L5: JavaScript

Web Engineering
188.951 2VU SS20

Jürgen Cito
L5: JavaScript

- Core Language Constructs
- Standard Library/Common APIs
- Event-driven and Asynchronous Programming
Learning Goals

• Become proficient in building client-side functionality in JavaScript
• Understand asynchronous/event-driven programming in the browser
• Be able to interact with common APIs in the browser
  (Document Object Model, Browser Object Model, HTTP, Local Storage)
JavaScript

JavaScript is an interpreted scripting language
- Originally developed by Netscape in 1995
- Standardized as ECMAScript language in the ECMA-262
  - Other ECMAScript implementations: JScript, ActionScript, QtScript, ...
- **Focus on this course on newest standard: ECMAScript 6**
- Integration into HTML
  - Internal in <head> – using the <script> element
  - Internal in <body> – using the <script> element
  - External – linking to an external JavaScript file in <head>

```
<script type="text/javascript">
  statement;
  statement;
  ...
</script>
```

- JavaScript is executed as soon as it is read!
JavaScript

Variables have dynamic types
- Changing data changes type
- String, Number, Boolean, Array
- **undefined** if there is no value
- **null** empties the variable

Objects (classically) have no underlying class!
- (Almost) everything is an object!
- Objects are associative arrays
  - key = property
  - Added when value assigned

Events
- List of pre-defined events for objects
- Mouse, keyboard, forms, loading, ...

```javascript
var data; // data undefined
data = true;  // Boolean object
data = '7';   // String object
data = Number(data); // Number object
data = new Array(); // Array object
data[0] = "A";   // 0-based
data = ["A"];   // literal
data = null;    // data is empty
```

```javascript
var student = new Object();
student.nr = 'e120...';
student.name = 'A Student';
student.age = 19;
student.hasSTEOP = true;

student = { nr : 'e120...',
            name : 'A Student',
            age : 19,
            hasSTEOP : true
            };
```

```html
<img src="..." alt="...
onclick="alert('Hello');" />
```
JavaScript

Declaration: \texttt{var/const/let \ <name> = \ <value>}

Variable literals

\begin{tabular}{|c|c|c|c|c|c|}
\hline
Number & String & Boolean & Array & Object & Function \\
\hline
10.50 & 'Text' & true & [1,2,3] & \{a: 2, b: '4'} & \texttt{function a() \{} \texttt{null} \\
105 & "Text" & false & & & \\
\hline
\end{tabular}

Dynamic type: \texttt{typeof true === "boolean"}

Type conversion functions

\begin{tabular}{|c|c|c|}
\hline
Number & Integer & String & Boolean \\
\hline
+"10" & \texttt{parseInt("10")} & 5.toString() & \texttt{Boolean("false")} \\
10 & 10 & '5' & true* \\
\hline
\end{tabular}

Every non-empty (0,"",null)-value is considered as true

Type conversions performed automatically

Lots of caveats, be careful!
JavaScript Operators

Typical Operators you expect

- Numerical operators: +, -, *, /, ++, --, %
- Comparison operators: >, >=, <, <=, ==, !=, ===, !==

Two kinds of comparison operators

- Equality with type conversion ==, !=
  - True if both expressions, converted to the same type, are equal
- (True) quality without type conversion ===, !==

```javascript
x = 0
if (x = 10) { //Assignment, returns true
    console.log("I: x has 10 assigned to it");
}
if (x == "10") { //Equality with type conversion, returns true now
    console.log("II: x is 10");
}
if (x === "10") { //Equality without type conversion, returns false
    console.log("III: x is 10");
}
```
JavaScript
Arrays (Heterogenous Lists)

Zero-based arrays with length, mixed typed elements allowed

Add/remove elements on start/end
- push/pop/shift/unshift

Sort array
- sort/reverse
- sort with comparison function

Arrays can be concatenated with `concat`
String conversion with `join(joinString)`

Beware: Certain functions on array change the structure, others don't

```javascript
x = new Array(2,3,1) //don't use
x = [2,3.6,1]
x.length //3
x[1] //3
x.push(0) // [2,3.6,1,0]
x[x.length] = 6 // [2,3.6,1,0,6]
x.pop() // [2,3.6,1,0]
x.shift() // [3.6,1,0]
x.unshift(4) // [4,3.6,1,0]
x.sort() // [0,1,3.6,4]
x.sort(function(a,b){return b-a;}) // [4,3.6,1,0]
x.reverse() // [4,3.6,1,0]
delete x[1] // [4,undefined,1,0]
x[1] = 5 // [4,5,1,0]
x = x.concat([5,6]) // [4,9,8,7,0,5,6]
x.join("-") // 4-9-8-7-0-5-6
```
Objects are associative arrays (dictionaries)
- Support dynamically adding/deleting values
- May contain functions

