



# Why is mutability such a pain in JavaScript?

<https://jit.team>

front.jit (29.03.2023)

# Today's agenda

- Motivation and real-life examples
- Pros and cons of immutability
- How to handle immutability in JS



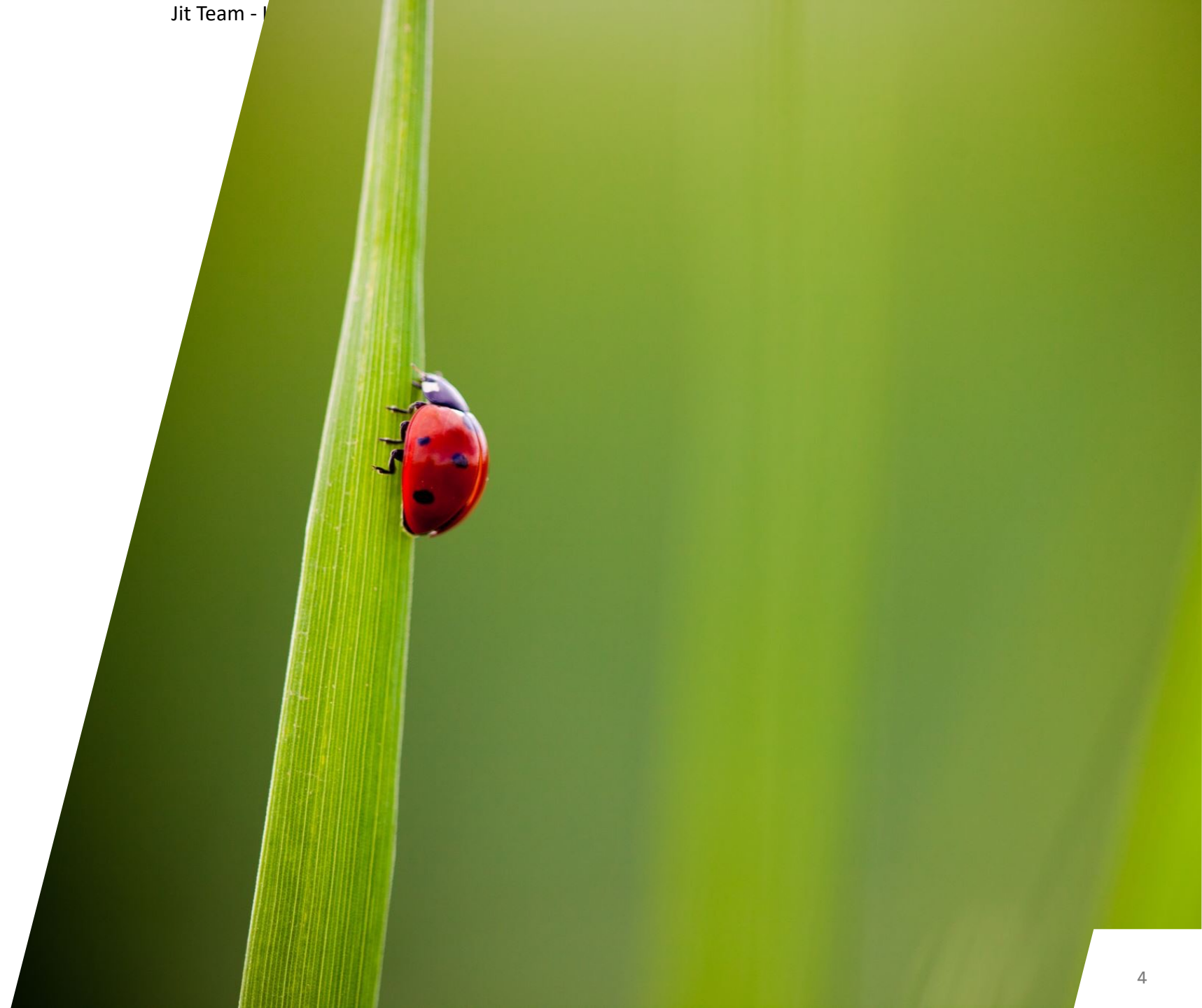
# Maciej Kankowski



- In Jit Team for 5 years 🤖
- Frontend (React/Next/Vue + 🎨)
- Mobile (🍏)
- Internal projects / mentoring 🎓 🎓
- Travel, drones, books, cars 🚀

