JAVASCRIPT

Jaka jest różnica między setTimeout() a setInterval()?

01

setTimeout to funkcja, która wykonuje kod lub funkcję **raz** po określonym czasie wyrażonym w milisekundach.

```
setTimeout(function () {
    console.log('Done after one second!');
}, 1000);
```

setInterval to funkcja, która wykonuje kod lub funkcję **cyklicznie** co określony czas, zaczynając po pierwszym okresie. Funkcja zwraca identifikator, którego można użyć do przerwania działania za pomocą **clearInterval()**.

```
const intervalID = setInterval(function () {
    console.log('Done every second...');
}, 1000);

// clearInterval(intervalID); // Przerwanie
```

JAVASCRIPT

Jaka jest różnica między == i ===?

02

Podwójny znak równości == (equality) sprawdza równość dwóch wyrażeń lub wartości. Przed porównaniem próbuje je przekonwertować do tego samego typu.

Potrójny znak równości === (identity) porównuje dodatkowo zgodność typów, ale nie dokonuje konwersji.

x	y	x == y	x === y
5	5	true	true
1	'1'	true	false
null	undefined	true	false
0	false	true	false
‘1,2’	[1,2]	true	false

JAVASCRIPT

Jak wyczyścić tablicę?

03

1) Nadpisanie tablicy pustą tablicą:

```
let arr = ['hello', 'world'];
arr = [];
console.log(arr);  // []
```

2) Ustawienie właściwości **length** tablicy na wartość 0:

```
let arr = ['hello', 'world'];
arr.length = 0;
console.log(arr);  // []
```

3) Wykorzystanie metody **splice**, która **może** być użyta do usuwania elementów z tablicy (**splice** modyfikuje tablicę i zwraca usunięte elementy:

```
let arr = ['hello', 'world'];
let removed = arr.splice(0, arr.length);
console.log(arr);  // []
console.log(removed); // ["hello", "world"]
```

JAVASCRIPT

Czym się różnią let i var?

04

Różnica dotyczy zasięgu w jakim zadeklarowana jest zmienna

- **var** deklaruje zmienną dla całej funkcji (***function scope***)
- **let** deklaruje zmienną dla bloku (***block scope***)

Dodatkowo, **wszystkie deklaracje** (function, var, let, const i class) są **hoistowane**, ale tylko **var** są inicjalizowane przez undefined - **let** i **const** pozostają niezainicjalizowane.

```
function saveCourse(courseData) {
  if (courseData) {
    console.log(maxStudents);  // undefined
    console.log(courseLength); // ReferenceError
    var maxStudents = 12;      // function scope
    let courseLength = 30;     // block scope
  }
  console.log(maxStudents);    // 12
  console.log(courseLength);  }// ReferenceError
}
```

JAVASCRIPT

Jakie są typy danych w JS?

05

W JavaScript wyróżniamy 9 typów:

undefined	`typeof undefined === "undefined"`
Boolean	`typeof true === "boolean"`
Number	`typeof 2 === "number"`
String	`typeof "2" === "string"`
BigInt	`typeof 2n === "bigint"`
Symbol	`typeof Symbol("id") === "symbol"`
null	`typeof null === "object"`
Object	`typeof new Object() === "object"`
Function	`typeof alert === "object"`

operator typeof wskazuje, że jest to object, jednak wynika to z faktu, że każdy konstruktor funkcji dziedziczy po konstruktorze Object.

JAVASCRIPT

Jak utworzyć obiekt?

06

```
function Course(courseTitle) {
    this.title = courseTitle;
}
// 1. Za pomocą konstruktora
const jsCourse = new Course('JS');

// 2. Za pomocą Object Literal
const angularCourse = { title: 'Angular' };

// 3. Za pomocą new Object
const reactCourse = new Object();
reactCourse.title = 'React';

// 4. Za pomocą Object.create
const vueCourse = Object.create(jsCourse);
vueCourse.title = 'Vue';
```

JAVASCRIPT

Jakie znasz *falsy values*?

07

Falsy values są to wartości, które są traktowane tak samo jak **false**, gdy są użyte w kontekście **Boolean**, np. w instrukcji **if**.

JavaScript używa konwersji typów rzutując każdą wartość na Boolean w sytuacjach, które tego wymagają, np. w instrukcji warunkowej if lub pętlach. Wyróżniamy 8 ***falsy values***:

`false`	`Domyślna wartość boolean`
`null`	`Brak wartości`
`undefined`	`Wartość niezdefiniowana`
`NaN`	*`Not a number`*`, np. "abc" / 4`
`“ “`	`Pusty string`
`0`	`Liczba 0`
`-0`	`Liczba -0`
`0n`	`Wartość 0 jako BigInt`

JAVASCRIPT

Co różni null od undefined?

08

W świecie JS **undefined** oznacza zmienną, która została zadeklarowana, ale nie posiada jeszcze wartości, na przykład:

```
let courseLength;
console.log(courseLength);          // undefined
console.log(typeof courseLength); // "undefined"
```

Z kolei **null** jest wartością, która może zostać przypisana do zmiennej, która nie posiada żadnej wartości:

```
let courseContent = null;
console.log(courseContent);          // null
console.log(typeof courseContent); // "object"

console.log(null === undefined);     // false
console.log(null == undefined);      // true
```

JAVASCRIPT

Czym jest spread operator?

09

Spread operator pozwala **rozwinąć** wyrażenie w miejscach, gdzie potrzebne jest **wiele argumentów** (do wywołań funkcji), **wiele elementów** (tworzenie tablic) lub **wiele zmiennych**.

```
const results = [5, 5, 4];
function calculateResults(a, b, c) {
    return a + b + c;
}
calculateResults(5, 5, 4);    // 14
calculateResults(...results); // 14

var courses = ['Vue', 'React', 'Angular'];
const [firstCourse, ...allTheRest] = courses;
console.log(firstCourse); // "Vue"
console.log(allTheRest); // ["React", "Angular"]
console.log([...firstCourse]);// ["V", "u", "e"]
```

JAVASCRIPT

Czym jest rest operator?

10

Rest operator wygląda dokładnie tak samo jak **Spread operator**, ale jest używany do destrukturyzacji obiektów i tablic.

W przeciwieństwie do Spread operatora, ***pakuje*** wiele elementów w jeden:

```
function sum(...results) {
    return results.reduce((a, b) => a + b);
}
sum(5)                    //  5
sum(5, 5)                 // 10
sum(5, 5, 4, 3, 1, 5); // 23
```

JAVASCRIPT

Co oznacza NaN?

NaN reprezentuje wartość, która ***nie jest liczbą***, przykładowo:

```
'abc' / 4                          // NaN
```

Do sprawdzenia czy zmienna lub wyrażenie zawiera **NaN**, możemy skorzystać z funkcji **Number.isNan()**

```
Number.isNan('abc' / 4);   // true
Number.isNan('100');       // false
```

Co ciekawe **NaN** nigdy nie jest równa innej liczbie, nie jest równa nawet samej sobie:

```
console.log(NaN == NaN);  // false
console.log(NaN === NaN); // false
console.log(typeof NaN);  // "number"
```

JAVASCRIPT

Jak poprawnie stworzyć URL?

12

Aby stworzyć poprawny adres URL można skorztystać z funkcji **encodeURI**, która zakoduje wszystkie znaki specjalne za wyjątkiem `< , / ? : @ & = + $ #>`.

W drugą stronę, aby odczytać adres URI zakodowany przez **encodeURI** można skorzystać z **decodeURI**. Funkcja rzuca wyjątek w przypadku, gdy nie można poprawnie odczytać adresu URI.

```
const uri = 'https://example.com/ąę';
const encodedURI = encodeURI(uri);
console.log(encodedURI);
// https://example.com/%C4%85%C4%99

console.log(decodeURI(encodedURI));
// https://example.com/ąę
```

JAVASCRIPT

Co odróżnia call od apply?

13

Funkcje **call** i **apply** służa do podmiany kontekstu wykonania funkcji, czyli podmianę wskaźnika **this**, przez co możliwe jest wywołanie funkcji zdefiniowanej w obiekcie A na obiekcie B.

Funkcja **apply** wywołuje funkcję używając do tego przekazanej wartości **this** oraz listy argumentów przekazanej **w formie tablicy**:

```
const numbers = [1, 2], others = [3, 4];
numbers.push.apply(numbers, others);
console.log(numbers);   // [1, 2, 3, 4]
```

Różnica pomiędzy **call** a **apply** polega jedynie na tym, że **call** przyjmuje argumenty **w postaci pojedynczych wartości**:

```
const numbers = [1, 2], others = [3, 4];
numbers.push.call(numbers, ...others);
console.log(numbers);  // [1, 2, 3, 4]
```

JAVASCRIPT

Do czego służy funkcja bind?

14

Funkcja **bind**, w porównaniu do **call** i **apply** nie zwraca wyniku, ale funkcję, którą można wywołać w przyszłości.

Dzięki powiązaniu z **this** funkcja ta - niezależnie kiedy zostanie wywołana - będzie znała poprawną wartość this z momentu kiedy została utworzona.

Dzięki takiemu zachowaniu, bind stosuje się do **obsługi asynchronicznej eventów**.

```
const numbers = [1, 2];
const addFn = numbers.push.bind(numbers);
addFn(3);
addFn(4);
console.log(numbers);  // [1, 2, 3, 4]
```

JAVASCRIPT

Co różni slice i splice?

15

W JavaScript funkcje **slice** oraz **splice** są często mylone. Mimo że mają bardzo podobne nazwy, mają zupełnie inne zadania:

slice	**splice**
Nie modyfikuje tablicy	Modyfikuje tablicę
Zwraca część tablicy jako nową tablicę	Zwraca usunięte elementy tablicy
Służy do wybierania elementów z tablicy	Służy do dodawania lub usuwania elementów z listy

POJĘCIA

Czym jest scope?

16

Scope jest to zakres zmiennych, który określa ich widoczność i dostępność w różnych miejscach kodu. Jest zależny od sposobu, w jaki zostaną zadeklarowane.

Przy pomocy słowa kluczowego **let** deklarujemy lokalne zmienne w ramach **block scope**. Zakres zmiennych jest ograniczony tylko do bloku kodu, w którym zostały zadeklarowane.

Słowo kluczowe **const** służy do deklarowania stałych. Ich zakres jest analogiczny do zmiennych zadeklarowanych przy pomocy **let** a różnica polega na tym, że wartość raz przypisana **const** nie może zostać już zmodyfikowana.

Za pomocą **var** można definiować zmienne, które są widoczne wewnątrz funkcji nawet po opuszczeniu bloku, w którym zostały zdefiniowane. Zakres tych zmiennych ograniczony jest do funkcji, tzw. **function scope**, lub do przestrzeni globalnej, jeżeli były zadeklarowane poza funkcją.

POJĘCIA

Jakie znasz typy scope?

17

W JavaScript wyróżniamy trzy rodzaje scope:

Global scope	zmienne poza obrębem funkcji
Function scope	zmienne w obrębie funkcji
Block scope	wprowadzony w ES6 ogranicza widoczność do bloku kodu

```
function foo(courseData) {
    let maxStudents = 10;
    var numLessons = 10;
    if (courseData) {
        let maxStudents = 4;  // block scope
        var numLessons = 4;   // function scope
    }
    console.log(maxStudents); // Wyświetla 10
    console.log(numLessons);  // Wyświetla 4
}
```

POJĘCIA

Co to jest dziedziczenie prototypowe?

18

Dziedziczenie prototypowe jest mechanizmem za pomocą którego obiekty dziedziczą właściwości i metody po obiektach - rodzicach. **Prototype** jest obiektem który służy jako szablon podczas tworzenia innych obiektow.

```
function Animal(name) {
    this.name = name;
}
function Dog(name) {
    // Podczas tworzenia Dog, wywoływany jest konstruktor Animal
    Animal.call(this, name);
}
// Dog dziedziczy po Animal
Dog.prototype = Object.create(Animal.prototype);
Animal.prototype.getName = function () {
    return 'Hello, ' + this.name;
}

// Mimo że 'name' jest zapisane w Animal, Dog ma do niego dostęp
var dog = new Dog('Husky');
console.log(dog.getName()); // 'Husky'
```

POJĘCIA

Co to jest hoisting?

19

Hoisting to mechanizm JavaScript przenoszący deklaracje funkcji oraz zmiennych na samą górę, tzn. do zasięgu funkcji, lub globalnego zasięgu.

Przenoszone są **deklaracje** a nie **definicje** funkcji oraz zmiennych.

```
var x = 1;                      // inicjalizacja x
console.log(x + ' ' + y); // '1 undefined'
var y = 2;                      // inicjalizacja y
```

Co dla interpretera jest równoznaczne z:

```
var x = 1;                      // inicjalizacja x
var y;                          // deklaracja y
console.log(x + ' ' + y); // '1 undefined'
y = 2;                          // inicjalizacja y
```

POJĘCIA

Co to jest closure?

20

Closure, czyli domknięcie to funkcja wewnętrzna, która ma dostęp do **a)** zmiennych, które sama definiuje, **b)** zmiennych w scope funkcji zewnętrznej, **c)** zmiennych globalnych.

W przykładzie poniżej funkcja **add10** stanowi domknięcie (funkcję wewnętrzną) funkcji **add** (zewnętrznej).

Pomimo że podczas wywołania funkcji **add10(20)** i **add10(50)** przekazujemy tylko jeden parametr, domknięcie zwraca poprawny wynik działania. Wartość 10 jest w scope funkcji zewnętrznej **add**. Została **zapamiętana** w momencie wywołania **add(10)**, dzięki czemu jest dostępna w domknięciu.

```
const add = x => y => x + y;
const add10 = add(10);

console.log(add10(20)); // 30
console.log(add10(50)); // 60
```

POJĘCIA

Co to jest IIFE?