JSON serialization
- JavaScript Object Notation
- Serialized JS objects as data exchange format
- Objects can be directly translated into JSON (stringify)
- JSON strings can be loaded as JS objects (parse)
- Limitations: Cannot de/serialize JS functions

```javascript
x = {name: "John Doe", nr: 123456}
x.name // "John Doe"
x["nr"] // 123456
x.name = "Jane Doe" // x = {name: "Jane Doe" ...}
del x["name"] // x = {nr: 123456}
x["a"] = 1 // x = {a: 1, nr: 123456}
x.func = function(x) {alert('Hello ' + x);}
x.func(5); // Displays 'Hello5'
x = {name: "John Doe", ... , func : function (...)}
```

```javascript
x = {name: "John Doe", nr: 123456}
JSON.stringify(x)
x = JSON.parse("{'nr':5}"
```

```javascript
x = {name: "John Doe", nr: 123456}
JSON.stringify(x)
x = JSON.parse("{nr:5}"
JavaScript
Strings / Template Literals

Contained in single or double quote (‘…’, ‘…”)
• Backslash escapes (\)

String concatenation with + operator
• "Dear " + s1.name + ", thank you for…"

JavaScript Template Literals
• Use backquote (`) as delimiter
• Can span multiple lines (new line included in the string)
• Can have embedded expressions
• ${expression} — computes expression and replaces it with string value
• Replaces concatenation operations and complex expressions

message = `Dear ${student.name},
Your GPA, ${gpa(student)}, is lower than the average: ${averageGPA}`
JavaScript
Control Structures

Java-like control structures
- Condition, Switch/Case
- For-Loops
  - For: “Classic” for loops through a block a number of specified times
  - For-in: Loop through object keys/array indices
  - For-of: Loop through values of an iterable object
- While-Loop

Debug messages
- Display only in developer console: `console.log(str)`
- Popup message: `alert(str)`

```javascript
if (a > 5) {
  console.log('Larger a!');
} else {
  console.log('Smaller a!');
}
```
```
x = [1, 'a', 3] // Iterates over values
for (let i of x) {
  console.log(i);
} // logs 1, a, 3
```
```
x = [1, 2, 3] // Iterates over indices
for (let i in x) {
  x[i] = x[i] * 4;
} // x === [4, 8, 12]
```
```
x = 3;
while (x > 0) {
  console.log(x * x);
  x--;
} // logs 9, 4, 1
```
```
x = [1, 2, 3] // C/Java-like For Loop
for (let i = 0; i < x.length; ++i) {
  x[i] = x[i] * 4;
} // x === [4, 8, 12]
```
```
switch (a) {
  case 0: alert('a is 0!'); break;
  default: alert('a is different!');
}
```
```
x = 3;
while (x > 0) {
  console.log(x * x);
  x--;
} // logs 9, 4, 1
```
```
x = 3;
while (x > 0) {
  console.log(x * x);
  x--;
} // logs 9, 4, 1
```
```
x = 3;
while (x > 0) {
  console.log(x * x);
  x--;
} // logs 9, 4, 1
```
JavaScript Functions

Declared with keyword `function`

- Can have arguments (no predefined types)
- Can have a return value (no predefined return type)
- Function call only matched by name, not parameters
  - Missing parameters replaced by `undefined`
  - Additionally passed parameters are ignored
- Since ES6: Optional parameters by specifying default
  - Example: `function hasTopGrades(student, threshold=1.5)`
  - Before achieved through check for `undefined`
    in method body `if(threshold === undefined){threshold=1.5}`
- Functions can be nested (difficult variable scoping)

```javascript
function addition(a, b) {
  return a + b;
}
// Can't control types
x = addition('A ', 'Student');
// b becomes undefined,
// 7+undefined returns NaN
x = addition(7);
```

```javascript
function gpa(student) {
  if(student.grades.length == 0) return 0;
  let sum = 0;
  for(grade of student.grades) {
    sum += grade;
  }
  return sum / student.grades.length;
}
function topGrade(student, threshold=1.5) {
  return gpa(student) <= threshold;
}
s1 = { grades : [1,2,2,1] }; // gpa(s1)==1.5
s2 = { grades : [1,2,2,2] }; // gpa(s2)==1.75
topGrade(s1); // returns true
topGrade(s2); // returns false
topGrade(s1, 1.8); // returns true
```