# Motivation

- Bugs and hard debugging
- Discover problems quickly
- Ways to avoid accidental object mutations



# Example: Reference vs value equality

```
JS index.js x
1 const x = 10;
2 const y = 20;
3
4 var object1 = { x: 10, y: 20 };
5 var object2 = { x: 10, y: 20 };
6
7 // reference equality
8 console.log("object1 == object2", object1 == object2);
9 console.log("object1 === object2", object1 === object2);
10
11 // value equality
12 console.log("object1.x == x", object1.x == x);
13 console.log("object1.x === x", object1.x === x);
```

Console 0 Problems 3

Console was cleared

object1 == object2 false

object1 === object2 false

object1.x == x true

object1.x === x true

<https://codesandbox.io/s/js-reference-vs-value-equality-zb2rdt?file=/src/index.js>



# Example: Object mutation

```
1 document.querySelector("#app header").innerHTML = `  
2 <h1>JS - object mutation</h1>  
3 `;  
4  
5 const car = { manufacture: "VW" };  
6 const sameCar = car;  
7 const newCar = { ...car };  
8  
9 sameCar.manufacture = "BMW";  
10  
11 document.querySelector("#app section").innerHTML = `

12 Car: ${car.manufacture},  
13 Same car: ${sameCar.manufacture},  
14 New car: ${newCar.manufacture}  
15 </p>`;


```

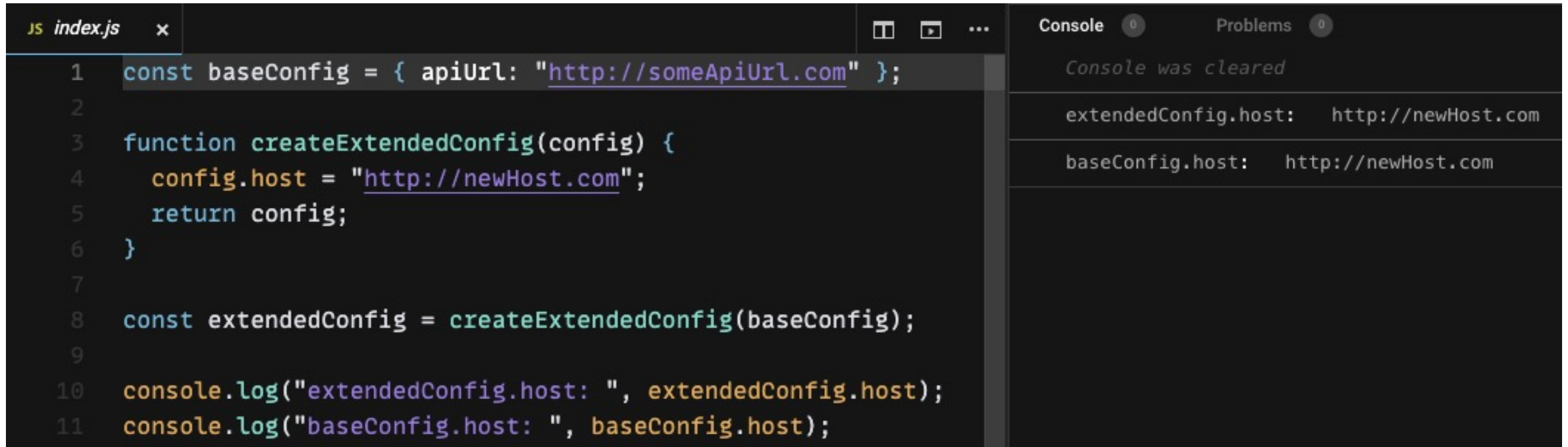
< > ↺ https://vcv37l.csb.app/

## JS - object mutation

Car: BMW, Same car: BMW, New car: VW

<https://codesandbox.io/s/js-object-mutation-vcv37l?file=/src/index.js>

# Example: Object mutation (from function)



The screenshot shows a code editor with a file named `index.js`. The code defines a base configuration object, a function to create an extended configuration, and then calls the function to create an extended configuration. The console output shows the result of the function call, demonstrating that the extended configuration has the new host URL, while the base configuration remains unchanged.

```
1 const baseConfig = { apiUrl: "http://someApiUrl.com" };
2
3 function createExtendedConfig(config) {
4   config.host = "http://newHost.com";
5   return config;
6 }
7
8 const extendedConfig = createExtendedConfig(baseConfig);
9
10 console.log("extendedConfig.host: ", extendedConfig.host);
11 console.log("baseConfig.host: ", baseConfig.host);
```

