining variables

- ▶ No type declaration required!
- ▶ Variable definition indicated by keyword: **const** or **let**
  - ▶ Use **const** as default method to declare variables:

```
const x = 2;
```

**const** variables cannot be reassigned:

```
const x = 2;  
x = 3;           // error
```

- ▶ Use **let** to declare variables that are later reassigned:

```
let x = 2;  
x = 3;           // works
```

# Defining variables

- ▶ The *"use strict"* directive:

- ▶ Turn on strict mode by inserting the string

```
use strict";
```

at beginning of file or function.

- ▶ non-strict mode is very tolerant to crappy code!
    - ▶ strict mode helps to catch errors!
    - ▶ Danger: file based strict mode applies strict mode to all files loaded later.

- ▶ Obsolete variable definition:

```
var x = 3;    // works, but don't do this!
```

**var** is old and has confusing scoping rules!

- ▶ Omitting **let** or **const** creates a global variable!

```
x = 3;    // works, but don't do this!
```

# Types

Javascript assigns one of the 7 types to each variable:

- ▶ **number**
- ▶ **string**
- ▶ **boolean**
- ▶ **undefined**
- ▶ **null** (similar to **undefined**)
- ▶ **Symbol** (we don't need this)
- ▶ **Object**

Remarks:

- ▶ Identification with the **typeof** operator:

```
typeof "Hallo" // => string  
typeof 12      // => number
```

- ▶ The first 6 types are *primitive*
  - ▶ Primitive types live on the stack and are passed by value!

# Types

## Objects:

- ▶ similar to C structs:
  - ▶ field access with dot-notation: `object.field`
  - ▶ creation with `{ }` brackets
  - ▶ Popular syntax: *JSON* (Javascript object notation) is common data exchange format
- ▶ Particularily important objects: **Array** objects
  - ▶ similar to C arrays
  - ▶ creation and element access with `[ ]` brackets
  - ▶ can store elements of different types
- ▶ Objects are reference types!
- ▶ All objects derived from **Object**

# Strings

- ▶ can be defined like `'this'` or like `"this"`
- ▶ String concatenation with `+`, e.g.

```
const s = 'abc' + 'def';    // s = 'abcdef'
```

- ▶ The function `console.log` prints strings to the console, e.g.

```
console.log('s=' + s);
```

- ▶ Numbers can be turned to strings with `parseFloat` or `parseInt` (global functions)
- ▶ Strings have many methods, e.g.

```
'abc'.charAt(1)    // => 'b'
```

See String type documentation

# The Math object

One use case for objects: define a *name space*, e.g. the `Math` name space.

Some of the `Math` functions and constants:

- ▶ `Math.PI`
- ▶ `Math.E`
- ▶ `Math.sin`
- ▶ `Math.cos`
- ▶ `Math.sqrt`
- ▶ `Math.pow`



# Logical operators

Same as in C except for logical comparison. There are two comparison operators:

- ▶ `==` takes type conversion into account.
- ▶ `===` ignores type conversion.

Almost always `===` is what you want.

## Example

```
2 == '2';      // true
2 === '2';     // false
```

# Control flow

Very similar to C:

► for - loop:

```
for(let k=0; k<5; k++) {  
  console.log(k);  
}
```

► while - loop:

```
let x = 0;  
while(x<5) {  
  console.log(x);  
  x;  
}
```

► **break** and **continue** work just as in C

# Control flow

## ► if - conditions:

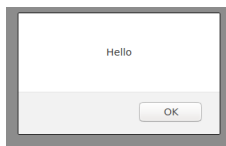
```
if (x>8) {  
    // executed if x > 8  
} else {  
    // executed if x not > 8  
}
```

## ► switch - statement:

```
switch(s) {  
    case 'case 1':  
        // some code for case 1  
        break;  
    case 'case 2':  
        // some other code for case 2  
        break;  
    default:  
        // ...  
}
```

# Simple user interaction

- ▶ `alert(msg)`: shows string *msg* in a popup window.



- ▶ `prompt(msg)`: shows string *msg* in a popup window and lets the user enter some text. Returns the text after pressing *OK*.

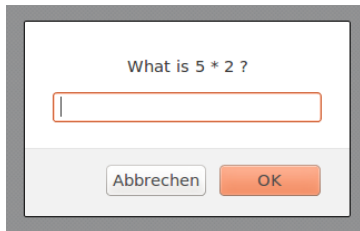


# Exercise 1

Write a web page that asks

What is  $n * m$ ?

with 'n' and 'm' random integers between 1 and 10 until the answer is correct.



A screenshot of a web form. The form has a white background and is enclosed in a gray border. At the top, it asks "What is 5 \* 2 ?". Below the question is a text input field with a red border and a cursor. At the bottom, there are two buttons: a light gray button labeled "Abbrechen" and an orange button labeled "OK".

Hints:

- ▶ `Math.random()` creates a random number.
- ▶ `Math.floor`, `Math.ceil`, `Math.round` are the usual rounding functions.

