Online Multimedia

Winter Semester 2019/20

Tutorial 03 – Advanced JS
Today’s Agenda

• The EventLoop
• JS Modules, Types & OOP
• TypeScript
• Roundup Quiz
The Event Loop
The Event Loop

• JavaScript is inherently single threaded
  • This for example prevents race conditions when modifying the DOM
  • WebWorkers are a way to explicitly program concurrency in JS

• But timeouts, callbacks, promises, etc. somehow happen „while“ other code is being executed

• This is due to very well-defined interleaving in the JavaScript EventLoop

• Every browser engine implements their own version based on standardized semantics
The Main Loop

• The Main Loop executes sequential code
• It also delegates when to execute messages and when to render
The Message Queue

• The message Queue waits for enqueued messages

• If messages are queued it executes a single message to completion and returns to the main loop

• If a messages takes a long time to complete, the main loop is blocked
  • Do not put long running code in messages, instead split it into multiple messages
Blocking with Messages

// This blocks the EventLoop
setTimeout(function () {
    while (true) {
    }
}, 0);

// This does not
setTimeout(function cb () {
    setTimeout(cb, 0);
}, 0);
Rendering

• Rendering first calculates the stylesheet and the result layout

• It then paints the WebPage

• Rendering never happens during a message execution, so it always paints the result of all DOM manipulations in a message

• But not every message triggers a re-rendering
To make sure you only modify the DOM when it is actually re-rendered, use `requestAnimationFrame`

All queued RAFs are executed as part of the rendering and therefore at the same framerate as the rendering

Making sure the EventLoop is not blocked maintains a consistent framerate and therefore responsiveness and user experience
Debugging the EventLoop

Network
Interactions
Timings

Main — https://www.youtube.com/watch?v=hSUtNUncS50

Task
Parse HTML
Evaluate Script (anonymous)
ytplayer.load (anonymous)

Summary
Bottom-Up
Call Tree
Event Log

Self Time
Total Time
Activity
1.7 ms 4.7 % Parse HTML
0.2 ms 0.7 % Evaluate Script
2.1 ms 5.8 % Function Call
0.2 ms 0.6 % g.h.init
7.4 ms 20.2 % Recalculate Style
0.4 ms 1.0 % g.h.Ad
0.2 ms 0.4 % (anonymous)

Frames 6.5 ms 41.5 ms
CPU

955 ms 960 ms 965 ms 970 ms 975 ms 980 ms 985 ms 990 ms 995 ms 1000 ms 1005 ms 1010 ms 1015 ms 1020 ms 1025 ms
Breakout #1

• Use the `breakout-01.html` skeleton

• There are three versions of a spinner that shows you how different animation methods perform

• Complete these implementations

• Observe and try to explain why the three spinners behave differently
Modules
ES6 Modules

• Modules allow splitting code into functionally distinct parts

• After many custom module implementations, there is now a standardized module system

• This allows exporting and importing of values, functions and classes:

  • `export function myCoolFunction () { /* code */ };`

  • `import { myCoolFunction } from './myCoolModules.mjs';`
ES6 Modules

• When using modules in HTML the script must specify that it is a module:
  `<script type="module" src="myScript.mjs"></script>`

• `<script nomodule>` can be used as fallback for old browsers.

• By convention, JS files with modules have the *.mjs file extension

• Modules can also be loaded dynamically:
  ```javascript
  const { foo } = await import('./myScript.mjs');
  ```
Module Systems

• With modules officially introduced into JavaScript, they will be adopted by major frameworks and libraries eventually.

• Browser support will still be tricky though, as it is for all new JS features.

• Currently there are still a lot of custom-built module systems in use, e.g. in NodeJS. So beware: Justs because it looks similar to the standardized ES6 modules, it might not be ES6 modules!
Types & OOJS
JavaScript Types

• While JavaScript is not type checked, types do exist
  • undefined
  • number
  • string
  • boolean
  • Object
  • Symbol
  • any

• A good editor or IDE can give code completion and warnings based on type information

```javascript
/** @type {Object} */
const x = { };

/** @type {{ a: number }} */
const x = { a: 5 };

/** @param {{ foo: string }[]} x */
function f (x) { x.map((i) => i.foo); }

/** @type {(any) => number} */
const f = function (x) { return 5; }
```
JavaScript Types

- `typeof` and `instanceof` can be used to determine the type of a value but their behaviour can be counter-intuitive

- `typeof` more or less returns the JavaScript type

- `instanceof` is concerned with the inheritance hierarchy

```javascript
const obj = { };  // "object"
const arr = [ ];  // "object"
const fun = function () { };  // "function"

typeof obj // "object"
typeof arr // "object"
typeof fun // "function"
typeof null // "object"

arr instanceof Object // true
arr instanceof Array // true
fun instanceof Object // true
fun instanceof Function // true
```
OO in JS