Console output:

```
Console was cleared
extendedConfig.host: http://newHost.com
baseConfig.host: http://newHost.com
```

<https://codesandbox.io/s/js-object-mutation-from-function-k2uyen?file=/src/index.js>

# Example: Date object mutation

```
1 document.querySelector("#app header").innerHTML = `
2 <h1>JS - date mutation</h1>
3 `;
4
5 const date = new Date();
6 const sameDate = date;
7 const newDate = new Date(date);
8
9 sameDate.setFullYear(2100);
10 newDate.setFullYear(3100);
11
12 document.querySelector("#app section").innerHTML = `<p>
13 Year: ${date.getFullYear()},
14 Same Year: ${sameDate.getFullYear()},
15 New Year: ${newDate.getFullYear()}
16 </p>`;
```



<https://codesandbox.io/s/js-date-mutation-mhkflp?file=/src/index.js>



# Pros & cons of immutability

- But before, what is data immutability?
- Simpler programming and debugging
- Immutable data is slower than mutable
- DOM rendering, database quering etc. is even much slower
- Being aware of working with reference and values data types
- Being aware of working with shallow and deep comparision
- It matters in "reactive" frameworks like React.js, Vue.js...

# How to handle immutability in JS

- Reduce the usage of mutative functions like **push**, **unshift** or **splice**
- Use shallow-copying functions like **map**, **filter** or **reduce**
- `const` vs `Object.freeze()` (runtime)
- **readonly** - Typescript modifier (compile-time only)
- Immer.js, immutable.js
- React: **useDeepCompare**, **shouldComponentUpdate** (memo)
- Pure Function
- Deep & shallow clone methods...

# Shallow clone methods

- Object.assign(), spread operator



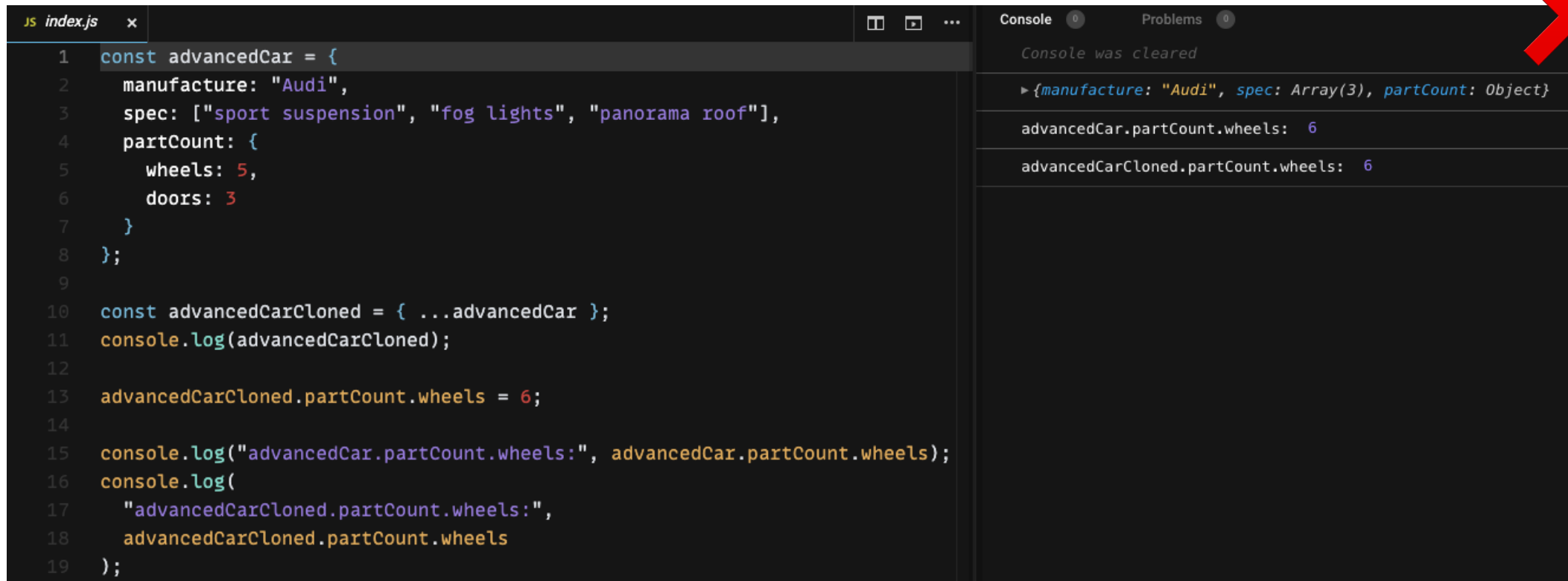
```
const game = {  
  title: "God of War",  
}  
// works fine since there is no nested object  
const gameCopy = {...game}
```



```
const game = {  
  title: "God of War",  
  date: new Date(2022),  
  genre: ["Action", "RPG"]  
}  
  
const gameCopy = {...game}  
  
gameCopy.genre.push("Fighting")  
  
gameCopy.date.setTime(2023)  
  
// game.genre will be the same array as gameCopy  
// game.date will be modified
```