21

IIFE (Immediately Invoked Function Expression) to funkcja, która jest wykonywana bezpośrednio po jej odczytaniu.

Używa się jej, aby zapobiec wydostawaniu się zmiennych do zewnętrznego *scope*. IIFE może **przyjmować argumenty** oraz **zwracać wynik**:

```
var message = (function () {
    var tempMsg = 'hello';
    return tempMsg;
})();
console.log(message) // 'hello'
```

Zmienna **tempMsg** nie jest dostępna poza scope IIFE.

POJĘCIA

Jak zmienić kontekst wywołania funkcji?

22

Zmiana kontekstu wywołania funkcji polega na wywołaniu metody z jednej klasy na obiekcie innej klasy. Można to zrobić przez podmianę wskaźnika **this** za pomoca funkcji **call**, **apply** lub **bind**.

```
const cat = {
    name: 'Kitty',
    say: function (sound, message) {
        console.log(`${this.name} ${sound} ${message}`);
    }
};
const dog = {
  name: 'Husky'
};
cat.say('meows', 'hello!');              // Kitty meows hello!
cat.say.apply(dog, ['barks', 'apply!']); // Husky barks apply!
cat.say.call(dog, 'barks', 'call!');     // Husky barks call!

const dogFn = cat.say.bind(dog);
dogFn('barks', 'bind!');                 // Husky barks bind!
```

POJĘCIA

Do czego służy use strict?

23

Dyrektywa **use strict** pozwala wymusić ściślejszą analizę i obsługę błędów przez silnik JavaScript w trakcie uruchomienia aplikacji. Sprawia to, że parser jest dużo bardziej rygorystyczny i zgłosi błędy, które bez tej dyrektywy są ignorowane.

Jest to zalecana praktyka w pracy z kodem JavaScript

- ułatwia debugowanie
- zapobiega przypadkowemu tworzeniu zmiennych w global scope
- wyłapuje powtórzone parametry w deklaracji funkcji
- zgłasza błąd przy niepoprawnym użyciu **delete**
- zapobiega utworzeniu niepoprawnych zmiennych, np. ze słowek kluczowym `let private = 1`

POJĘCIA

Object.freeze() vs Object.seal?

24

Obiekt, na którym wywołano funkcję **freeze()** staje się ***immutable*** i nie można zmieniać jego właściwości.

```
const scores = { react: 89, vue: 95, angular: 91 };
Object.freeze(scores);
scores['react'] = 100;        // nie działa
Object.isFrozen(scores);      // true
console.log(scores);          // {react: 89, vue: 95, angular: 91}
```

Natomiast jeżeli na obiekcie wywołano funkcję **seal()**, można zmieniać jego **istniejące** właściwości ale nie można dodawać nowych, ani usuwać istniejących.

```
const scores = { react: 89, vue: 95, angular: 91 };
Object.seal(scores);
delete scores.react;          // nie działa
scores['react'] = 100         // działa!
scores['svelte'] = 88;        // nie działa
Object.isSealed(scores);      // true
console.log(scores);          // {react: 100, vue: 95, angular: 91}
```

POJĘCIA

Czym jest event loop?

25

Event loop jest to mechanizm który wykonuje zadania ze **stosu wywołań** (call stack), **kolejki zadań** (task queue) oraz **kolejki renderowania** (render queue).

Każde użycie funkcji powoduje umieszczenie jej na **stosie wywołań**. Utworzenie callback dla funkcji asynchronicznej, powoduje dodanie go do **task queue**. Natomiast w **render queue** umieszczane są wszystkie operacje, które zmieniają wygląd strony.

Proces działania **Event loop** można sprowadzić do cyklicznego wywoływania:

1. Wykonaj wszystkie zadania ze **stosu wywołań**.
2. Wykonaj wszystkie operacje z **render queue**, które zmieniają coś na stronie (zmiana stylu, przeliczenie wysokości, zmiana DOM).
3. Sprawdź, czy jest coś w **task queue**. Jeżeli jest, sprawdź, czy **call stack** jest pusty. Jeżeli **call stack** jest pusty, weź jedno zadanie z **task queue** i przenieś je na call stack.

POJĘCIA

Czym są Persistent Data Structures?

26

Persistent Data Structures, czyli Trwałe struktury danych są to struktury, które zawsze zachowuje swoje poprzednie wersje, kiedy są modyfikowane. W efekcie są niezmienne - ***immutable***, ponieważ operacje na nich nie powodują zmiany struktury, lecz powstanie nowej, uaktualnionej wersji.

W świecie JavaScript **Persistent Data Structures** najłatwiej tworzyć korzystając z biblioteki **immutable.js**, która nie tylko wspomaga tworzenie i korzystanie z niemodyfikowalnych obiektów, ale również zapewnia memoizację, wygodne API oraz wysoką wydajność przez współdzielenie danych.

```
import { Map } from 'immutable';

const post = Map({ title: 'Angular Exam' });
const updatedPost = post.set({ title: 'React Exam!' });
console.log(post.get('title'));                  // Angular Exam
console.log(updatedPost.get('title'));           // React Exam
```

POJĘCIA

Jak stworzyć namespace?

27

Namespace, czyli przestrzeń nazw można w JavaScript zasymulować przykładowo korzystając z **IIFE** (Immediately Invoked Function Expression)

```
var message = (function () {
    var tempMsg = 'hello';
    return tempMsg;
})();
console.log(message)  // 'hello'
```

lub z **object literal**, czyli inicjalizatora obiektu.

```
var message = {};
message.tempMsg = 'hello';
message.print = function() {
    console.log(message.tempMsg);
}
message.print();      // 'hello'
```

POJĘCIA

Jak debugować kod JavaScript?

28

Najpopularniejsze sposoby debugowania kodu JavaScript to:

- Chrome Devtools: zakładka `Source` + breakpoints
- Wyrażenia `debugger`
- Dodawanie w kodzie

`console.log` - do sprawdzania zawartości zmiennych

`console.time` - do mierzenia czasu wykonania kodu

`console.table` - do sprawdzania zawartości tablic

- Debugger wbudowany w IDE, np. WebStorm
- Do debugowania requestów HTTP – Postman
- Do kontrolowania zmian w HTML – inspektor wbudowany w przeglądarkę

POJĘCIA

Jak zabezpieczyć obiekt przed rozszerzaniem?

29

Aby zabezpieczyć obiekt przed rozszerzaniem można skorzystać z funkcji:

`Object.preventExtensions()` - sprawi, że nie będzie możliwe dodanie nowych właściwości do obiektu; możliwe będzie za to zmiana lub usunięcie istniejących

`Object.seal()` - sprawi, że możliwa będzie jedynie zmiana już istniejących właściwości; dodanie nowych lub usunięcie istniejących nie powiedzie się

`Object.freeze()` - sprawi, że nie będzie możliwa żadna zmiana istniejących właściwości; dodanie nowych lub usunięcie również się nie powiedzie.

POJĘCIA

Co różni funkcje synchroniczne od asynchronicznych?

30

Funkcje synchroniczne są wykonywane w momencie ich uruchomienia. Interpreter czeka na ich skończenie zanim uruchomi kolejne linie kodu. Funkcje synchroniczne blokują renderowanie UI aż do ich skończenia.

Funkcje asynchroniczne są również wykonywane w momencie ich uruchomienia, ale interpreter nie czeka na jej wykonanie, tylko zostaje uruchomiona kolejna linia kodu. Funkcje asynchroniczne zazwyczaj są funkcjami korzystającymi z operacji I/O, np. żądania HTTP, odczyt plików, obsługa UI.
W JavaScript jest wiele możliwości obsługi funkcji asynchronicznych, np.

- Callbacks
- Promises
- Async/await

FUNKCJE

Jak działają funkcje map, filter i reduce?

31

Metoda **map** tworzy nową tablicę z rezultatem wykonania wskazanej funkcji na każdym elemencie podanej tablicy:

```
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(number => Math.pow(number, 2));
console.log(squared); // [1, 4, 9, 16, 25]
```

Metoda **filter** tworzy nową tablicę z wszystkimi elementami, które spełniają warunek określony w **predykacie**:

```
const odds = numbers.filter(number => number % 2 !== 0);
console.log(odds);   // [1, 3, 5]
```

Metoda **reduce** ma na celu **redukcję** całej tablicy do pojedynczej wartości. Odbywa się to przez wywołanie funkcji i przekazanie jej zakumulowanego rezultatu z poprzednich wywołań i kolejnego elementu tablicy (od lewej do prawej).

```
const sum = numbers.reduce((total, current) => total+current, 0);
console.log(sum); // kroki: 0+1=1; 1+2=3; 3+3=6; 6+4=10; 10+5=15
```

FUNKCJE

Jaka jest różnica między forEach() a map()?

32

Metoda **forEach** wykonuje przekazaną funkcję jeden raz na każdy element tablicy.

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(number => {
    const squared = Math.pow(number, 2);
    console.log(`${number}^2 = ${squared}`);
});
```

Metoda **map** tworzy nową tablicę z rezultatem wykonania wskazanej funkcji na każdym elemencie podanej tablicy

```
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(
   number => Math.pow(number, 2)
);
console.log(squared); // [1, 4, 9, 16, 25]
```

FUNKCJE

Czym są first class functions?

Funkcje pierwszej klasy - **first class functions** - posiada język programowania, w którym są one traktowane jak każda inna zmienna.

W świecie JavaScript wszystkie funkcje są funkcjami pierwszej klasy, ponieważ można je

- przekazywać jako argumenty wywołania innych funkcji,
- zwracać jako rezultat działania funkcji
- przypisywać do zmiennych zupełnie jak każdą inną wartość

W przykładzie poniżej **isEven** jest funkcją, którą możemy przekazać jako parametr metody **filter**:

```
const numbers = [1, 2, 3, 4, 5];
const isEven = number => number % 2 === 0;
numbers.filter(isEven);   // [2, 4]
```

FUNKCJE

Czym są funkcje wyższego rzędu?

34

Funkcje wyższego rzędu - **higher order functions** - to funkcje, które operują na innych funkcjach. Przykładowo:

- przyjmują funkcję jako argument wywołania, lub
- zwracają funkcje jako wynik swojego działania

Przykładami często używanych funkcji wyższego rzędu są: **map**, **filter** i **reduce**.

W przykładzie poniżej **greaterThan** jest funkcją wyższego rzędu, która przyjmuje *limit* a zwraca inną funkcję, sprawdzającą czy przekazana liczba jest większa od limitu:

```
const greaterThan = limit =>
        number => number > limit;

const greaterThan10 = greaterThan(10);
console.log(greaterThan10(20));  // true
```

FUNKCJE

Co to jest pure function?

35

Pure function to funkcja, która ma poniższe cechy:

- jest **deterministyczna** - dla tych samych parametrów wejściowych zawsze zwraca ten sam wynik,
- wynik zależy tylko od parametrów wywołania
- nie modyfikuje zewnętrznych wartości ani zmiennych poza swoim zakresem
- nie ma efektów ubocznych - ***side effects*** - takich jak manipulacja DOM, zapytania HTTP, operacje I/O

Przykładowo funkcja **push()** dodająca element do tablicy **nie jest pure function**, ponieważ modyfikuje tablicę i kolejne wywołania dają inne rezultaty.

Z kolei funkcja **concat()** łącząca tablice w jedną, **jest pure function**, ponieważ nie modyfikuje istniejących tablic tylko tworzy nową zawierającą połączone tablice.

BROWSER

Czym jest event bubbling?

36

Event bubbling jest typem propagacji zdarzeń, gdzie zdarzenia obsługiwane są w pierwszej kolejności przez element **najbardziej wewnętrzny**, a dalej przez kolejne **parent** elementy w hierarchii aż do najbardziej zewnętrznego elementu drzewa **DOM**.

Metoda **addEventListener** pozwala zarejestrować listenera zdarzeń określonego typu. Trzecim parametrem tej funkcji jest **useCapture**, które kontroluje czy korzystamy z **event bubbling** (domyślne) czy **event capturing**.

Event bubbling możemy przerwać poprzez funkcję **event.stopPropagation()**.

```
element.addEventListener('click', handle, false);
// useCapture = false, obsługa bubbling
```

BROWSER

Czym jest event capturing?

37

Event capturing jest typem propagacji zdarzeń, gdzie zdarzenia obsługiwane są w pierwszej kolejności przez element **najbardziej zewnętrzny**, a dalej przez kolejne **child** elementy w hierarchii aż do najbardziej wewnętrznego elementu **DOM**.

Metoda **addEventListener** pozwala zarejestrować listenera zdarzeń określonego typu.

Trzecim parametrem tej funkcji jest **useCapture**, które kontroluje czy korzystamy z **event bubbling** (domyślne) czy **event capturing**.

```
element.addEventListener('click', handler, true);
// useCapture = true, obsługa capturing
```

BROWSER

Na czym polega delegacja zdarzeń?

38

Event bubbling oraz **event capturing** pozwalają na implementację ważnego wzorca obsługi zdarzeń zwanego delegacją zdarzeń - **event delegation**.

Jeżeli mamy wiele elementów drzewa **DOM** obsługiwanych w ten sam sposób, to zamiast tworzyć *handler* dla każdego z tych elementów, tworzymy jeden *handler*, do którego będziemy **delegować** obsługę zdarzeń wszystkich elementów.

Przykładowo, zdarzenie *click* na elemencie **<li>** delegujemy do *handlera* umieszczonego na **<ul>**. Informacja, o tym który element **<li>** został kliknięty uzyskujemy przez **event.target**.