```javascript
function counter(a) {
  function plusOne() {
    return a + 1;
  }
  plusOne();
  return a + 1;
}
```
JavaScript
Higher-Order Functions

Functions are first-class citizens
- Functions can be passed as parameters
- Functions can return other functions

Anonymous functions are functions without a name
- Also known as “Lambdas” in other languages
- Only works when passing functions as parameters or assigning them to a variable
- Syntactic Sugar: Fat Arrow Syntax =>
  - One parameter, one expression in body as return value
    student => gpa(student.grades)
  - Multiple parameter, one expression in body
    (a, b) => a + b
  - Multiple parameters, multiple expressions in body
    (a, b, message) => { console.log(message);
                        return a + b; }

```javascript
s1 = { grades : [1,2,2,1] };
s2 = { grades : [1,2,2,2] };
let students = [ s1, s2 ];

function assignGPA(student) {
  student.gpa = gpa(student)
  return student;
}

let gpa_list = students.map(assignGPA)
//returns new map
[ { grades: [ 1, 2, 2, 1 ], gpa: 1.5 },
  { grades: [ 1, 2, 2, 2 ], gpa: 1.75 } ]

gpa_list.filter(  
  student => student.gpa <= 1.5
)
// [ { grades: [ 1, 2, 2, 1 ], gpa: 1.5 } ]
```
Functions (function) are first-class objects

- Functions are objects too
- As objects, they can have properties and methods
- Difference to pure objects:
  They can be called and can have return a value

"Methods"
- Definition assigned as properties (use new and this)

```javascript
function Student(nr, name, age, hasSteop) {
  this.nr = nr;
  this.name = name;
  this.age = age;
  this.hasSteop = hasSteop;

  this.finishSteop = function() {
    this.hasSteop = true;
  }
}
var jc = new Student('e0828...', 'Jurgen Cito', 29, false);
jc.finishSteop();
```

```javascript
class Student {
  constructor(nr, name, age, hasSteop) {
    this.nr = nr;
    this.name = name;
    this.age = age;
    this.hasSteop = hasSteop;
  }

  finishSteop() {
    this.hasSteop = true;
  }
}
var jc = new Student('e0828...', 'Jurgen Cito', 29, false);
jc.finishSteop();
```

ES6 Class is syntactic sugar for functions as objects
JavaScript
Object Prototypes

All objects have a **prototype**
- All prototype are object, any object can be a prototype
- Objects inherit properties and methods from prototype
- Object.prototype is top of prototype chain (its prototype is null)

```javascript
var aStudent = { nr : 'e120...', ... }; // aStudent -> Object.prototype
var students = [ aStudent, ... ]; // students -> Array.prototype -> Object.prototype
var jc = new Student('e08...', ...); // aoStudent -> Student.prototype -> Object.prototype
function print() { ... }; // print -> Function.prototype -> Object.prototype
```

Existing prototypes can be extended at any time
- Beware of monkey patching!

```javascript
String.prototype.distance = function() { ... }; // monkey patching

function Student(nr, name, age, hasSteop) { ... } // as before
Student.prototype.university = 'TU Wien'; // add property
Student.prototype.summary = function() { // add "method"
  return this.nr + ' ' + this.name;
};
```
JavaScript
Variable Declaration and Scoping

Declaring variables is possible with keywords
var, let, const

- Function-scope: var
  - When a variable is declared within a function with var, it will only live in that function
  - If the variable has been declared outside the function, it lives in the outer scope
  - Function scopes can nest with nested functions
- Block-scope: let, const
  - Variables declared with let or const will only exist within its block
  - const identifiers cannot change
- Global Scope: Variables are global by default
  - If no keyword is used, the variable is available anywhere (globally)

