

Responsive web. JavaScript

Alen Murtić



01 Responsive web & responsive design

02 Javascript - Introduction and history

03 Syntax, objects and functions

04 Prototype inheritance, closure and classes

01



Responsive web & responsive design

Responsive web - mobile vs desktop

- Initially, web was desktop-only and then desktop-orientated, mobile was an afterthought
- Mobile browsers used to rearrange web page flow to fit in on their screens
 - Not that bad but not great, browsers rendered the pages differently
 - Alternatives: special mobile sites which didn't need to be re-arranged (e.g. m.sofascore 📱)
- Mobile phones started having "insane" resolutions, e.g. 5" phone was "Full HD" like 32" TV
 - Solution: **viewport** of those phones is smaller, **CSS pixel isn't the same as physical pixel**
- Add viewport tag: `<meta name="viewport" content="width=device-width, initial-scale=1.0">`
 - More: https://www.w3schools.com/css/css_rwd_viewport.asp
- Deeper explanation: [Viewport vs Screen Resolution, DPR vs PPI](#)

Accessible web basics

- Can user redefine font size on a website to make it more readable for them?
 - YES, on a browser level! But, the website needs to be implemented in a particular way
- Elements sizing: different CSS units - absolute and relative
 - **px (pixel)** - absolute - **CSS pixels**
 - **em** - font-size relative to parent font-size; width, etc. relative to the font-size of the element
 - **rem** - relative to font-size of the root element (mostly to html tag)
- If root element has relative font-size like **100%**, then it reads browser settings

Accessible web - what to use?

- Sizing in px, font-size in px
 - Complex layouts like Sofascore web page
 - Because of that viewport tag, zooming-in pages makes them accessible
- Sizing in px, font-size in rem
 - Layout is fixed, content is browser-resizable
- Sizing in rem, font-size in rem
 - Accessible simplish layouts
 - Everything is browser-resizable
 - Also works for complex layouts, but it's a bit harder

- Responsive design - page must look good on a large number of different devices
 - To consider different device capabilities - resolution, orientation, speed
 - Possibly different style rules by group
- Two strategies:
 - Progressive enhancement - “mobile first” - start with smaller screens and then handle wider devices
 - Graceful degradation - start with the best and largest devices, then handle all the problems

CSS Media

- Media queries - CSS checks for device capabilities
 - `@media screen and (min-width: 400px) { ...will apply styles only on screens wider than 400px }`
 - Much more than just screen size, e.g. `@media (pointer: coarse|fine) {...}` - [link to docs](#)
- Using viewport width and height in CSS: 100vh and 100vw - full viewport width and height
 - Great for desktop, a bit messy on mobile 🙄
 - Problem: each browser handles disappearing toolbars and menus differently

CSS Media Breakpoints

- Media queries are usually not set at random screen sizes, but on few standardized values called media breakpoints
 - Standardized per site/development team/framework, not industry-wide
 - [Bootstrap breakpoints](#)
- Sofascore supports devices from 320px upwards
 - Lower resolutions work, but don't look particularly great
- We'll talk more about responsive design later

CSS Math Functions

- CSS supports 4 Math functions as values: `calc`, `min`, `max`, `clamp`
 - [Practical Uses of CSS Math Functions: calc, clamp, min, max](#)
 - e.g. setting main's `min-height: calc(100vh - 30px)`
 - We have a 30px header and the rest of the viewport should have content
 - Allows to dynamically calculate some value directly in CSS
 - Plain CSS is often faster than JS