• JavaScript supports **Prototype-based** Object Orientation

• Classes are defined by their **constructor functions**:

```javascript
function Animal (name) {
    this.name = name;
}
```

and instantiated with the **new** keyword

```javascript
new Animal('Lion');
```
OO in JS – Inheritance

• Inheritance in JS happens via **Prototype**

• Each constructor function has a prototype object

• The prototype contains all inherited members, usually functions and constants

```
Animal.prototype.greeting =
    function () { return this.name + ' says hello!'; }
```

```
(new Animal('Lion')).greeting() // 'Lion says hello!'  
```

• To inherit members one must establish a prototype chain along which members are inherited
OO in JS – Inheritance

function Animal (name) { this.name = name; }
Animal.prototype.greeting = function () { return this.name + ' says hello!'; }

function Lion () {
    Animal.call(this, 'Lion');  // ~ super call
    this.maneLength = Math.random();
}

Lion.prototype = Object.create(Animal.prototype);  // "Lion extends Animal"
Lion.prototype.roar = function () { return 'ROOAAAAR!'; }
// Lion extends Animal
Lion.prototype = Object.create(Animal.prototype);

// MountainLion extends Lion
MountainLion.prototype = Object.create(Lion.prototype);

// Prototype Chain:
// MountainLion.__proto__ -> Lion
// MountainLion.__proto__.__proto__ -> Animal
class Animal {
    constructor (name) {
        this.name = name;
    }

    greeting () { return this.name + ' says hello!'; }
}

class Lion extends Animal {
    constructor () {
        super('Lion');
        this.maneLength = Math.random();
    }

    roar () { return 'ROOOAAAR!'; }
}
## OO in JS – ES3 vs. ES6

<table>
<thead>
<tr>
<th>Old function syntax</th>
<th>New ES6 class syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater flexibility</td>
<td>Familiar syntax</td>
</tr>
<tr>
<td>Better information hiding, quasi-private values</td>
<td>Easier to define static members and inheritance</td>
</tr>
<tr>
<td>Closer to the actual JS behaviour</td>
<td>Hides weird JS details</td>
</tr>
<tr>
<td>Full browser support</td>
<td>Good support in modern browsers</td>
</tr>
<tr>
<td>No real private members</td>
<td>No real private members</td>
</tr>
</tbody>
</table>
WTH is this?

• You may have noticed that we used `function` instead of fat arrow for defining member functions

• That is because fat arrow does bind the `this` reference from the object instance, so `this` would be whatever it was when defining the member function
WTH is this?

```javascript
Animal.prototype.greeting =
  function () { return this.name + '...'; }

new Animal('Lion').greeting() // 'Lion ...

Animal.prototype.greeting =
  () => this.name + '...';

new Animal('Lion').greeting() // '...'
```
TypeScript
ANOTHER LANGUAGE

Yeah, sure, whatever
WHAT IF I TOLD YOU

THAT TYPESCRIPT ISN'T
REALLY A NEW LANGUAGE
TypeScript

- TypeScript is a super-set of JavaScript

- It essentially is JavaScript with type-annotations and some other features, like
  - Class-member visibility
  - Interfaces
  - Enums
  - etc.

- This is good because it allows
  - More concise code
  - More developer control
  - Static checking
const x: number = 5;

function joinString (x: string[]): string { ... }

// There are interfaces in TS
interface Engine {
  horsePower: number;
  ignite (): number;
}

// Classes can have member-visibility
class Car {
  private engine: Engine;
  public start (): void { ... }
}
const x: number = 5;

function joinString (x: string[]): string { ... }

// There are no interfaces in JS
interface Engine {
  horsePower: number;
  ignite (): number;
}

// Classes can't have member-visibility
class Car {
  private engine: Engine;
  public start (): void { ... }
}
The JS/TS Ecosystem

• Obviously your browser does not know how to handle TS
• It has to be transpiled into JS with \texttt{tsc}, the TypeScript compiler
• This, and many many more tools, can be installed with npm via

\texttt{npm install \textasciitilde g typescript}

• The \texttt{\textasciitilde g} is to install it globally so you can use it as CLI

• Now you can compile \texttt{*.*.ts} files to \texttt{*.*.js} files:

\texttt{tsc my-file.ts}

• \textbf{Source-maps} allow you to debug the TS code, even though the browser uses the generated JS.
Breakout #2

- In the breakout skeleton, a small and buggy Javascript web application is given

- Improve the code’s data structures by applying OO-programming patterns through TypeScript, so that one recognized the bugs at compilation time
Roundup Quiz

1. Why does it make sense to keep JS single-threaded?
2. How do you convert between a float and an int in JS?
3. Is a static private value in Javascript part of the prototype or the constructor function?
4. What does the –g flag do for npm?