```javascript
students = [...]
function GPAMessage(student) {
  let message = '';
  // no declaration of students required
  const averageGPA = avgGPA(students);
  const studentGPA = gpa(student)
  if(studentGPA < averageGPA) {
    const GPADifference = studentGPA - averageGPA;
    message = `Great job, ${student.name}. Your GPA is ${GPADifference}
    higher than the average`;
  } else {
    // GPADifference is now not available any longer in this scope
    message = `Dear ${student.name},
    Your GPA, ${studentGPA}, is lower than the average: ${averageGPA}`;
  }
  return message;
}
```
Establish namespace in separate module files
- One module per file, one file per module
- Ability to create named export or default exports

Export specific elements
- `export function gpa(student) {...}
- function gap(student) {...}; export gap;

Import in other files
- `import * as Student from './student.js';
s = {...}; s.gpa = Student.gpa(s);
- `import { gpa } from './student.js';
s = {...}; s.gpa = gpa(s);

Source: samanthaming.com
Browser/Web APIs

- Browser Object Model (window)
- Storage (Local Storage, Cookies)
- Document Object Model
- (many more we are not discussing)

Browser Object Model (BOM)

- Allows access to browser objects
- Not standardized! (But very similar in all modern browsers)

Window is the global object
- All global objects, functions, and variables are members of window

<table>
<thead>
<tr>
<th>Object</th>
<th>Property and Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>Other global objects, open(), close(), moveTo(), resizeTo()</td>
</tr>
<tr>
<td>screen</td>
<td>width, height, colorDepth, pixelDepth, ...</td>
</tr>
<tr>
<td>location</td>
<td>hostname, pathname, port, protocol, assign(), ...</td>
</tr>
<tr>
<td>history</td>
<td>back(), forward()</td>
</tr>
<tr>
<td>navigator</td>
<td>userAgent, platform, systemLanguage, ...</td>
</tr>
<tr>
<td>document</td>
<td>body, forms, write(), close(), getElementById(), ...</td>
</tr>
</tbody>
</table>

**Popup Boxes**
- alert(), confirm(), prompt()

**Timing**
- setInterval(func,time,p1,...), setTimeout(func,time)
Storing Data

Cookies
- String/value pairs, Semicolon separated
- Cookies are transferred on to every request

Web Storage (Local and Session Storage)
- Store data as key/value pairs on user side
- Browser defines storage quota

Local Storage (window.localStorage)
- Store data in users browser
- Comparison to Cookies: more secure, larger data capacity, not transferred
- No expiration date

Session Storage (window.sessionStorage)
- Store data in session
- Data is destroyed when tab/browser is closed

```javascript
document.cookie = "name=Jane Doe; nr=1234567; expires="+date.toGMTString();
```

```javascript
let storage = permanent ? window.localStorage : window.sessionStorage;
if(!storage["name"])
    storage["name"] = "A simple storage"
alert("Your name is " + storage["name"]);
```
Document Object Model (DOM)

- Tree structure for interacting with (X)HTML and XML documents
  - HTML elements as objects with properties, methods and events
- Standardized by the W3C
  - Platform- and language-independent
  - Levels: Level 1 ⊆ Level 2 ⊆ Level 3

```
Document
   ├── Root Element: <html>
       │   └── Element: <head>
       │       └── Element: <title>
       │           └── Text: "My Title"
       └── Element: <body>
            ├── Element: <a>
            │    └── Attribute: "href"
            │        └── Text: "My link"
            └── Element: <h1>
                └── Text: "Hello"
```
Document Object Model (DOM)

Retrieving Elements
- ID, tag name, class name
- Document property
- ES6: Selector-based Access

Change Elements
- Content (innerHTML)
- Element attributes
- Element style
- Element class (className, classList)

Manipulating DOM Nodes
- Create, append, remove, ...

DOM Traversal on Element
- parentElement, nextElementSibling, previousElementSibling, childNodes

```javascript
let title = document.getElementById("title");
let links = document.getElementsByTagName("a");
let greens = document.getElementsByClassName("green");
let imgs = document.images;
let firstParaBox = document.querySelector("p.box");
let allBoxes = document.querySelectorAll("p.box,div.box");

// Change Elements

// Content
title.innerHTML = "newTitle";

// Attributes
links[0].href = "http://...";
greens[0].setAttribute("href", ...)

// Style
greens[0].style.color = "red";

// Class
greens[0].className = "red"
greens[0].classList.add("dangerzone")

// Create, append, remove, ...
let header = document.createElement("h2");
let text = document.createTextNode("SubTitle");
header.appendChild(text);
document.removeChild(title);
document.replaceChild(title, header);
```
DOM Updates and Accessibility

Changing the DOM can introduce accessibility issues
- (Frequent) updates can confuse screen readers
- Updates may not be able in high magnification
- Certain updates may be invisible (e.g., introducing a red border for errors)
- Updates may come too fast (before pre-update part of the page was read)

Guidelines
- If the content updates for more than 5 seconds, provide ability to pause, stop, or hide them
- Inform users of changes: Setting focus, Highlight, Alert, Live Region (ARIA Term)
- Communicate to user that page is dynamic
- Provide static HTML page alternatives

Careful testing necessary (screen reader testing, proxy testing with text browser)
Event-driven and Asynchronous Programming

- Events in HTML/DOM
- Asynchronous Programming: Callbacks vs. Promises
- async/await Syntax
- HTTP (Fetch)
Event-driven JavaScript Programming

Event-driven Programming:
- Flow of the program is determined by responding to user actions called events
- Writing programs driven by user events

Source: Hovik, University of Washington, CSE154, https://courses.cs.washington.edu/courses/cse154/19sp/sections/week03-tues/slides/index.html#4
DOM Events

Event callback attached to HTML elements