02



Javascript - Introduction and history

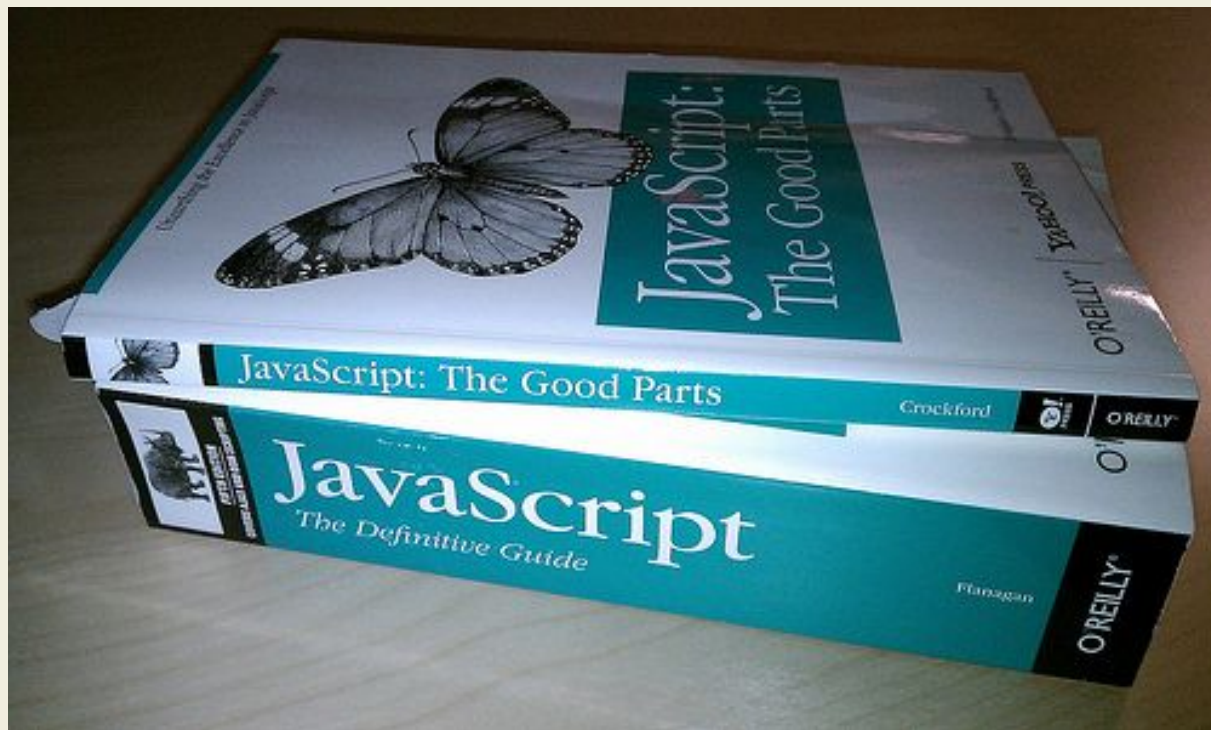
Introduction and history

Literature

- Kyle Simpson: You don't know JS
- Douglas Crockford: JavaScript: The Good Parts
- “Java is to JavaScript as ham is to hamster” - Jeremy Keith, 2009

Introduction and history

Literature



Introduction

- Multi-paradigm language (Procedural, Object-Oriented, Functional)
 - Choice is on a programmer, the mix of concepts from multiple paradigms
- Backward compatible (not forward compatible)
 - Backward compatible -> old code can run on newer versions
 - We achieve forward compatibility in browser by [transpiling](#) new code for older browsers and using [polyfills](#)
- Implementation of the ECMAScript standard (TC39 Committee)
- Core technology of the Web

History

- 1994
 - Web pages are booming, Netscape Navigator is the most popular web browser
 - Pages are static, people want dynamic pages
- 1995
 - Companies tried embedding Java into browsers -> **FAIL**
 - [Brendan Eich](#) created LiveScript
 - LiveScript shipped as JavaScript in Netscape Navigator -> **SUCCESS**
 - Microsoft IE uses JScript -> reverse engineered JavaScript with all the quirks

Introduction and history

History - rest

- ECMA standardization (1996.)
- Microsoft IE has 95% market cap -> standardized bugs from JScript
- Major improvements:
 - ECMA2009 (ES5)
 - ECMA2015 (ES6)

Introduction and history

JavaScript engine

- Key part of a browser
- Runs JavaScript code
 - Compiles - JIT compiler
 - Optimizes
- **V8** (Chrome & Node & Edge), **SpiderMonkey** (Firefox), **Nitro** (Safari)
 - All implement official specifications, but there may be differences in undefined behaviour



Syntax, objects and functions

Syntax, objects and functions

JavaScript Basics

- C like syntax (`if`, `switch`, `while`)
 - Dynamically typed
- Types (`typeof` operator):
 - Primitive: `number`, `boolean`, `string`, `undefined`
 - Complex: `object`, `null`, `function`
- Variables:
 - `var` - function scope, reassignable
 - `let` - block scope, reassignable
 - `const` - block scope, cannot be reassigned
- Semicolons - [it's complicated](#)
 - At Sofascore, we prefer not to use them