# Shallow clone methods

- Object.assign(), spread operator



```
1 const advancedCar = {
2   manufacture: "Audi",
3   spec: ["sport suspension", "fog lights", "panorama roof"],
4   partCount: {
5     wheels: 5,
6     doors: 3
7   }
8 };
9
10 const advancedCarCloned = { ...advancedCar };
11 console.log(advancedCarCloned);
12
13 advancedCarCloned.partCount.wheels = 6;
14
15 console.log("advancedCar.partCount.wheels:", advancedCar.partCount.wheels);
16 console.log(
17   "advancedCarCloned.partCount.wheels:",
18   advancedCarCloned.partCount.wheels
19 );
```

Console

Console was cleared

► {manufacture: "Audi", spec: Array(3), partCount: Object}

advancedCar.partCount.wheels: 6

advancedCarCloned.partCount.wheels: 6

<https://codesandbox.io/s/js-shallow-merge-ow7jgq?file=/src/index.js>

# Deep clone methods: JSON api

- JSON.stringify(), JSON.parse()
- Basic objects, arrays, primitives

```
let ingredients_list = ["noodles", { list: ["eggs", "flour", "water"] }];
let ingredients_list_deepcopy = JSON.parse(JSON.stringify(ingredients_list));

// Change the value of the 'list' property in ingredients_list_deepcopy.
ingredients_list_deepcopy[1].list = ["rice flour", "water"];
// The 'list' property does not change in ingredients_list.
console.log(ingredients_list[1].list);
// Array(3) [ "eggs", "flour", "water" ]
```



⚠ Functions, Symbols, objects that represent HTML, recursive data... cannot be serialized

# Deep clone methods: structuredClone

- Original values to be transferred rather than cloned to the new object.
- It's browser (or Javascript runtime) feature rather than JavaScript language itself (window)



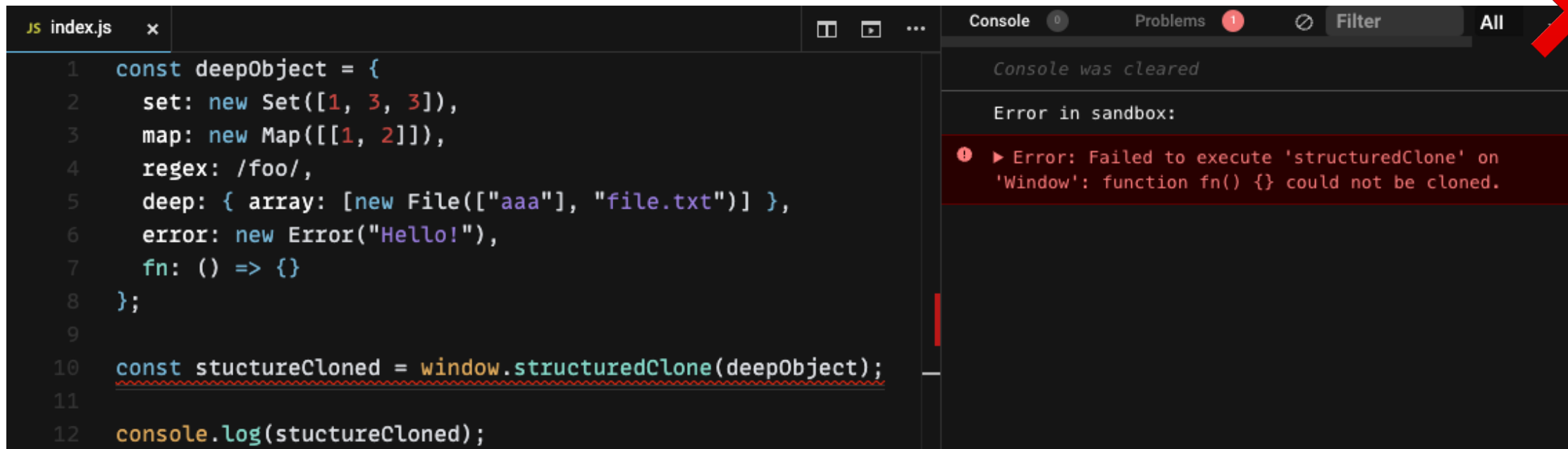
```
// Create an object with a value and a circular reference to itself.  
const original = { name: "MDN" };  
original.itself = original;  
  
// Clone it  
const clone = structuredClone(original);  
  
console.assert(clone !== original); // the objects are not the same (not same identity)  
console.assert(clone.name === "MDN"); // they do have the same values  
console.assert(clone.itself === clone); // and the circular reference is preserved
```