```
const form = document.querySelector('#signin');
form.addEventListener('input', function (event) {
    console.log(event.target);
}, false);
```

BROWSER

Do czego służy obiekt *history*?

39

Obiekt *window.history* zawiera historię przeglądanych stron. Jest używany do przełączania się **wstecz** lub **dalej** pomiędzy stronami odwiedzanymi przez użytkownika. Obkekt posiada kilka przydatnych metod:

- `history.back()` - przejście wstecz
- `history.forward()` - przejście dalej
- `history.go(number)` - przejście dalej o zadaną liczbę stron lub wstecz (jeśli liczba ujemna)
- `history.pushState` - przydatne, gdy chcemy zmienić URL bez przeładowania strony
- `history.replaceState` - zmienia ostatni wpis

BROWSER

Jaka jest różnica między attribute a property?

40

Attributes są zdefiniowane na poziomie **HTML** a **properties** na poziomie drzewa **DOM**. Używamy **properties** w kodzie JavaScript, aby modifikować wartości **atrybutów** HTML.

Przykładowo, **<input>** posiada **atrybut** value typu string, którego wartość wynosi 0:

```
<input type="text" value="0">
```

Korzystając z JavaScript możemy odczytać wartość **property:**

```
const input = document.querySelector('input');
console.log(input.value);     // '0'
```

Modyfikując wartość **property** po stronie JavaScript, wartość **attribute** w HTML **nie ulegnie zmianie**. Aby zmodyfikować wartość po stronie HTML, należy użyć:

```
input.setAttribute('value', 9);
```

STORAGE

Czym jest web storage?

41

Web storage to API zapewniające mechanizmy, dzięki którym przeglądarki mogą zapisywać i odczytywać rekordy w postaci par klucz-wartość w sposób znacznie bardziej przyjazdy niż w przypadku obsługi plików *cookies*.

Web storage zapewnia dwa podstawowe mechanizmy przechowywania danych:

`Session storage` - przechowuje dane w czasie trwania sesji - dane zostaną skasowane, jeśli przeglądarka zostanie zamknięta

`Local storage` - przechowuje dane również po zamknięciu przeglądarki - dane nie zostaną skasowane automatycznie, ale będą przechowywane do czasu usunięcia przez użytkownika, bądź za pomocą JavaScript

STORAGE

Jak skorzystać z web storage?

Web storage jest dostępny za pomocą obiektu ***window***, poprzez properties **localStorage** oraz **sessionStorage**, które udostępniają obiekty **WindowLocalStorage** oraz **Window-SessionStorage**.

Za pomocą tych obiektów można tworzyć, odczytywać, zapisywać i usuwać dane w pamięci przeglądarki, przykładowo:

```
const courses = ['React', 'Vue', 'Angular'];
localStorage.setItem('courses', courses);

const saved = localStorage.getItem('courses');
console.log(saved); //['React', 'Vue', 'Angular']

localStorage.removeItem('courses');
localStorage.clear();
```

STORAGE

Czy można odczytać *localStorage* innej domeny?

Nie ma możliwości odczytania danych zapisanych w **localStorage** albo **sessionStorage** przez inną domenę.

Przed dostępem do danych zapisanych przez inną domenę chroni nas mechanizm ***Same Origin Policy***.

Aby uzyskać dostęp do danych z **web storage** muszą się zagadzać: protokół, host i port.

Przykładowo do danych zapisanych przez

```
https://store.company.com/page.html
```

można uzyskać dostęp z (inna ścieżka)

```
https://store.company.com/checkout/page.html
```

natomiast nie można z (inny protokół)

```
http://store.company.com/page.html
```

STORAGE

Co różni cookies, sessionStorage i localStorage?

Cookies	**localStorage**	**sessionStorage**
Obsługiwane po stronie klienta i serwera	Obsługiwane tylko po stronie klienta	Obsługiwane tylko po stronie klienta
Czas życia konfigurowany przez parametr *Expires*	Dane czyszczone *ręcznie* lub z poziomu JavaScript	Dane czyszczone po zamknięciu przeglądarki (lub zakładki)
Maksymalny rozmiar 4KB	Maksymalny rozmiar do 10MB	Maksymalny rozmiar do 10MB

STORAGE

Jakie metody są dostępne w sessionStorage?

45

sessionStorage udostępnia następujące metody

`setItem(key, value)`	- zapisuje nową wartość lub aktualizuje istniejącą
`getItem(key)`	- odczytuje wartość
`removeItem(key)`	- usuwa klucz i wartość
`clear()`	- czyści wszystkie klucze

```
const courses = ['React', 'Vue', 'Angular'];
sessionStorage.setItem('courses', courses);

const saved = localStorage.getItem('courses');
console.log(saved); //['React', 'Vue', 'Angular']

localStorage.removeItem('courses');
localStorage.clear();
```

ASYNC

Co to jest Promise?

46

Promise to obiekt reprezentujący wynik działania operacji asynchronicznej. Może występować w trzech różnych stanach: **fulfilled** (zakończone), **rejected** (odrzucone) albo **pending** (w trakcie).

Zadanie asynchroniczne może zakończyć się:

- powodzeniem – wtedy wywoływana jest metoda **resolve()**
- porażką – wtedy wywoływana jest **reject()**

```
function asyncOperation() {
    return new Promise((resolve, reject) => {
        // zadanie asynchroniczne...
        resolve(/* result */);
    });
}
```

ASYNC

Co to jest callback?

47

Callback jest funkcją, którą przekazano jako argument do innej funkcji i która zostanie wywołana z jej wnętrza, aby zasygnalizować ukończenie jakiegoś działania, na przykład nadejście odpowiedzi z REST API za pomocą **fetch()**.

Callback jest wykorzystywany, aby umożliwić przetwarzanie wyników działania operacji **asynchronicznej**:

```
function greetingsCallback(name) {
  console.log('Hello ' + name);
}
function emptyNameCallback() {
  console.error('Cannot process empty value!');
}
function processUserName(callback, errorCallback) {
  var name = prompt('Please enter your name.');
  if(!name) errorCallback();
  else callback(name);
}

processUserName(greetingsCallback, emptyNameCallback);
```

ASYNC

Czym jest callback hell?

48

Callback hell jest jednym z częściej występujących antywzorców. Występuje w kodzie, w którym przeplata się wiele zagnieżdżonych callbacków co czyni kod trudnym do czytania, utrzymywania i podatnym na błędy.

Pojawia się, gdy w kodzie wykonywanych jest wiele operacji asynchronicznych, które muszą nastąpić kolejno po sobie:

```
function findUserMessages(username, callback) {
  fetchUserData(username, user => {
    fetchUserMessages(user.userId, messages => {
      messages.forEach(message => {
        getContent(message.uuid, content => {
          callback(content);
        })
      });
    })
})}
```

ASYNC

Jakie są korzyści w korzystaniu z Promise?

49

Dzięki korzystaniu z **Promise** kod jest czytelniejszy i łatwiejszy do utrzymania poprzez:

- Pozbycie się **callback hell**
- Łączenie kolejnych wywołań funkcji asynchronicznych za pomocą metody **.then()**
- Łączenie równoległych wywołań funkcji asynchronicznych za pomocą metody **Promise.all()**
- Łatwiejszą obsługa wyjątków i niepoprawnych odpowiedzi w **.catch()**

Kod uzyskany dzięki zastosowaniu Promise będzie prostszy niż w przypadku callback.

W niektórych sytuacjach może się okazać konieczne stosowanie **polyfill** podczas pracy z legacy kodem.

ASYNC

Co to jest async/await?

50

Async/await to prostszy sposób zapisu i obsługi funkcji asynchronicznych, zbudowany w oparciu o **Promise**.

Słowo kluczowe **async** przed deklaracją funkcji sprawia, że będzie domyślnie zwracała **Promise**.

Słowo kluczowe **await** może wystąpić tylko w ramach funkcji **async**, w przeciwnym razie zostanie zwrócony wyjątek.

await sparawia że operacja asynchroniczna po prawej stronie wyrażenia (najczęściej Promise) musi się zakończyć zanim przetworzona zostanie dalsza część funkcji.

```
async function findUserMessages(username) {
    const user = await findUserData(username);
    const messages = await findMessages(user.id);
    return messages;
}
```

JAVASCRIPT

Jak można stworzyć tablicę?

51

Standardowo nową tablicę możemy w JavsScript utworzyć na co najmniej 5 różnych sposobów:

```
// sposób podstawowy
const courses = ['React', 'Angular', 'Vue'];

// za pomocą new - tworzy Array o danej długości
const names = new Array(5);

// za pomocą Array.of
const animals = Array.of('Tiger', 'Shark');

// na bazie innej tablicy za pomocą Array.from
const wildAnimals = Array.from(animals);

// za pomocą operatora spread
const jsCourses = [...courses];
```

JAVASCRIPT

Jak sklonować obiekt w JavaScript?

52

```
// sposób podstawowy
const reactCourse = { title: 'React' };
const cloned = Object.assign({}, reactCourse);
```

Najprostszym sposobem na sklonowanie obiektu jest zastosowanie **Object.assign**. Ten sposób jednak wykonuje **shallow copy**, czyli nie kopiuje referencji do innych obiektów. Obiekty **angular** i **cloned** odwołują się do tego samego obiektu **react**

```
const react = { title: 'React' };
const angular = { title: 'Angular', next: react };
const cloned = Object.assign({}, angular);
react.title = 'Vue';
console.log(cloned.title) // 'Vue'
```

Aby wykonać **deep copy**, można skorzystać z **lodash**:

```
const cloned = _.cloneDeep(angular);
```

JAVASCRIPT

Czym jest JSON i jak go obsługiwać?

53

JSON, czyli JavaScript Object Notation jest formatem tekstowym bazującym na składni JavaScript. Jest przydatny do **przesyłania danych przez sieć** oraz zapisywania ich do baz danych lub plików.

W JavaScript obiekt JSON posiada dwie metody:

`JSON.stringify()` - zamienia obiekt na string

`JSON.parse()` - zamienia string na obiekt i rzuca SyntaxError w przypadku błędów

```
const react = { title: 'React' };
const data = JSON.stringify(react);
console.log(data);      // "{"title":"React"}"

const retrieved = JSON.parse(data);
console.log(retrieved); // { title : "React" }
```

JAVASCRIPT

Jak odczytać dane zapisane w JSON?

54

Aby odczytać dane zapisane w JSON, należy użyć metody **JSON.parse()**. Jeżeli dane nie są poprawnym JSON-em, metoda rzuci wyjątek **SyntaxError**.

```
const userData = '{"name": "Bob", "age": 26}';
const user = JSON.parse(userData);
console.log(user.name); // "Bob"
```

JSON.parse() jest szczególnie przydatna podczas odczytywania danych otrzymanych w wyniku zapytania HTTP. Dane z serwera **przychodzą zawsze jako string**.

JAVASCRIPT

Jaki jest efekt użycia *setTimeout* z wartością 0?

55

Kiedy funkcja **setTimeout()** zostanie wywołana z drugim parametrem ustawionym jako 0, silnik JavaScript wykona wskazaną funkcję *tak szybko jak to tylko możliwe*, choć nie od razu. Wywołanie funkcji zostanie umieszczone w kolejce zdarzeń **eventQueue** i zostanie wykonane w kolejnym cyklu.

Przykładowo, wyrażenia **(1)** oraz **(4)** zostaną wykonane od razu w kolejności utworzenia, natomiast **(2)** oraz **(3)** zostaną umieszczone w kolejce zdarzeń. Kiedy silnik JavaScript zakończy przetwarzanie, pobierze następne zadanie z kolejki **(3)** i je obsłuży.

```
console.log(1);
setTimeout(() => console.log(2), 1000);
setTimeout(() => console.log(3), 0);
console.log(4);
// 1, 4, 3 ... po sekundzie ..., 2
```

JAVASCRIPT

Po co stosujemy *Object.seal()*?

56

Zastosowanie **Object.seal()** sprawia, że do obiektu **nie można dodać** żadnych nowych właściwości. Nie można również **usunąć** żadnej z już istniejących. Próba wykonania takiej czynności nie powiedzie się, chociaż nie zostanie rzucony żaden wyjtek.

Można za to **modyfikować** istniejące właściwości obiektu.

```
const user = { name: 'Bob', age: 26 };
Object.seal(user);

Object.isSealed(user);// true
user.name = 'Alice';  // można zmienić property
user.city = 'London'; // nie można dodać
delete user.age;      // nie można usunąć
console.log(user);    // {name:"Alice",age:26}
```

JAVASCRIPT

Po co stosujemy *Object.freeze()*?

57

Zastosowanie **Object.freeze()** sprawia, że obiekt staje się ***immutable*** i nie można zmieniać jego właściwości, dodawać nowych, ani usuwać istniejących. W prównaniu do **Object.seal()**, nie można zmieniać wartości properties.

Aby sprawdzić, czy obiekt jest ***zamrożony*** można skorzystać z funkcji **Object.isFrozen()**.

```
const user = { name: 'Bob', age: 26 };
Object.freeze(user);
Object.isFrozen(user);// true
user.name = 'Alice';  // nie można zmienić
user.city = 'London'; // nie można dodać
delete user.age;      // nie można usunąć
console.log(user);    // {name:"Alice",age:26}
```

JAVASCRIPT

Co różni Object.values od Object.entries?

58

Obie funkcje możemy wykorzystać do iterowania po właściwościach obiektu w JavaScript:

`Object.values()` zwraca tablicę zawierającą wartości wszystkich właściwości tekstowych obiektu.

`Object.entries()` zwraca tablicę zawierającą pary *klucz-wartość* każdej właściwości tekstowej obiektu.