```html
<button onclick="alert('Test!')">Test me!</button>
```

Event types (selection)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load, unload</td>
<td>User enters/leaves page</td>
</tr>
<tr>
<td>change</td>
<td>Form input field changes</td>
</tr>
<tr>
<td>focus / blur</td>
<td>User focuses/unfocuses an input field</td>
</tr>
<tr>
<td>Submit</td>
<td>Form is submitted</td>
</tr>
<tr>
<td>mouseover, mouseout</td>
<td>Mouse enters/leaves region</td>
</tr>
<tr>
<td>mousedown/mouseup/click</td>
<td>Mouse click events</td>
</tr>
<tr>
<td>keydown / keyup/keypress</td>
<td>Keyboard events</td>
</tr>
<tr>
<td>drag</td>
<td>User drags an element</td>
</tr>
</tbody>
</table>
Sending Asynchronous Requests (Callbacks)
HTTP (XMLHttpRequest)

- Used callbacks - a mechanism to provide a function that gets called once you receive a response from HTTP
- Callbacks resulted in increasingly nested callback chains dubbed “callback hell”

http://callbackhell.com/

const API_BASE_URL = 'https://pokeapi.co/api/v2';
const pokemonXHR = new XMLHttpRequest();
pokemonXHR.responseType = 'json';
pokemonXHR.open('GET', `${API_BASE_URL}/pokemon/1`);
pokemonXHR.send();

pokemonXHR.onload = function () {
    const moveXHR = new XMLHttpRequest();
    moveXHR.responseType = 'json';
    moveXHR.open('GET', this.response.moves[0].move.url);
    moveXHR.send();
    moveXHR.onload = function () {
        const machineXHR = new XMLHttpRequest();
        machineXHR.responseType = 'json';
        machineXHR.open('GET', this.response.machines[0].machine.url);
        machineXHR.send();
        machineXHR.onload = function () {
            const itemXHR = new XMLHttpRequest();
            itemXHR.responseType = 'json';
            itemXHR.open('GET', this.response.item.url);
            itemXHR.send();
            itemXHR.onload = function () {
                itemInfo = this.response;
                console.log('Item', itemInfo);
            }
        }
    }
}

Code adapted from Wendell Adriel: https://medium.com/@wendell_adriel/from-highway-to-callback-hell-to-stairway-to-async-await-heaven-e90f73309ed4
Sending Asynchronous Requests (Promises)
HTTP (fetch)

`fetch` API allows to process HTTP requests/responses using **promises**:

- Promises are a general wrapper around asynchronous computations and callbacks
- They represent how to get a value - you tell it what to do as soon as it receives the value
- Promise is a proxy object for a value that is not yet known. It is modelled with the following states
  - Pending (initial state)
  - Fulfilled (execution successful)
  - Rejected (operation failed)

```javascript
fetch('./movies.json')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

A promise defines a function (`resolve`, `reject`) asynchronously loading data
- `resolve(data)` on success
- `reject(errorObject)` otherwise

The callback is defined with `then(success, failure)`
- `success(result)`
- `failure(errorObject)`

`catch(failure)` is a shortcut for `then(undefined, failure)`
async/await is a special syntax to work with promises

- async is a keyword around a function that wraps a promise around its return value.

  ```javascript
  async function f() { return 1; }
  f().then(alert); //requires then to resolve result
  ```

- await is a keyword that makes JavaScript wait until the promise is resolved and can the return the value (only works within async functions!)

  ```javascript
  let response = await fetch("./movies.json")
  ```

- send the avatar

  ```javascript
  let bucket = document.createElement('img');
  bucket.src = githubUser.avatar_url;
  bucket.className = "promise-avatar-example";
  document.body.appendChild(bucket);
  ```

- wait 3 seconds

  ```javascript
  await new Promise((resolve, reject) =>
    setTimeout(resolve, 3000));
  ```

- remove the avatar

  ```javascript
  bucket.remove();
  return githubUser;
  ```

Adapted from https://javascript.info/async-await