# Functions

- ▶ There are several types of function definition:
  - ▶ Function declaration with **function** keyword
  - ▶ Function expression with **function** keyword
  - ▶ Arrow functions
- ▶ The concepts of function parameters, return value and local variables are like in C.
- ▶ Functions can be passed around (e.g. as arguments to other functions) like function pointers in C.

# Functions

Function declaration example:

```
function mysum(a,b,c=0) {  
  const s = a + b + c;  
  return s;  
}
```

- ▶ no semicolon at end of statement.
- ▶ optional argument c has default value 0.

Exercise: what is wrong with this?

```
function mysum(a,b,c=0) {  
  s = a + b + c;  
  return s;  
}
```

# Functions

Function definition as variable assignment:

```
const mysum = function(a,b,c=0) {  
  const s = a + b + c;  
  return s;  
};
```

► Semicolon at end of expression!

Functions can be passed as parameters, e.g.

```
function mysum(a,b,printFunc) {  
  const s = a + b;  
  if(printFunc !== undefined) printFunc('s=' + s);  
  return s;  
}  
let x = mysum(1,2,console.log);
```



## Exercise 2

- Write a function

**mysqrt(a, tol)**

calculating  $\sqrt{a}$  with Newtons iteration:

$$x_{k+1} = \frac{1}{2} \left( x_k + \frac{a}{x_k} \right), \quad k = 1, 2, \dots$$

Start the iteration with  $x_0 = a$  and stop if

$$|x_k^2 - a| < \text{tol}.$$

- Use the function to calculate  $\sqrt{2}$  with  $\text{tol} = 10^{-8}$ .

# Objects

Example of an object definition:

```
let obj = {  
  x: 3,  
  y: "Hello"  
};
```

- ▶ Field access with dot-notation:

```
console.log("obj.x=", obj.x);
```

- ▶ Fields can be added *after* object creation:

```
obj.z = {a:1, b:2};
```

# Objects

Objects can have function fields: methods!

```
let obj = {  
  x: 3,  
  f() {console.log("Hello");}  
};  
obj.f();    // => prints "Hello"
```

The keyword **this** has to be used to refer to fields from within methods:

```
let obj = {  
  x: 3,  
  f() {console.log("x="+this.x);}  
};  
obj.f();    // => prints 3
```

## Exercise 3

1. Create an object representing a bank account, which
  - ▶ has a number
  - ▶ keeps track of depositing and withdrawing money
  - ▶ can print its state
2. Model an account with number '1234' where
  - ▶ 300 Euros are deposited,
  - ▶ 200 Euros are withdrawn,
  - ▶ the final state of the account is printed.

The output could look like this:

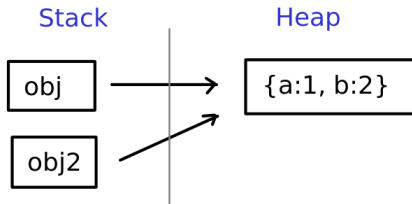
```
Account 1234 contains 100 Euros
```

# Objects

Questions: Does line 3 create an error? What is printed to the console?

```
1  const obj = {a:1, b:2};  
2  const obj2 = obj;  
3  obj2.a = 12;  
4  console.log(obj.a);
```

- ▶ Objects are reference types:
  - ▶ the object itself lives on the *heap*.
  - ▶ `obj` and `obj2` referring to the object live on the *stack*.
  - ▶ line 2 does *not* create a new object, just a new reference.



# Objects

Objects can also be created with constructor functions:

```
function Obj(a,b) {  
  this.a = a;  
  this.b = b;  
}
```

- ▶ Convention: constructors start with capital letter.
- ▶ New objects created with `new` operator

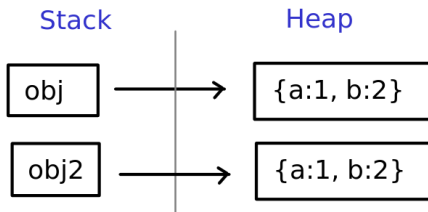
```
const obj = new Obj(1,2);  
const obj2 = new Obj(1,2);
```

Question: What is printed to the console?

```
obj2.a = 12;  
console.log(obj.a);
```

# Objects

- ▶ Each call to **new** creates a new object:



- ▶ Pitfall: forgetting **new** works but is a bug!!

```
const obj = Obj(1,2);
```

- ▶ this adds fields `a` and `b` to global object!
- ▶ Catch this by adding `"use strict";` as first line of constructor

## Exercise 4

Rewrite the Bank Account application using constructor functions.



# Arrays

Arrays are objects with special features.

► array creation:

```
let a1 = [1,2,3,4];  
let a2 = []; // empty array  
let a3 = new Array(3); // 3 x 'undefined'
```

► length of an array:

```
let len = a1.length; // len=4
```

► element access identical to C:

```
console.log(a1[0]); // prints 1  
console.log(a1[3]); // prints 4
```

# Arrays

- ▶ Looping over an array:

```
for(let k=0; k<a1.length; ++k) {  
  console.log(a1[k]);  
}
```

- ▶ Slightly shorter special looping syntax:

```
for(let elem of a1) {  
  console.log(elem);  
}
```

- ▶ Useful array methods:

```
a1.push(5);    // adds new element to end of array  
a1.pop();     // returns and removes last element
```

## Exercise 5

- ▶ Write a Javascript version

**colon(start, end, incr)**

of Matlabs colon operator :.