```
const user = { name: 'Bob', age: 26 };

Object.values(user);
// ["Bob", 26]

Object.entries(user);
// [["name", "Bob"], ["age", 26]]
```

JAVASCRIPT

Jaka jest różnica między for...in a for...of?

59

Obie konstrukcje możemy wykorzystać do iterowania po właściwościach obiektu w JavaScript:

- `for..in` iteruje po wszystkich kluczach właściwości obiektu. W przypadku tablic, kluczem są indeksy kolejnych elementów tablicy.
- `for..of` iteruje po wartościach obiektu, elementach tablicy, znakach w stringach.

```
const courses = ['Angular', 'React', 'Vue'];
courses.level = 'advanced';
for(const key in courses) {
  console.log(key); // 0, 1, 2, level
}
for(const val of courses) {
  console.log(val); // 'Angular', 'React', 'Vue'
}
```

JAVASCRIPT

Jak sprawdzić czy obiekt posiada *property*?

60

Do sprawdzenia, czy obiekt posiada *property* w postaci klucza, można skorzystać z jednej z trzech metod:

1. Zastosowanie operatora ***in***
2. Zastosowanie funkcji ***hasOwnProperty***
3. Porównanie z ***undefined***

```
const user = { name: 'Bob', age: 26 };
"city" in user                    // false
"age"  in user                    // true

user.hasOwnProperty("city")       // false
user.hasOwnProperty("age")        // true

user.city !== undefined           // false
user.age  !== undefined           // true
```

POJĘCIA

Jakie są korzyści stosowania modułów?

61

Dzięki stosowaniu modułów otrzymujemy kod, który:

- jest łatwiejszy w **utrzymaniu**
- można używać **wielokrotnie** w różnych kontekstach
- zapewnia interfejs modułu ze światem zewnętrznym
- zapobiega wydostawaniu się zmiennych do ***global scope***

```
// scoring.js
export function hasFailed(results, minScore=.9) {
  return results.some(score => score < minScore);
}

// course.js
import { hasFailed } from './scoring.js'

const results = [.79, .65, .89, .91];
console.log(hasFailed(results));          // true
```

POJĘCIA

Czym jest call stack?

62

Call stack czyli stos wywołań jest strukturą w silniku JavsScript, która przechowuje informacje o wywołaniach funkcji. Gdy funkcja jest wywoływana, zostaje umieszczona na stosie i pozostaje tam aż do zakończenia obsługi.

Przykładowo wywołując funkcję **calculateTotal()**

1. Umieść **calculateTotal()** na szczycie stosu i wykonaj
2. Dodaj **getTaxRate()** na szczyt stosu i wykonaj
3. Po zwróceniu podatku usuń **getTaxRate()** ze stosu
4. Po obliczeniu wyniku usuń **calculateTotal()** ze stosu

```
function calculateTotal(units, price) {
    return getTaxRate() * units * price;
}
function getTaxRate() {
    return 0.19;
}
```

POJĘCIA

Co to jest memory heap?

63

Memory heap to obszar pamięci, w którym przechowywane są **obiekty** gdy przypisujemy je do zmiennych. Innymi słowy jest to miejsce gdzie silnik JavaScript tworzy i usuwa obiekty, gdy nie są już potrzebne.

Aby sprawdzić co aktualnie znajduje się w **heap** można skorzystać z Dev Tools i zakładki memory.

Można również skorzystać z interfejsu **Performance**, który w Chrome udostępnia m.in. obiekt **memory** informujący o aktualnie używanej i dostępnej pamięci (w bajtach).

```
// użyta pamięć na bieżącej stronie
performance.memory.usedJSHeapSize
// pamięć zaalokowana, "zarezerwowana" dla strony
performance.memory.totalJSHeapSize
// maksimum pamięci jakie może być dostępne
performance.memory.jsHeapSizeLimit
```

POJĘCIA

Co to jest polyfill?

64

Polyfill - *szpachlówka*, *wypełniacz* - to kod JavaScript, który jest używany aby udostępnić **najnowsze możliwości** języka starszym przezglądarkom, które jeszcze ich nie wspierają.

Przykładowo, za pomocą polyfilli można uruchamić kod napisany w **ES6** na starszych wersjach Internet Explorera.

Z drugiej strony, dzięki polyfillom można wykorzystywać najnowsze funkcje języka, które nie weszły jeszcze nawet do **oficjalnej specyfikacji** a są jedynie propozycją. Dopóki przeglądarki nie będą wspierać tych funkcjonalności, konieczne będzie korzystanie z polyfilli.

Częstą praktyką jest korzystanie z biblioteki **Babel** czyli transpilatora, który *tłumaczy* kod JavaScript na starsze wersje standardu języka JavaScript.

POJĘCIA

Na czym polega tree shaking?

65

Tree shaking jest mechanizmem usuwania nieużywanego kodu. Oznacza, że podczas procesu budowania do powstającej paczki kodu **nie zostaną włączone moduły** JavaScript, które nie są używane.

Tree shaking został rozpowszechniony razem z popularyzacją **ES6** i jest oparty na wprowadzonym wtedy standardzie dotyczącym **modułów**.

Ma ogromne znaczenie w zmniejszaniu **rozmiaru aplikacji**. Im mniejsza jest aplikacja, tym mniej danych trzeba przesłać do klienta i krótszy czas oczekiwania na jej załadowanie.

Przykładowo prosta aplikacja **Angular** zajmuje kilka MB, jednak dzięki zastosowaniu tree shaking można jej rozmiar zmniejszyć do kilkuset kB.

POJĘCIA

Czym jest łańcuch prototypów?

66

Każdy **obiekt** posiada prywatną własność łączącą go z innym obiektem zwanym jego **prototypem**. Obiekt prototypu posiada swój własny prototyp, i tak dalej aż obiekt osiągnie **null** jako swój prototyp. **null** nie ma prototypu i działa jak zakończenie **łańcucha prototypów**.
Przykładowo, szukając dowolnej właściwości w obiekcie **react**, sprawdzony zostanie jego prototyp **Course**, następnie prototyp **Object**. Prototypem **Object** jest **null**. W tym miejscu łańcuch się kończy.

```
function Course(name) {
    this.name = name;
}
const react = new Course('react');
react.__proto__                  // Course.prototype
react.__proto__.__proto__        // Object.prototype
react.__proto__.__proto__.__proto__ // null
```

POJĘCIA

Jaka jest różnica między *__proto__* a *prototype*?

67

__proto__ jest jednym z *property* (właściwości) obiektu, wskazującym na jego **prototyp**. Można za pomocą niego przechodzić *w górę* po łańcuchu prototypów. W tym celu można również skorzystać z **Object.getPrototypeOf(o)**.

Z kolei **prototype** jest właściwością obiektu funkcji, wskazującym na prototyp jaki będą miały obiekty utworzone na bazie tej funkcji za pomocą operatora **new**.

```
function Course(name) {
    this.name = name;
}
const course = new Course('react');
course.__proto__ === Course.prototype
course.__proto__.__proto__ === Object.prototype
Course.__proto__ === Function.prototype
course instanceof Course
course instanceof Object
```

POJĘCIA

Czym są klasy ES6?

68

Klasy zostały wprowadzone jako element JavaScript w ES6. Jednak jest to tylko lukier składniowy (*syntactic sugar*) dla istniejącego, opartego na **prototypach** modelu dziedziczenia. Składnia klas nie wprowadza nowego zorientowanego obiektowo modelu dziedziczenia.

Klasy wprowadzają za to znacznie prostszą i bardziej czytelną **składnię** do tworzenia obiektów i dziedziczenia.

```
class Course {
    constructor(name) {
        this.name = name;
    }
    getDetails() {
        return 'I love ' + this.name;
    }
}
```

POJĘCIA

Na czym polega obfuskacja?

69

Obfuskacja jest procesem tworzenia kodu JavaScript w taki sposób, aby utrudnić jego odczytanie przez innych. Ma na celu:

- Utrudnienie odczytania kodu - ogranicza *reverse engineering*, choć go nie uniemożliwia
- Ukrycie logiki biznesowej i sposobu działania aplikacji
- Zmniejszenie rozmiaru kodu źródłowego
- Przyspieszenie pobierania aplikacji przez klientów

Obfuskacja kodu nie oznacza jednak, że kod zostanie **zaszyfrowany**, tzn. że nie da się go odczytać bez posiadania klucza. Dlatego nie należy polegać na obfuskacji jako metodzie ukrycia najważniejszych informacji, kluczy czy sekretów.

POJĘCIA

Na czym polega minifikacja?

70

Minifikacja jest to proces kasowania z kodu (JavaScript, HTML, CSS lub innego) zbędnych znaków, np. spacji, pustych linii, odstępów czy komentarzy. Zachowuje się przy tym **poprawność** i działanie kodu.

Dodatkowo zamieniane są także **nazwy zmiennych** na literki alfabetu tak, aby jeszcze bardziej ograniczyć wielkość ciągu. Ma to na celu:

- Poprawę szybkości ładowania stron i aplikacji
- Ograniczenie transferu danych
- Zmniejszenie rozmiaru kodu źródłowego

Przed minifikacją kodu zalecane jest połączenie wszystkich plików w jeden i minifikacja całości.

FUNKCJE

Na czym polega memoizacja?

Memoizacja jest techniką poprawy wydajności aplikacji poprzez zapisywanie w ***cache*** wyników działania kosztownej funkcji. Przy każdym wywołaniu funkcji z parametrami sprawdzany jest ***cache***. Jeśli zapisany jest w nim wynik dla tych parametrów, można go zwrócić bez obliczania na nowo.

```
function memoize(func) {
    const cache = {};
    return value => {
        if(cache[value]) {
            return cache[value];
        }
        const result =  func(value);
        cache[value] = result;
        return result;
    }
}
const capitalize = str => str.toUpperCase();
const memoizedCap = memoize(capitalize);
console.log(memoizedCap('aaa')); // Zapis do cache
console.log(memoizedCap('aaa')); // Odczyt z cache
```

FUNKCJE

Czym jest currying?

Currying jest procesem polegającym na zmianie krotności funkcji. Innymi słowy, currying zmienia funkcj przyjmującą **wiele parametrów** w sekwencję funkcji, z których każda przyjmuje tylko **jeden parametr**.

Dzięki temu uzyskujemy funkcje, który są czytelne, reużywalne i które można składać z innymi. Przykładowo, funkcję **add5** można stosować wielkrotnie z różnymi parametrami.

```
const adder = (a, b) => a + b
const curriedAdder = a => b => a + b

console.log(adder(5, 7));       // 12

const add5 = curriedAdder(5);   // currying
console.log(add5(7));           // 12
console.log(add5(100));         // 105
```

FUNKCJE

Na czym polega partial application?

Partial application polega na zmianie funkcji przyjmującej wiele parametrów w funkcję, która przyjmuje **mniejszą** liczbę parametrów. Przekazane parametry będą *związane* z nową funkcją, dzięki czemu nie trzeba ich za każdym razem podawać.

Stosując technikę partial application można zaimplementować **currying**.

```
const getUrl = (hostname, resource, id) =>
  `https://${hostname}/${resource}/${id}`;

// partial application (hostname)
const getResourceUrl = (resource, id) =>
  getUrl('https://some.api', resource, id);

// https://some.api/users/123
console.log(getResourceUrl('users', 123));
```

FUNKCJE

Czym są *arrow functions*?

74

Arrow functions - funkcje strzałkowe - to skrócony sposób zapisu funkcji wprowadzony od ES6. Funkcje takie mają dodatkowe cechy ułatwiające pisanie kodu: nie posiadają swojego własnego **this**, **arguments** ani **super**.

Ponieważ nie posiadają własnego **this**, nie mogą być wykorzystane jako konstruktory.

Najczęściej są wykorzystywane do tworzenia funkcji anonimowych lub znajdujących się poza obiektem.

```
// można pominąć () przy jednym parametrze
const capitalize = str => str.toUpperCase();
const adder = (a, b, c) => a + b + c;

// błąd! zapisze name w global scope
const Dog = name => this.name = name;
new Dog('Spark'); // Dog is not a constructor!
```

FUNKCJE

Co to funkcje anonimowe?

Funkcje anonimowe są to funkcje, które nie posiadają nazwy. Najczściej są wykorzystywane w sytuacjach, w których nazwa funkcji nie ma znaczenia.

Można je przypisać do zmiennych, używać jako *callback* lub przekazywać jako parametr podczas wywołania *higher order function* np. `map`, `filter` lub `reduce`.

Poniżej przykłady funkcji anonimowych: 1) klasycznej, 2) zapisanej jako arrow function, 3) przypisanej do zmiennej.

```
const courses = ['react', 'angular'];
courses.map(function (course) {                    // 1
    return course.toUpperCase();
});
courses.map(course => course.toUpperCase()); // 2
const capitalize = str => str.toUpperCase(); // 3
courses.map(capitalize);
```

BROWSER

Do czego służy preventDefault()?

76

`Event.preventDefault()` jest funkcją, której wywołanie powoduje przerwanie domyślnej obsługi zdarzenia. Oznacza to, że **domyślna** akcja przypisana zdarzeniu nie zostanie wykonana, przykładowo, nie zostanie przesłany formularz po zatwierdzeniu go przyciskiem *submit*, natomiast nie zostanie przerwana **propagacja** tego zdarzenia.

Wywołanie tej metody na zdarzeniu, które nie jest *cancelable*, np. utworzone przez `EventTarget.dispatchEvent()` bez podawania `cancellable:true` nie przyniesie efektów.

```
const user = document.getElementById('username');
user.addEventListener('keypress', function(evt) {
  const charCode = evt.charCode;
  if (charCode < 97 || charCode > 122) {
    evt.preventDefault();    // tylko małe litery
  }
}, false);
```

BROWSER

Do czego służy stopPropagation()?

77

`Event.stopPropagation()` jest funkcją, której wywołanie powoduje przerwanie propagacji zdarzenia niezależnie czy jest to **bubbling**, czy capturing.

Wywołanie tej metody nie spowoduje jednak, że domyślne akcje przypisane do zdarzenia nie zostaną wykonane. Aby im zapobiec, należy użyć `Event.preventDefault()`.