Hello world example - helloWorld.js






Semicolons example - asi.js





Types example - types.js



Syntax, objects and functions

Type Coercion

- Process of converting a value from one type to another
- Happens implicitly during comparison using `==`
 - Implemented that way in initial version to make life easier for developers
- Comparison using `===` was added later
- **Always use triple equal**
 - Or at least be certain that double equal works

Syntax, objects and functions

Type Coercion



[Explanation](#)

Syntax, objects and functions

Plus operator

- Used for:
 - Concatenating strings
 - Adding numbers
- If arguments are number and string -> coerce number to a string

```
2 + 4 // 6
'Sofa' + 'score' // Sofascore
2 + 4 + 'score' // 6score
'Sofa' + 2 + 4 // Sofa24
```

Objects

- Most important structure in JavaScript
 - Collection of named values
 - Values can be of any type (even function)
 - Functions can be stored in variables
- **MUTABLE**
 - Adding or changing the property of an object doesn't change object (its reference stays the same) -> IMPORTANT for React (how state updates and why props changing is bad)
- Creation:
 - `new Object`
 - `Object.create`
 - `{ foo: 'bar' }` -> most used


This

- Reference to an object it belongs to
- Used in functions, to access properties of an owner (similar to Java)
 - Java methods have `this` as a reference to an instance on which method is called
 - In JavaScript, functions have `this`
 - If global function is called (without owner), this refers to the global object (window, global)
 - Try to play with `whatIsThis.js` example
- Making your code dependent on value of global `this` can lead to bugs, we rarely use it in modern React (modern React is mostly functional components)

Syntax, objects and functions

Functions

- Can be pictured as **Object**
 - can have properties and methods, even have object prototype (e.g. console.log instanceof Object is true)
- Function Declaration
 - `function doSomething() { ... }`
- Function Expression
 - `const doSomething = function() { ... }`
- Arrow functions
 - `const return2024 = () => 2024`
 - `function` and `return` can be avoided
 - no this



Function arguments example - functionArgs.js





Prototype inheritance, closure and classes

Prototype inheritance

- Mechanism by which JS objects inherit features
- Prototype chain -> chain of prototypes for an object (all accessible)
 - Prototype chain -> list of inherited prototypes starting from more specific to more general. If an object doesn't have property or function then look in its prototype. If a more specific prototype doesn't contain searched property or function, go into the next, more generic prototype, ...
 - All prototypes end with object prototype -> "Object is the king in JavaScript"
 - `instanceof` operator tests if an object has some prototype in its chain (e.g. function is an instance of an object)
- Most OO languages -> class inheritance
- `constructor` method is called with `new` keyword

Prototype inheritance, closure and classes

Prototype inheritance advantage over class inheritance

- Additional methods can be added to existing prototypes
 - e.g. we can add a method to Array prototype
 - This is how polyfills work
- Closest comparable language feature are *extension methods* in C#



Prototypes example - prototypes.js



Prototype inheritance, closure and classes

Classes

- Classes are type of functions in JS
- Cleaner and more familiar syntax
 - **constructor** function
 - **static** methods
 - **extends** and **super**
- Classes are actually functions -> functions are objects -> all classes can be written as objects
- Class syntax is similar to other OO languages (Java)
- Classes can have getters and setters too (get and set keywords)



Classes example - classes.js



Prototype inheritance, closure and classes

Hoisting

- Process of moving all **DECLARATIONS** to the top of a file
 - Declarations are hoisted (`var x`), not definitions (`x = 'X'`)
- Hoisted:
 - Function declarations (`function() {...}`)
 - `var` variables
- Not Hoisted:
 - Function expressions (`const a = () => {}`)
 - `const` and `let` variables
 - classes



Hoisting example - hoisting.js



Prototype inheritance, closure and classes

Closure

- Functions are bundled with references to the surrounding state (Lexical Environment)
- Inner function can access outer function scope **even after** outer function has finished its execution
- Makes encapsulation possible



Closure example - closure.js



**Thank you for your
attention!**