- ▶ Write a Javascript version

**linspace(start, end, N)**

of Matlabs `linspace` function.

# Vectorization in Javascript

Most functions in Javascript don't operate naturally on arrays (unlike Matlab)!!

The Javascript way of vectorization:

- ▶ Use Array-methods like (among many)
  - ▶ `Array.forEach`: just for side effect
  - ▶ `Array.map`: returns values
- ▶ Pass in a function operating on elements
  - ▶ arrow functions useful here

Example:

```
const sq = function(x) {return x*x;}  
const square = [1,2,3,4].map(sq);  
square.forEach(function(x) { console.log(x); });
```

# Vectorization with arrow functions

There is yet another way to define functions: *arrow* functions

```
f1 = x => x*x;           // single argument
f2 = (a,b) => a+b;       // two arguments
f3 = (x) => {x+=5;       // multiple statements as block
  return x*x; }          // return necessary in blocks
f4 = () => 5;             // no argument
```

Features:

- ▶ implicit return statement.
- ▶ doesn't rebind **this**  $\implies$  don't use arrow functions as constructor function.

Vectorization example:

```
const square = [1,2,3,4].map(x => x*x);
square.forEach((x) => console.log(x));
```

# Loading of libraries

Two options:

1. The *old* way: load `js`-files as text files
  - ▶ Easy to use
  - ▶ Order of script tags in html file matters
  - ▶ Can lead to names clashes
  - ▶ Not suitable for serious projects
2. The *new* way: use Javascripts module system
  - ▶ Systematic way to handle names
  - ▶ Requires running a web server

# Loading of libraries as text files

Option 1 (the old way): use script tags with text type

```
<script type="text/javascript" src="filename1.js"/>
<script type="text/javascript" src="filename2.js"/>
...
```

- ▶ loads one file for each script tag, in the order of the script tags.
- ▶ Usual file format for libraries: *minified* Javascript
  - ▶ comments, whitespace and new lines stripped
  - ▶ plain Javascript
  - ▶ Extension: `.min.js`, e.g. `three.min.js`.
- ▶ File name: path *relative* to html file

# Loading of libraries as text files

Example: loading `three.js` and working with some `Vector3` objects

In html file:

```
<script type="text/javascript" src="lib/three.min.js"/>
<script type="text/javascript" src="chap3/vec3Example1.js"/>
```

- ▶ `three.min.js` is located in `lib` subdirectory, `vec3Example1.js` in `chap3` subdirectory relative to html file
- ▶ `three.min.js` must be loaded before `vec3Example1.js`

In `vec3Example1.js` file: just use `three.js` library

```
const v1 = new THREE.Vector3(1, 2, 3);
console.log("v1.x = ", v1.x);
```



# Loading of libraries as text files

Same example: if html file were located in `chap3` subdirectory the script tags would be

In html file:

```
<script type="text/javascript" src="../../lib/three.min.js" />
<script type="text/javascript" src="vec3Example.js" />
```

- ▶ All paths are relative to location of html file
- ▶ `..` denotes the parent directory

## Exercise 6

1. Load the `three.js` library
2. Read the `Vector3` documentation
3. Write the following function:

```
/**  
 * calculate specular reflection  
 * @param {Vector3} incoming vector  
 * @param {Vector3} normal vector  
 * @returns {Vector3} outgoing vector  
 */  
function specRef(vin, n) {
```

- ▶ Test with the example from slide 18 of chapter 2.
- ▶ Make sure arguments `vin` and `n` are not changed by `specRef`.

# Loading of libraries as modules

Option 2 (the new way): libraries as modules

- ▶ by now: supported by all major browsers
- ▶ html loads just main js file with module type script tag
- ▶ libraries loaded from Javascript files with `import` statements
- ▶ Only works if files are provided by web server

# Loading of libraries as modules

Example: loading `three.js` and working with some `Vector3` objects

In html file:

```
<script type="module" src="chap3/vec3Example2.js"/>
```

- ▶ no need to load `three.js` module from html!

In `vec3Example2.js` file: load `three.js` module from `lib` directory:

```
import * as THREE from "../moduleLibs/build/three.  
module.js"; // path relative to js-file!  
  
const v1 = new THREE.Vector3(1, 2, 3);  
console.log("v1.x = ", v1.x);
```

# Loading of libraries as modules

*Problem:* Loading the previous example will create a *Cross-Origin* error!

*Solution:* Run a webserver in a directory which contains all required files

▶ Python: `python3 -m http.server`

▶ Node.js: `http-server . -p 8000`

Then open the page `http://localhost:8000/`

▶ see *How to run things locally* in `three.js` documentation.