```
<table id="table" onclick="alert('hello!');">
  <tr id="tablerow">
    <td id="tablecell">click here!</td>
  </tr>
</table>
```

```
const elem = document.getElementById('tablecell');
elem.addEventListener('click', function (evt) {
    evt.stopPropagation();
}, false);
```

BROWSER

Jak zmienić URL bez przeładowania strony?

78

Aby zmienić URL należy skorzystać z interfejsu **History**, który pozwala na modyfikowanie historii sesji przeglądarki.

Posiada on metody `pushState()` oraz `replaceState()`, które pozwalają na dodawanie oraz modyfikowanie wpisów historii przeglądarki, również dotyczących bieżącej strony.

```
const state = { 'page_id': 1, 'user_id': 5 }
const title = 'Hello World!'
const url = 'hello-world.html'

history.pushState(state, title, url)
```

Zmiana url jest możliwa również *starym* sposobem, poprzez `window.location.href`, ale oprócz zmiany URL spowoduje to także przeładuje stronę.

BROWSER

Jak wywołać redirect?

79

Aby przekierować użytkownika na inną stronę i wywołać **redirect** można skorzystać z

- interfejsu **Location**, pozwalającego na zmiany URL, lub
- interfejsu **History**, służącego do tworzenia i modyfikowania historii przeglądarki.

```
// zachowanie podobne do HTTP redirect
window.location.replace('https://www.fiszkijs.pl')
// zachowanie podobne do załadowania strony
window.location.assign('https://www.fiszkijs.pl')
// zachowanie podobne do kliknięcia w link
window.location.href = 'https://www.fiszkijs.pl'

// zachowanie jak przy kliknięciu 'Wstecz'
window.history.back()
// można cofać historię o dowolną liczbę
window.history.go(-1)
```

BROWSER

Jak odczytać adres strony w JavaScript?

80

Aby odczytać adres strony z poziomu JavaScript, należy skorztystać z interfejsu **Location**, przechowującego informacje o URL.

```
const url = window.location;

// https://fiszkijs.pl:8080/pl-PL/search?q=URL#js
url.href

url.origin      // https://fiszkijs.pl:8080
url.protocol    // https:
url.host        // fiszkijs.pl:8080
url.hostname    // fiszkijs.pl
url.port        // 8080
url.pathname    // /pl-PL/search
url.search      // ?q=URL
url.hash        // #js
```

BROWSER

Z jakich elementów składa się URL?

81

Do manipulowania URL w JavaScript służy interfejs **Location**, który jest dostępny jako **global object**. Obiekt **location** posiada, m.in.

`href`	- pełny adres URL
`protocol`	- HTTP (bez SSL) lub HTTPS (z SSL)
`hostname`	- Nazwa hosta, np. fiszkijs.pl
`port`	- Numer portu, np. 8081
`pathname`	- Ścieżka do zasobu, np. /en-US/search
`hash`	- Anchor (kotwica), np. #js
`search`	- Dodatkowe parametry zapytania wraz z "*?*", np. ?q=react
`host`	- Nazwa hosta razem z portem, np. fiszkijs.pl:8081

BROWSER

Czym są data attributes?

82

Data attributes w postatci `data-*` tworzą dodatkowy zestaw atrybutów i pozwalają na przesyłanie dodatkowych informacji pomidzy stroną HTML a jej reprezentacją DOM.

Wszystkie **data attributes** są dostępne poprzez interfejs `HTMLElement` elementu, na którym zostały umieszczone.

Atrybuty można odczytać po stronie JavaScript za pomocą metody `getAttribute()` lub elementu `dataset` (nazwa atrybutu zostanie zmieniona na camelCase), np.

```
<div id="auth" data-user-name="Bo" data-id="123">

const elem = document.getElementById('auth');
elem.getAttribute('data-user-name'); // Bo
elem.getAttribute('id');             // 123
elem.dataset.userName;               // Bo
elem.dataset.id;                     // 123
```

BROWSER

Jaka jest różnica między *load* a *DOMContentLoaded*?

83

Zdarzenie **DOMContentLoaded** jest odpalane w momencie, gdy cała strona HTML jest załadowana bez czekania na pobranie pozostałych skryptów lub arkuszy CSS. Ważne jest, aby pozostałe zasoby były pobierane w sposób **asynchroniczny**. W przeciwnym razie parsowanie zostanie wstrzymane.

Z kolei zdarzenie **load** jest odpalane po pełnym załadowaniu strony ze wszystkimi zależnościami, skryptami i obrazkami.

```
document.addEventListener('DOMContentLoaded', (event) => {
    console.log('DOM fully loaded and parsed');
});

window.addEventListener('load', (event) => {
    console.log('Entire page is fully loaded');
});

window.onload = (event) => {
    console.log('Entire page is fully loaded');
};
```

BROWSER

Czym jest operator void?

84

Operator void postawiony przed danym wyrażeniem spowoduje, że zostanie ono wykonane, jednak zwrócony wynik zostanie zmieniony na **undefined**. Stosujemy je w miejscach, gdzie wyrażenie powinno się wykonać, ale nie oczekujemy wyniku.

Przykładowo, można stworzyć link, po kliknięciu którego zostanie wykonany kod, ale nie nastąpi przekierowanie na nową stronę:

```
<a href="javaScript:void(0);"
   onclick="alert('Dobra robota!')">
  Kliknij tutaj!
</a>
<a href="javascript:void(document.body.style.
backgroundColor='red');">
  Kliknij tutaj i zmień tło!
</a>
```

BROWSER

Jaka jest różnica między *window* a *document*?

85

Window to interfejs reprezentujący **okno** (lub tab) wraz z załadowanym dokumentem DOM.

Zawiera wiele funkcji i obiektów, które są dostępne globalnie. Można się do niego odwołać poprzez obiekt `window` lub `document.defaultView`.

Z kolei **Document** to interfejs reprezentujący stronę załadowaną w oknie przeglądarki i stanowi punkt dostępu do drzewa **DOM**. Jest dostępny globalnie przez `document` lub `window.document`.

```
document.defaultView === window
window.document === document
```

COOKIES

Czym są *cookies*?

86

Cookies to dane, które serwer wysyła do przeglądarki internetowej użytkownika, która może je przechowywać i wysyłać ponownie do **tego samego serwera** wraz z kolejnym żądaniem.

Są używane do określenia czy dwa żądania pochodzą od tego samego użytkownika i są sposobem na **zapamiętanie informacji o stanie sesji** pomimo bezstanowej natury HTTP. Istnieje opcja ustawienia daty wygaśnięcia lub czasu trwania, po których ciasteczko nie będzie wysyłane.

Cookies są wykorzystywane do:

- zarządzania sesją,
- obsługi koszyków sklepów internetowych,
- przechowywania preferencji użytkownika oraz
- śledzenia i analizy jego zachowań

COOKIES

Jakie znasz opcje *cookie*?

87

Najważniejsze opcje stosowane przy obsłudze **cookie**:

- Domyślnie cookie jest usuwane w momencie zamknięcia przeglądarki, ale można utworzyć *trwałe ciasteczka*, które wygasają w konkretnym terminie `Expires` lub po określonym czasie `Max-Age`
- `Domain` określa dozwoloną domenę oraz pozwala na dołączenie subdomen, np. `Domain=fiszkijs.pl` obejmuje również `https://blog.fiszkijs.pl`
- `Path` określa dozwoloną ścieżkę URL, np. `/product`
- `HttpOnly` sprawia, że ciasteczka są niedostępne z poziomu JavaScript `document.cookie`
- Zastosowanie `Secure` powoduje, że ciasteczko może być wysłane do serwera tylko przez HTTPS

COOKIES

Jak tworzone są *cookies*?

88

Cookies są tworzone przez przeglądarkę po odebraniu odpowiedzi od serwera, która zawiera nagłówek **Set-Cookie**.

```
HTTP/2.0 200 OK
Content-type: text/html
Set-Cookie: selected_course=react

[content]
```

Nastepnie do każdego kolejnego żądania wysłanego do tej strony internetowej, przeglądarka dołączy nagłówek **Cookie** z zawartością w postaci *klucz-wartość*.

```
GET /demo.html HTTP/2.0
Host: www.fiszkijs.pl
Cookie: selected_course=react
```

Istnieje sposób na tworzenie cookies z poziomu JavaScript poprzez użycie **Document.cookie**, jednak nie jest to sposób bezpieczny. Lepiej przechowywać dane w *localStorage*, *sessionStorage* lub *IndexedDB*.

```
document.cookie = "selected_course=React";
```

COOKIES

W jaki sposób odczytać *cookies*?

89

Aby odczytać listę **cookie** powiązanych z bieżącą stroną, można skorzystać z `document.cookie`, które zwraca listę ciasteczek rozdzieloną średnikami, przykładowo:

```
// selected_course=react; username=Bob
console.log(document.cookie)

// [selected_course=react, username=Bob]
console.log(document.cookie.split('; '))
```

W ten sposób można uzyskać dostęp **tylko** do ciasteczek, które nie mają ustawionej flagi **HttpOnly**.

Aby ograniczać możliwości przeprowadzenia ataku **cross-site scripting** (XSS), ciasteczka **HttpOnly** są niedostępnie z poziomu JavaScript. Można je tylko wysyłać do serwera.

COOKIES

Jak usunąć *cookie*?

90

Nie istnieje funkcja pozwalająca na usunięcie **cookie** bezpośrednio, jednak można to zrobić w sposób *pośredni* poprzez ustawienie parametru **expires** na przeszłą datę, przykładowo:

```
document.cookie = "selected_course=; "
      + "expires=Wed, 01 Jan 2020 00:00:00 UTC; "
      + "path=/;";
```

Nie trzeba przy tym znać aktualnej wartości ciasteczka, wystarczy podanie klucza, pod jakim zostało zapisane.

Ważne jest ustawienie parametru **path**, ponieważ większość przeglądarek automatycznie ten parametr do ciasteczka, nawet jeśli nie został podany w momencie tworzenia. Bez niego nie będzie możliwe usunięcie **cookie**.

HTTP

Jakie są metody żądań HTTP?

91

Specyfikacja **HTTP** określa metody żądań wskazujące na akcję, która ma być wykonana na zasobie:

- `GET` - pobranie zasobu; parametry są przekazywane w adresie żądania
- `POST` - przesłanie danych do serwera; parametry są zapisane w ciele zapytania
- `PUT` - zmiana zasobu, lub jego aktualizacja; jeżeli zasób nie istnieje po stronie serwera, zostanie utworzony
- `DELETE` - usunięcie zasobu na serwerze
- `HEAD` - podobny do żądania GET, ale nie zwraca zawartości zasobu; jest używane do pobrania metadanych o zasobie w postaci nagłówków HTTP, np. sprawdzenia czy zasób się ostatnio zmienił
- `OPTIONS` - służy do pobrania informacji na temat tego, jakie żądania są obsługiwane przez serwer

HTTP

Czym jest CORS?

92

CORS, czyli **Cross-Origin Resource Sharing** jest mechanizmem pozwalającym aplikacjom działającym na jedym adresie **origin** na dostęp do zasobów zdefiniowanych na innym adresie. Przykładowo `https://fiszkijs.pl` wysyła request do `https://www.googleapis.com/gmail/v1/`

Ze względów **bezpieczeństwa** przeglądarki domyślnie blokują możliwość wykonywania takich zapytań.

Aby je umożliwić, serwer, do którego trafi takie zapytanie, musi w odpowiedzi umieścić nagłówek `Access-Control-Allow-Origin: *` lub lepiej `Access-Control-Allow-Origin: https://fiszkijs.pl`

HTTP

Na czym polega polityka same-origin?

93

Same-origin to mechanizm bezpieczeństwa, który zapobiega wysyłaniu żądań HTTP z poziomu JavaScript do serwera działającego na innym **origin** definiowanym jako połączenie protokół + host + port.

Przykładowo do danych zapisanych przez

`https://store.company.com/page.html`

można uzyskać dostęp z (inna ścieżka)

`https://store.company.com/checkout/page.html`

natomiast nie można z (inny protokół)

`http://store.company.com/page.html`

Aby umożliwić dostęp do danych pomiędzy różnymi **origin** należy skorzystać z mechanizmu CORS, czyli **Cross-Origin Resource Sharing**.

HTTP

Na czym polega idempotentność?

94

Idempotentność metod HTTP polega na tym, że identyczne żądania wysyłane wielokrotnie mają ten sam efekt.

Jeżeli zostaną zaimplementowane poprawnie po stronie serwera, idempotentne są metody:

- `GET` - za każdym razem zwraca ten sam zasób
- `HEAD` - za każdym razem zwraca te same metadane
- `PUT` - jeśli encja nie istnieje, utworzy ją, a każdy kolejny request będzie ją aktualizował
- `DELETE` - usunie encję przy pierwszym wywołaniu, kolejne zwrócą kod 404

Natomiast metoda `POST` nie jest idempotentna, ponieważ każdy request powoduje utworzenie nowej encji po stronie serwera.

HTTP

Jak wykonać zapytanie HTTP?

95

W JavaScript zapytania HTTP najlepiej wykonywać za pomocą Fetch API, które ma interfejs podobny używanego wcześniej **XMLHttpRequest**, ale korzysta z Promise, dzięki czemu ma prostsze API bez potrzeby używania callbacków.

Najważniejszą metodą jest `fetch()`, która przyjmuje jeden obowiązkowy parametr - ścieżkę do zasobu. Zwraca **Promise**, który opakowuje **Response**. Drugi argument nie jest obowiązkowy i służy do konfigurowania żądania, przykładowo:

```
fetch('./api/v1/products', {
  method: 'post',
  headers: {
    "Content-type": "application/x-www-form-urlencoded"
  },
  body: 'name=fiszki&category=js'
}).then(response => console.log(response.json()))
  .catch(err => console.log(err));
```

ASYNC

W jakich stanach może wystpować Promise?

96

Promise może występować w trzech różnych stanach:

- **Pending** (w trakcie) - domyślny stan, w którym jest Promise zanim zostanie wykonana; z tego stanu może przejść zarówno do stanu fulfilled, jak i do rejected
- **Fulfilled** (zakończone) - oznacza zakończenie obsługi
- **Rejected** (odrzucone) - oznacza niepowodzenie. W tej sytacji zostanie rzucony wyjątek, który należy obsłużyć

Obsługa Promise, który znajduje się w stanie **fulfilled**, lub **rejected** jest zakończona, w związku z tym Promise nie może zmieniać dalej stanu na inny.

Wartość Promise, którego obsługa została **zakończona** nie może ulec zmianie.

ASYNC

Na czym polega *Promise chaining*?

97

Promise chaining polega na łączeniu i wykonywaniu sekwencji operacji **asynchronicznych** jedna po drugiej.

Jest alternatywą dla tworzenia skomplikowanego kodu opartego na callbackach, który często prowadzi do **callback hell**.

Chaining opiera się na tym, że funkcja `then()` zwraca nowy Promise, który reprezentuje nie tylko wynik wykonania operacji, ale również wynik poprzednich operacji w łańcuchu, przykładowo:

```
fetch('./api/user')
  .then(response => response.json())
  .then(user => fetch(`./api/users/${user.id}`))
  .then(response => response.json())
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

ASYNC

Do czego służy Promise.all()?