⚠ The same serialization issues like JSON.stringify()



# Deep clone methods: structuredClone

- Original values to be transferred rather than cloned to the new object.
- It's browser (or Javascript runtime) feature rather than JavaScript language itself (window)



```
JS index.js x
1  const deepObject = {
2    set: new Set([1, 3, 3]),
3    map: new Map([[1, 2]]),
4    regex: /foo/,
5    deep: { array: [new File(["aaa"], "file.txt")] },
6    error: new Error("Hello!"),
7    fn: () => {}
8  };
9
10 const stuctureCloned = window.structuredClone(deepObject);
11
12 console.log(stuctureCloned);
```

Console 0 Problems 1 Filter All

Console was cleared

Error in sandbox:


! ▶ Error: Failed to execute 'structuredClone' on 'Window': function fn() {} could not be cloned.

⚠ The same serialization issues like JSON.stringify()

<https://codesandbox.io/s/js-window-structuredclone-xkt0i7>

# Deep clone methods: external libraries

- Lodash (`_cloneDeep`), Ramda
- Structural sharing
- Bundle size



```
JS index.js x
1  const cloneDeep = require("lodash.clonedeep");
2
3  const deepObject = {
4    set: new Set([1, 3, 3]),
5    map: new Map([[1, 2]]),
6    regex: /foo/,
7    deep: { array: [new File(["aaa"], "file.txt")] },
8    error: new Error("Hello!"),
9    fn: () => {}
10 };
11
12 const newDeepObject = cloneDeep(deepObject);
13
14 console.log(newDeepObject);
```

Console 0 Problems 0

Console was cleared

```
▼ {set: Set, map: Map, regex: /foo/}
  ▼ set: Set
    0: 1
    1: 3
  ▼ map: Map
    1: 2
    regex: /foo/
  ▼ deep: Object
    ► array: Array(1)
    error: ► Error: Hello!
  ▼ fn: f fn() {}
    ► <constructor>: "Function"
```

# Conclusions

- Bugs & debbuging: Working with immutable data prevents the developers to avoid accidental changs
- Performance impact: DOM renders is much slower than using immutable data
- Shallow (vs. Deep) comparition is a better performant so it's a default solution for React, Redux etc.
- „Immutable libraries such as Immer can employ structural sharing, which effectively returns a new object that reuses much of the existing object being copied from”



# References

- [https://developer.mozilla.org/en-US/docs/Glossary/Shallow\\_copy](https://developer.mozilla.org/en-US/docs/Glossary/Shallow_copy)
- [https://developer.mozilla.org/en-US/docs/Glossary/Deep\\_copy](https://developer.mozilla.org/en-US/docs/Glossary/Deep_copy)
- <https://redux.js.org/faq/immutable-data>
- <https://daveceddia.com/react-redux-immutability-guide/>
- <https://reactkungfu.com/2015/08/pros-and-cons-of-using-immutability-with-react-js/>
- <https://blog.klipse.tech/javascript/2021/02/26/structural-sharing-in-javascript.html>
- <https://javascript.plainenglish.io/deep-clone-objects-the-right-way-in-javascript-a7ded9d23860>
- <https://monsterlessons-academy.com/posts/mutable-and-immutable-data-in-javascript>
- <https://www.builder.io/blog/structured-clone>