JavaScript – old JS, ES5, ES6, and ES7 features needed in e.g. React development and seen in many React/Redux/Material UI/Node backend examples.

The list of the ES features needed in React development. Some are even older than ES5, but tricky. Focus is on features that are 1. tricky, 2. risky, 3. confusing, 4. treacherous, 5. difficult, 6. error-prone or many of those.

See the Mozilla Developer Network links for all of these!

- **let** – block-scoped variable (Until ES6 we only had 'var' with only two possible scopes: function and global, though implicit global vars and var hoisting cause some risks if not coding well) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
- **const** - block-scoped constant (the first immediate value needs to be assigned right away and will be constant, e.g. the object reference. But the _contents_ of that object and so on are not protected by const!). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
- **pass-by-value** (JavaScript and e.g. Java have only this) vs. **pass-by-reference** (E.g. C/C++/C# have also this). For beginners it must be confusing to pass values that are references and still understand it as pass-by-value! When we pass the value of the reference value, a reference, it's pass-by-value. Only if we would be able to pass a reference to that reference variable itself, it would be pass-by-reference! (Then the function could change the value of the original variable, which is only rarely desirable)
- **shallow copy** (first layer of objects is duplicated as separate objects, but after that the references refer to original second layer objects = not independent copy) vs. **deep copy** (all objects in the, even deeper object structure are duplicated as separate objects and the original objects are safely separate)
- **arrow functions** (shorter syntax, implicit return, reference 'this' auto-bound to outer scope, 2 more) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
- **.map** method/function https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- **.forEach** function for many kind of collections https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/forEach https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/forEach
- **.filter** method https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- **.reduce** method https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
- ES6 **class** syntax https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class
- ES6 class **inheritance** syntax https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends

- **template literals** and **placeholders** (with backticks ` and ${ } to get rid of this kind of String concatenation clumsiness: ”Hello”+name+”!”)
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

- **spread operator** (spread notation/spread syntax) to make a ‘deeper copy’ of an object, instead of the ‘totally shallow copy’. Copying goes **one level deep** = the properties of the original and copy object are separate. (But those separate properties may contain references to same objects)
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator

- ES6 **export** and **import** from a **module** to another (default export or named export)
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export
  - So after ES 2015 = ES6 this version has been spreading wider in JS:
    - in original.js **export default** someObj; // default export (one per module)
    - in file using.js **import** myObj **from** './original'; // default import and naming ‘myObj’
    - in original.js **export** someObj; // named export (multiple items possible)
    - in file using.js **import** {someObj **as** myObj} **from** './original'; // named import (and rename)
  - (It replaced the older the CommonJS way: https://en.wikipedia.org/wiki/CommonJS)
    - (in original.js **module.exports** = someObject; // exposing someObject as/from module)
    - (in file using.js var copyOfSomeObject = **require(**'/original.js'**)**; // getting an instance of it)

- extra **trailing comma** was allowed at the end of lists already in old JS. ES5 added it to object literals and ES8 to functions. [1,2,3**,**] {name:”Joe”,yob:1986**,**} foo(2,3**,**);
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Trailing_commas

- **Property accessor** used so that its name is not hard-coded string, but comes from a variable:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer Scroll down to ”**Computed property names**”.

```
this.setState({[event.target.name]: event.target.value});
```

  compare to this: `this.setState({firstName: event.target.value});`

  if the event’s target’s name was string ”firstName”. Note: same feature as in our **{[a]:a,[b]:b}** example

- OLD JS: function **parameter default values**
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

- OLD JS: **leaving arguments out is only allowed at the end of a function argument** list while calling a function

  That’s why we need to write e.g. **(_ , index) => index%2==0** where we are marking the skipped parameter with dummy name _ . That is counted as a parameter, but not needed/used. We need to write the _ as otherwise index would not be the second parameter like it needs to be. Similar use:
  (_ => whatever_code_here)

- OLD JS: **falsy values.** Anything that will be considered false while e.g. given to if condition. if(a)
  https://developer.mozilla.org/en-US/docs/Glossary/Falsy (You could remember 3-9 from Scrum team size)
  https://developer.mozilla.org/en-US/docs/Glossary/Truthy

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Equality_()

- **short notation object literals** of this kind: { a } which means same as { a : a}

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#New_notations_in_ECMAScript_2015

  In React JSX {{a}} means first going to JS mode using the outer { } and then having that shortened {a} object literal inside

- **IIFE**, ~~SIAF, SEAF~~ https://developer.mozilla.org/en-US/docs/Glossary/IIFE Learn the first example(s) here: https://developer.mozilla.org/en-US/docs/Glossary/IIFE#Examples

- **Destructuring assignment**. Destructuring object or array values into separate variables https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

- **Difference between** JavaScript **Object literals (=JavaScript code)** and **JSON (=Text, String in JS, thus not JS)**: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#Object_literal_notation_vs_JSON So JSON is not JavaScript, but is text that is compatible with JS object serialization.

- A **new way of defining methods** (Methods: object-attached functions, object's function members) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions#Description

- (Smaller curiosity) JavaScript doesn't allow **identifiers starting with number**. But what if you get the JSON text {"123":"Yeah"} and parse it as an JavaScript object?

  ```
  var a = JSON.parse('{"123":"Yeah"}');
  console.log(a.123);        // Error, unexpected number
  console.log(a."123");      // Error, unexpected String
  console.log(a["123"]);     // ok, prints: Yeah
  console.log(a[123]);       // ok, prints: Yeah
  ```

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#Accessing_properties

******* END ********

~~Advanced features for the very highest grade(s).~~

Not in the 2021-09-13 exam: (Some of these **just because** they were **not included yet, possibly will be in future exams**). Most likely **in future exams**

- (A bit abstract and advanced) JavaScript **closures** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
- **async functions** with an implicit Promise and a possible **await** inside where e.g. AJAX call will be initiated, but then we start to wait for the answer at the await. The thread though is freed to do other stuff in the mean time: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- ES6 **promises** (promise1.then(function2) ….) Easier to read handling of asynchronous function calls and their callbacks. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

OUT OF SCOPE: The items below in this list:

- (so far nothing added here)

Went to the other exams:

- React/Redux/Ajax/Material-UI stuff to front-end exam. (But JS/ES features found in React code belong to this exam, basically all the features above are such). Some JavaScript will be applied in Front-end and Back-end exams. But then 100% related to the full-stack project code