98

Funkcja **Promise.all()** służy do łączenia wielu powiązanych operacji asynchronicznych w jedną.

Przyjmuje jako parametr **tablicę Promise** i zwraca **tablicę wyników** w momencie, gdy wszystkie Promise się zakończą.

Jeżeli którykolwiek ze składowych Promise zostanie odrzucony - **rejected** - całość również zostanie odrzucona.

```
const fetchUser = fetch('./api/v1/users');
const fetchPrefs = fetch('./api/v1/prefs');

Promise.all([fetchUser, fetchPrefs])
  .then(response => response.map(r => r.json()))
  .then(values => console.log(values))
  .catch(err => console.log(err));

// [{name: 'Bob', id: 2}, {color: 'red'}]
```

ASYNC

Do czego służy Promise.race()?

99

Funkcja **Promise.race()** przyjmuje jako parametr **tablicę Promise** i zwraca **pojedynczy wynik** od tego Promise, który został wykonany najszybciej.

Przykładowo, chcąc przedstawić użytkownikowi wyniki notowań giełdowych w jak najkrótszym czasie, odpytujemy kilku brokerów, ale interesuje nas tylko pierwszy, najszybciej otrzymany wynik:

```
const getStock1 = getStock('broker1');
const getStock2 = getStock('broker2');
const getStock3 = getStock('broker3');

Promise.race([getStock1, getStock2, getStock3])
    .then(response => response.json())
    .then(stockPrice => console.log(stockPrice))
    .catch(err => console.log(err));
```

ASYNC

Jak można przerwać działanie Promise()?

100

Niestety **nie ma możliwości** przerwania Promise, ponieważ jest on tylko sposobem na wykonywanie i łączenie wyników operacji asynchronicznych. Nie ma *wglądu* do tego jak te operacje działają, więc nie może ich przerwać.

Jednym ze sposobów na *obejście* tego ograniczenia, jest zastosowanie `Promise.race([])`, gdzie oryginalny Promise, np. `getData()` łączymy w tablicy z innym, np. `signal`, który możemy zakończyć szybciej. Jeżeli `signal` zostanie wykonany wcześniej, wynik `getData()` będzie zignorowany.

Jednak nawet takie rozwiązanie nie pozwala na rzeczywiste przerwanie operacji asynchronicznej.

Innym sposobem jest użycie biblioteki do programowania reaktywnego **rxjs**, która korzysta z **Obervables**, które można przerwać na żądanie.

```
(function(){
    var x = y = 5;
})();

console.log(typeof x);
console.log(typeof y);
```

101

Ponieważ zarówno `x`, jak i `y` są zdefiniowane wewnątrz bloku **IIFE** (Immediately Invoked Function Expression), spodziewanym wynikiem jest `"undefined"` w obu przypadkach.

Okazuje się jednak, że `var x = y = 5` nie oznacza

```
var x = 5;
var y = 5;
```

ale

```
y = 5;
var x = y;
```

W efekcie zmienna `y` zostaje utworzona w **global scope**, a `x` w **block scope**. Aby zapobiec tego typu problemom, należy stosować **strict mode**, który w tym przypadku spowoduje zgłoszenie **runtime error**.

```
const course = {
    title: "react",
    printDetails: function () {
        const self = this;
        console.log("a = " + this.title);
        console.log("b = " + self.title);
        (function () {
            console.log("c = " + this.title);
            console.log("d = " + self.title);
        }());
    }
};
course.printDetails();
```

102

Przedstawiony kod wyświetli następujące wartości:

```
a = react
b = react
c = undefined
d = react
```

Wewnątrz `printDetails` zarówno `this`, jak i `self` wskazują na ten sam obiekt `course`, więc mogą odwołać się do właściwości `title`.

W wewnętrznym bloku **IIFE** `this` nie odnosi się już do obiektu `course`, ale do obiektu `window` z **global scope**.

W efekcie wywołanie `this.title` zwraca undefined, podczas gdy `self.title` nadal przechowuje poprawną referencję do obiektu `course`.


console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
103

Przedstawiony kod wyświetli następujące wartości:

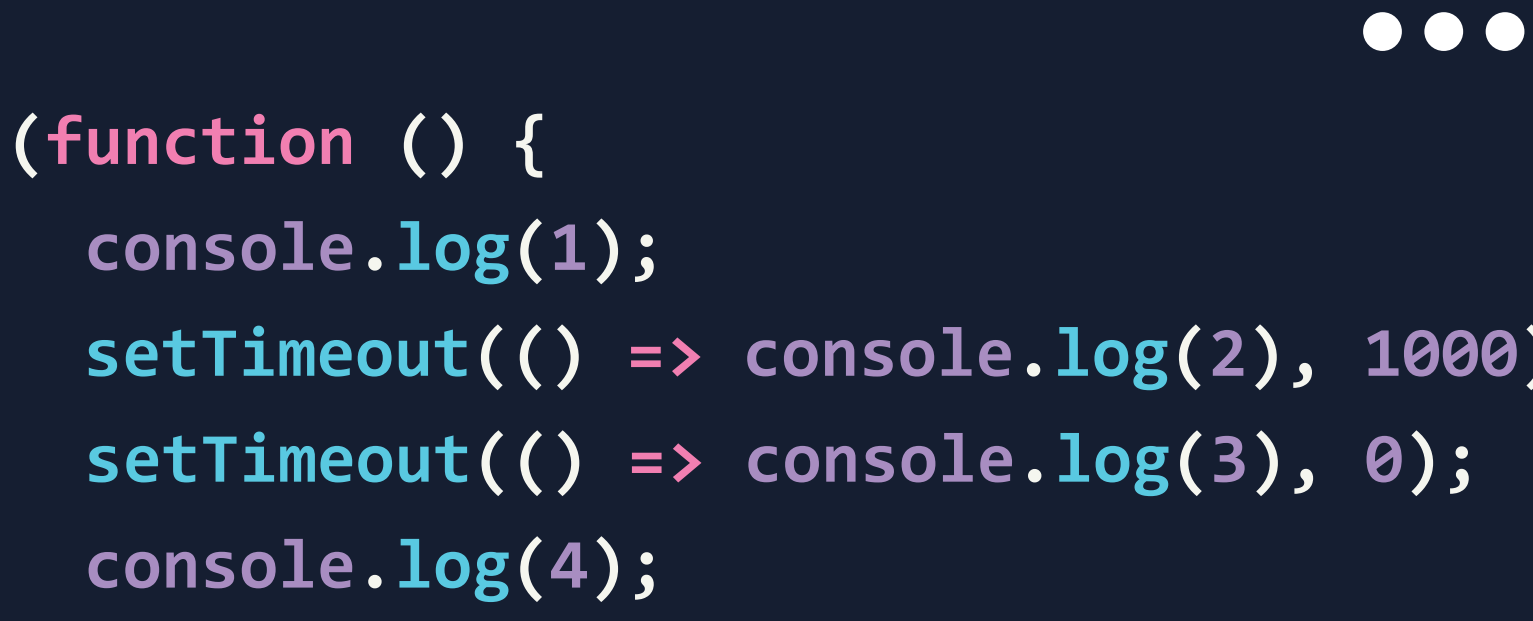
```
0.30000000000000004
false
```

Dzieje się tak, ponieważ liczby w JavaScript są przechowywane jako liczby **zmiennoprzecinkowe**. Wynik niektórych prostych operacji może nie dawać oczekiwanych wyników.

Aby wygodnie wykonywać operacje arytmetyczne na liczbach zmiennoprzecinkowych w JavaScript można skorzystać ze specjalnej stałej `Number.EPSILON`, na przykład:

```
function areEqual(num1, num2) {
    return Math.abs(num1 - num2) < Number.EPSILON;
}

console.log(areEqual(0.1 + 0.2, 0.3)); // true
```


(function () {
console.log(1);
setTimeout(() => console.log(2), 1000);
setTimeout(() => console.log(3), 0);
console.log(4);
})();

104

Wartości zostaną wyświetlone w następującej kolejności:

```
1
4
3
2
```

Wartości `1` oraz `4` są wyświetlone od razu za pomocą prostego wywołania `console.log`.

Wartość `3` jest wyświetlona po opóźnieniu `0ms`, co jednak nie oznacza, że jest wyświetlona natychmiast. Zgodnie z działaniem **Event Loop**, wywołanie `setTimeout` umieszcza funkcję z parametru w **Event Queue** - kolejce zdarzeń - co sprawia, że zostanie obsłużona najwcześniej w kolejnej rundzie, podczas kolejnego *ticku* timera.

```
for (var i = 0; i < 5; i++) {
  setTimeout(() => console.log(i), i * 1000);
}
```

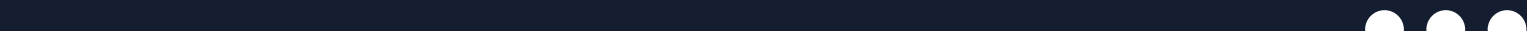
105

Zostaną wyświetlone następujące wartości:

```
5 // po 1 sekundzie
5 // po 2 sekundach
5 // po 3 sekundach
5 // po 4 sekundach
5 // po 5 sekundach
```

Użycie `setTimeout` sprawia, że `console.log` zostanie wykonane dopiero po zakończeniu pętli. Wykorzystanie `var` sprawia, że zmienna `i` jest dostępna w **function scope** również poza blokiem `for`. W efekcie wszystkie wywołania funkcji `console.log` wykorzystają ostatnią wartość `i=5`.

Aby pozbyć się tego problemu, należy zamienić `var` na `let` lub opakować wywołanie `setTimeout` w dodakowe *closure*.



```
for (let i = 0; i < 5; i++) {
  setTimeout(() => console.log(i), i * 1000);
}
```

106

Zostaną wyświetlone następujące wartości:

```
1 // po 1 sekundzie
2 // po 2 sekundach
3 // po 3 sekundach
4 // po 4 sekundach
5 // po 5 sekundach
```

Wykorzystanie `let` sprawia, że zmienna `i` jest dostępna tylko wewnątrz **block scope** i nie jest widoczna poza blokiem `for`.

```
const auth = {
    accessToken: 'secret-token',
    getSecretToken: function (){
        return this.accessToken;
    }
};

const stealToken = auth.getSecretToken;
console.log(stealToken());
console.log(auth.getSecretToken());
```

107

Zostaną wyświetlone następujące wartości:

```
undefined
secret-token
```

Przy pierwszym wywołaniu `console.log` zostanie wyświetlona wartość **undefined**, ponieważ `this.accessToken` nie odnosi się już do obiektu `auth`, ale do obiektu **window**.

Podczas drugiego wywołania `this` wskazuje na obiektu `auth` i ma dostęp do jego właściwości.

Aby naprawić ten problem można skorzystać z funkcji `bind`, która umożliwia zmianę kontekstu wywołania funkcji z powrotem na obiekt `auth`:

```
const stealToken = auth.getSecretToken.bind(auth);
```

```
var price = 99;
var getCoursePrice = function () {
    console.log(price);
    var price = 100;
};
getCoursePrice();
```

108

W tym przykładzie wyświetlona zostanie wartość `undefined`.

Jest to spowodowane tym, że w JavaScript **wszystkie** deklaracje (`function`, `var`, `let`, `cost` i `class`) są **hoistowane**, ale tylko `var` są inicjalizowane przez `undefined`.

Deklaracja `var price = 100;` jest więc przenoszona na górę **function scope** jako `var price;`. W momencie wywołania silnik JavaScript odwołuje się do tej lokalnej zmiennej, która przesłania zmienną istniejącą poza zakresem funkcji.

Deklaracja `var price = 99;` nie jest w ogóle wykorzystywana podczas działania skryptu.

```
var price = 10;
function getCoursePrice() {
    var price = 100;
    function calculatePrice() {
        price++;
        var price = 1000;
        console.log(price)
    }
    calculatePrice();
}
getCoursePrice();
```

109

Zostanie wyświetlona wartość **1000**.

W przykładzie istnieją trzy osobne zmienne o nazwie `price`, w trzech różnych **scope**.

Dekalracje tych zmiennych będą **hoistowane** w górę swojego **function scope**, w efekcie czego funkcja `calculatePrice` będzie odczytywana w następujący sposób:

```
function calculatePrice () {
    var price;              // "undefined"
    price++;                // NaN
    price = 1000;           // 1000
    console.log(price);
}
```

```
var price = 99;
function calculatePrice() {
    price = 100;
    return;
    function price() { }
}
calculatePrice();
console.log(price);
```

110

Zostanie wyświetlona wartość `99`.

W pierwszej kolejności deklaracja funkcji wewnętrznej `function price() {}` jest **hoistowana** w górę zakresu i zachowuje się jak deklaracja `var price = function() {}`. Powstaje więc lokalna zmienna `price`.

Ponieważ zmienne lokalne przykrywają globalne, wywołanie `price = 100` powoduje zmianę zmiennej lokalnej, a nie globalnej.

Wartość zmiennej globalnej pozostaje nie zmieniona.

Gdyby nie istniała deklaracja funkcji `price`, lub nazywałaby się inaczej, wyświetlona zostałaby wartość 100.

```
function calculatePrice(){
    function getPrice() {
        return 99;
    }
    return getPrice();
    function getPrice() {
        return 100;
    }
}
console.log(calculatePrice());
```

111

Zostanie wyświetlona wartość `100`.

W pierwszej kolejności deklaracje obu funkcji wewnętrznych `getPrice` są **hoistowane** w górę zakresu. Powstaje lokalna zmienna `getPrice`, która zostaje natychmiast nadpisana przez `var getPrice = function() { return 100; }`

W scope może istnieć tylko jedna zmienna o tej nazwie. Podczas wywołania `getPrice()`, wykonywany jest kod drugiej funkcji `getPrice`.

```
const reactCourse = {
    price: 59.99,
    getPrice : function() {
        return this.price;
    }
};
const vueCourse = Object.create(reactCourse);
vueCourse.price = 69.99;

delete vueCourse.price;
console.log(vueCourse.getPrice());
```

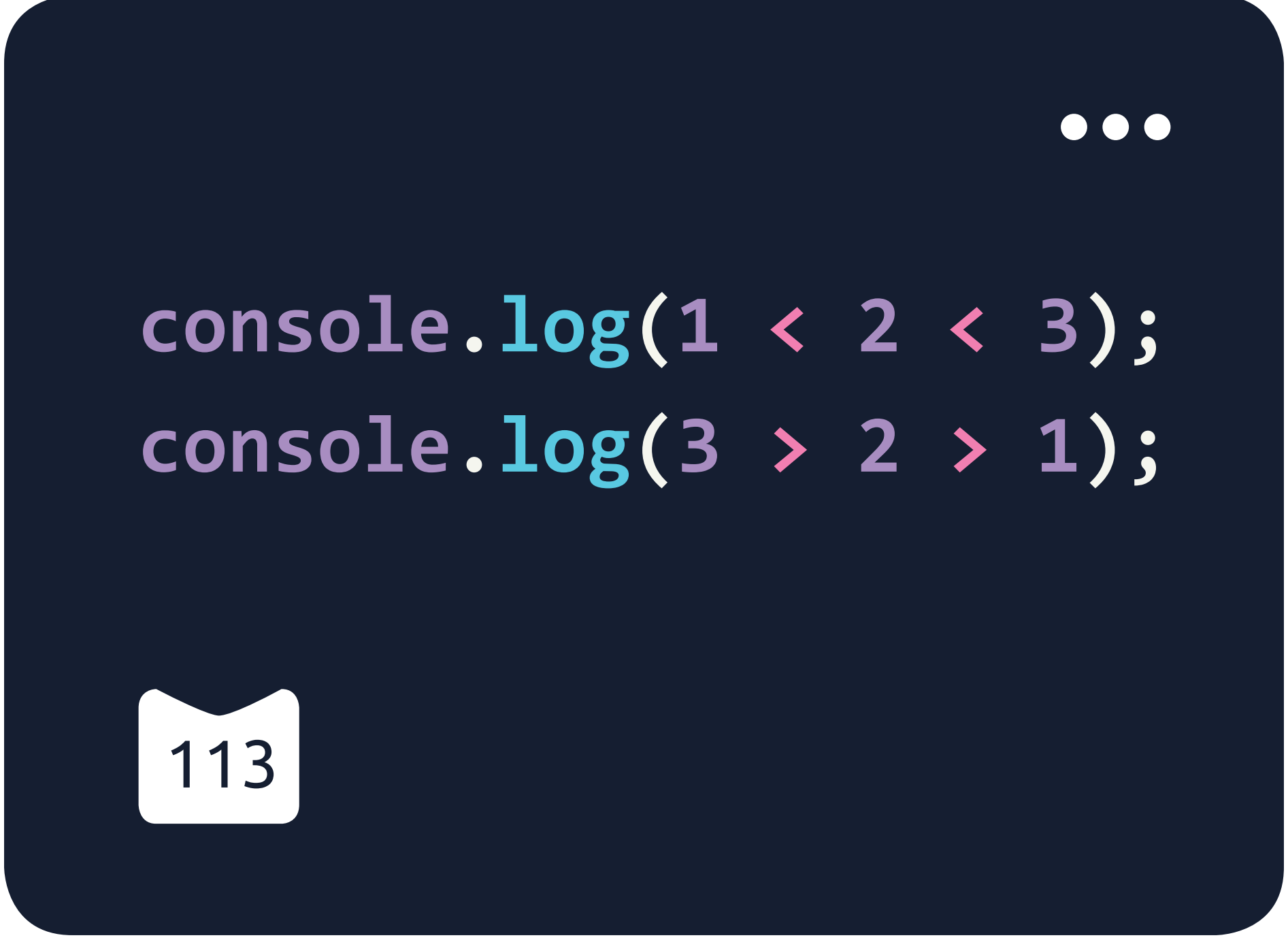
112

Zostanie wyświetlona wartość `59.99`.

Wywołanie funkcji `Object.create` tworzy nowy obiekt `vueCourse`, który będzie miał `reactCourse` jako rodzica.

Wartość `price` jest zapisana jedynie w obiekcie rodzica `reactCourse`. Dopiero próba przypisania nowej wartości do `price`, powoduje powstanie tej *property* na poziomie obiektu dziecka - `vueCourse`. Oznacza to, że próba usunięcia *property* `price` powoduje usunięcie jej tylko z `vueCourse`.

Wywołanie `getPrice` sprawdza, czy **price** jest dostępne w obiekcie `vueCourse`. Jeżeli nie jest, silnik JavaScript przechodzi do obiektu rodzica i dalej, aż dotrze do końca **łańcucha prototypów**.


console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
113

Zostaną wyświetlone następujące wartości:

```
true
false
```

Jest to związane z tym w jaki sposób JavaScript realizuje *łączność* operatorów `<` oraz `>`.

Takie operacje są realizowane **od lewej do prawej strony**. W pierwszej kolejności wykonane zostanie sprawdzenie `1 < 2`, które da wynik `true`. W drugim kroku nastąpi porównanie `true < 3`, które również da wynik `true`, ponieważ lewa strona jest **rzutowana** na wartość 1.

Analogicznie w drugim przypadku `1 > 2` daje `false`, a następnie `false > 3` również daje wynik `false`, ponieważ lewa strona jest **rzutowana** na 0.

```
const courses = ['React']
courses[5] = 'Angular';
courses[4] = undefined

console.log(courses);
console.log(courses[3]);
console.log(courses.map(c => 'Vue'));
```

114

Mimo że tablica posiada początkowo tylko jeden element `"React"`, można do niej dodać nowy element na dowolnej pozycji. Miejsca pomiędzy tymi elementami pozostaną *puste*. Dlatego tablica `courses` wygląda następująco:

```
["React", empty × 3, undefined, "Angular"]
```

Co ważne, puste miejsca są faktycznie *puste* a nie wypełnione wartością `undefined`.

Wywołanie `courses[3]` daje w wyniku `undefined`, jednak w rzeczywistości nie ma tam takiej wartości. Przetworzenie tablicy za pomocą `map(c => 'Vue')` daje

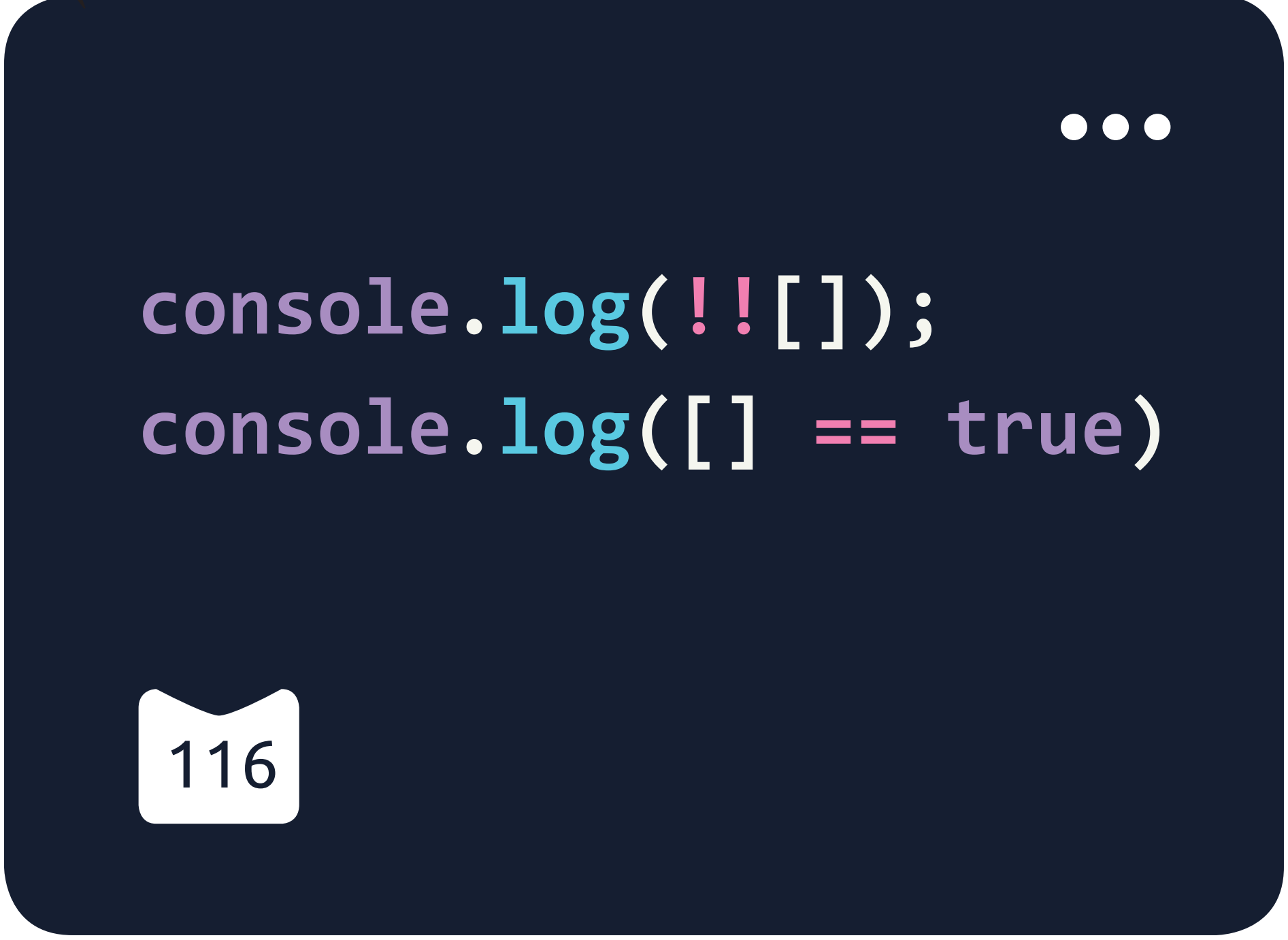
```
["Vue", empty × 3, "Vue", "Vue"]
```

co pokazuje, że *puste miejsca* w tablicy zostały zupełnie pominięte podczas iteracji.


console.log(typeof typeof 1);
115

Wywołanie `typeof 1` daje w wyniku wartość `"number"`, która jest zwykłym stringiem.

Ponowne wywołanie `typeof "number"` zwraca (tak samo jak w przypadku każdego innego napisu) wartość `"string"`.


console.log(!![]);
console.log([] == true)
116

Zostaną wyświetlone wartości

```
true
false
```

Pusta tablica `[]` jest *truthy*. Nie oznacza to jednak, że taka wartość jest równa `true`. Operator `==` przed sprawdzeniem równości wykona rzutowanie typów, jeśli będzie konieczne.

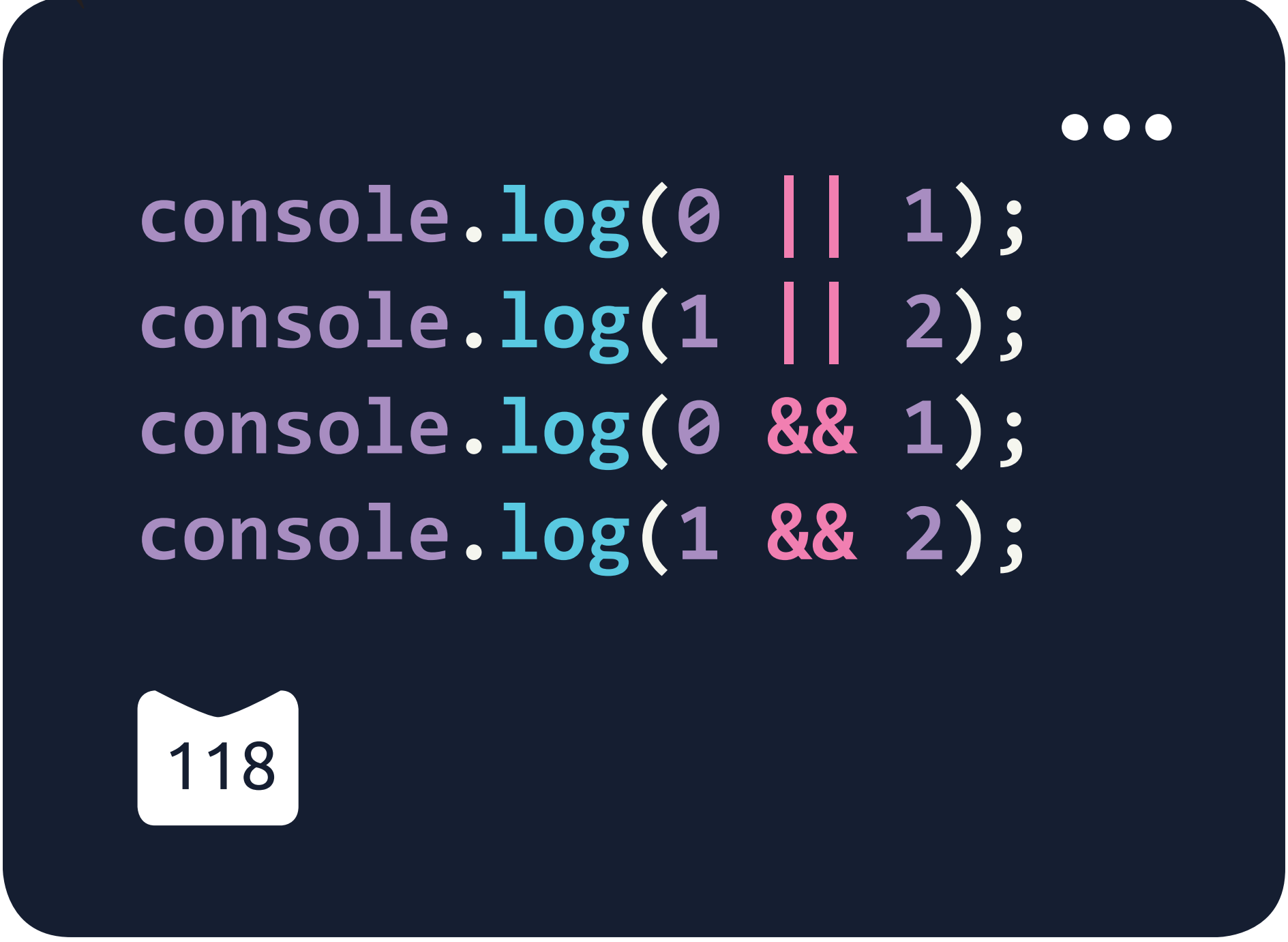
Ponieważ po obu stronach operatora `==` są wartości różnych typów, JavaScript nie potrafi ich bezpośrednio porównać.

W pierwszej kolejności prawa strona zostanie zrzutowana do liczby i będzie wynosić 1. Lewa strona zostanie przekonwertowana do `""` za pomocą `Symbol.toPrimitive`.

W efekcie nastąpi porównanie `"" == 1`, co po kolejnym rzutowaniu daje `0 == 1`.


console.log(1 'false' == false);
console.log(2 2 + true);
console.log(3 '6' + 9);
console.log(4 '6' - 9);
console.log(5 1 + 2 + "3");
console.log(6 2 in [1, 2]);
117

1. `false` - **string**, który nie jest pusty jest *truthy*.
2. `3` - operator `+` pomiędzy liczbą a boolean przekonwertuje boolean do liczby, w tym przypadku do 1
3. `69` - jeśli po którejkolwiek stronie operatora `+` jest **string**, druga strona zostanie również przekonwertowana na **string** a następnie nastąpi ich połączenie
4. `-3` - w przypadku operatora `-` obie strony zostaną przekonwertowane do liczby
5. `33` - po pierwej operacji zostanie `3 + "3"`. Liczba będzie zmieniona na **string** i nastąpi połączenie.
6. `false` - operator `in` sprawdza, czy **property** lub **indeks** znajduje się w obiekcie. W tym przypadku, tablica ma indeksy `0` oraz `1`, ale nie ma `2`


console.log(0 || 1);
console.log(1 || 2);
console.log(0 && 1);
console.log(1 && 2);
118

Zostaną wyświetlone wartości w kolejności `1, 1, 0, 2`

Przy korzystaniu z operatora logicznego `||` (OR) początkowo lewa strona jest konwertowana na **boolean**. Jeżeli jej wartość jest `true`, prawa strona operatora nie jest wyznaczana, a wynikiem jest lewa strona wyrażenia. Jeżeli wartość jest `false`, obliczona zostanie prawa strona operatora.

Operator logiczny `&&` (AND) sprawia, że początkowo lewa strona jest kowertowana na **boolean**. Jeżeli jest `false`, prawa strona operatora nie jest wyznaczana. Jeżeli wartość jest `true`, obliczona zostanie prawa strona operatora.

Operatory **zwracają wartość obliczonego wyrażenia**, więc jeśli po prawej lub lewej stronie były wartości nie będące boolean (np. liczby), takie wartości zostaną zwrócone.

```
const delay = () => new Promise(
    resolve => setTimeout(resolve, 2000)
)
async function displayScore(score) {
    await delay();
    console.log(score);
}
async function processScores(scores) {
    scores.forEach(score => {
        await displayScore(score);
    })
}
processScores([1, 2, 3, 4]);
```

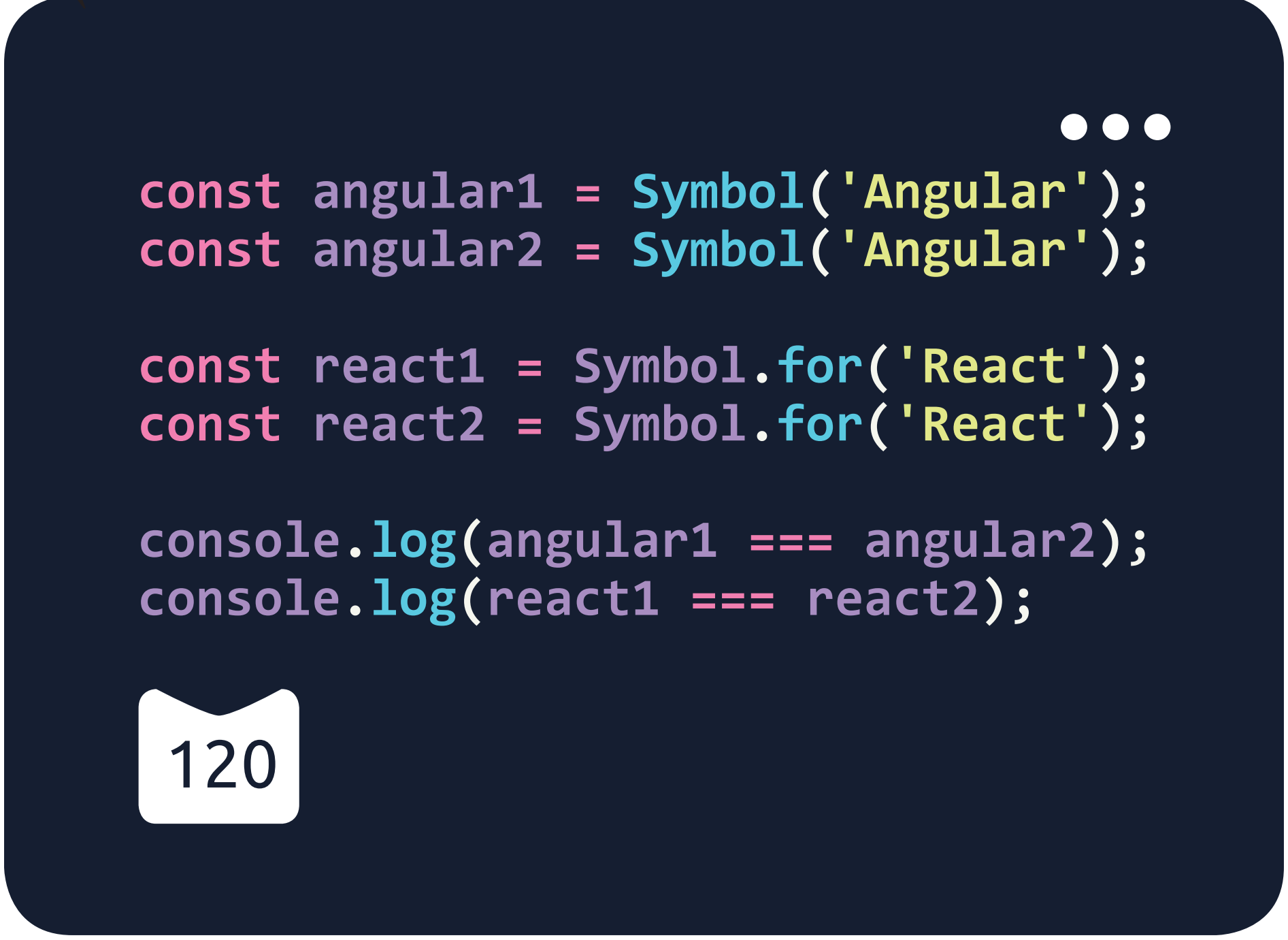
W tym przypadku wyświetlony zostanie błąd

```
SyntaxError: await is only valid in async function
```

Jest to związane ze sposobem w jaki obsługiwane są funkcje **async/await**.

Mimo że funkcja `processScores` jest funkcją **asynchroniczną**, to funkcja anonimowa znajdująca się wewnątrz wywołania `forEach` jest **synchroniczna**.

Jeżeli użyjemy konstrukcji async/await wewnątrz funkcji synchronicznej, zostanie rzucony wyjątek.


const angular1 = Symbol('Angular');
const angular2 = Symbol('Angular');
const react1 = Symbol.for('React');
const react2 = Symbol.for('React');
console.log(angular1 === angular2);
console.log(react1 === react2);
120

Zostaną wyświetlone wartość:

```
false
true
```

Każdy symbol utworzony za pomocą `Symbol()` jest **unikalną wartością** niezależnie od przekazanego parametru. W szczególności dwa symbole utworzone z tą samą nazwą wskazują na dwie różne wartości.

Z drugiej strony symbole utworzone przez `Symbol.for()` są tworzone w **globalnym rejestrze symboli**.

Nie oznacza to, że przy każdym wywołaniu utworzony zostanie nowy symbol. Stanie się tak tylko, jeśli symbol o danym kluczu jeszcze nie istnieje. W przeciwnym razie **zostanie zwrócony istniejący symbol**.

PERFORMANCE

Jaka jest różnica między <script async> a <script defer>?

121

```
// załaduj skrypt bez przerywania renderowania
<script src="courses.js" async></script>
```

Atrybut `async` pozwala na **asynchroniczne pobieranie** skryptów podczas ładowania strony. Dzięki temu skrypty pobierają się jednocześnie **nie przerywając procesu renderowania**. Znacznie przyśpiesza to łączny czas załadowania strony.

```
// załaduj skrypt po zakończeniu ładowania strony
<script src="courses.js" defer></script>
```

Z kolei atrybut `defer` **opóźnia wykonanie** skryptów do czasu pełnego załadowania strony. Skrypt zostanie pobrany wcześniej, ale wykonany dopiero po pełnym parsowaniu strony, tuż przed odpaleniem zdarzenia **DOMContentLoaded**.

PERFORMANCE

Ładowanie czcionek przez <link> czy @import?

122

Z punktu widzenia wydajności strony, lepszym sposobem jest ładowanie czcionek poprzez elementy `<link>`, ponieważ przeglądarki internetowe są w stanie pobierać je całkowicie **równolegle**.

```
<link rel="stylesheet"
      href="https://fonts.googleapis.com/...">
```

Istnieje możliwość ładowania czcionek przez dodanie dyrektywy `@import` w pliku CSS, jednak to **uniemożliwia** równoległe pobieranie zasobów.

Przeglądarka, parsując kod strony, najpierw pobierze wszystkie arkusze CSS a dopiero później odczyta ich zawartość, łącznie z dyrektywami `@import`.

```
@import url('https://fonts.googleapis.com/...');
```

PERFORMANCE

Czym jest *garbage collector*?

123

Garbage Collector to mechanizm czyszczenia pamięci wykorzystywany w różnych językach programowania.

W Javascript stosuje się **automatyczne zarządzanie pamięcią**. Oznacza to, że gdy tworzymy nowe obiekty, jest im automatycznie przydzielana pamięć, a gdy nie są już używane, są z niej automatycznie usuwane.

Do sprawdzenia, czy obiekt może zostać usunięty z pamięci można wykorzystać różne **algorytmy**.

Opierają one swoje działanie na badaniu ***dostępności obiektów***. Oznacza to, że jeżeli do danego obiektu nie można się w żaden sposób odwołać (nie ma do niego żadnej referencji), staje się on nieużywalny, a więc i może być usunięty z pamięci.

PERFORMANCE

Jakie znasz algorytmy *garbage collectora*?

124

Najprostszym algorytmem jest algorytm liczenia odwołań do obiektu - **Reference Counting Algorithm**, w którym obiekt może zostać usunięty z pamięci, jeśli nie ma do niego żadnych odwołań.

Był stosowany w starszych przeglądarkach, jednak posiada wadę polegającą na tym, że obiekty, które mają referencje do siebie nawzajem, **nie zostaną nigdy usunięte**, co prowadzi do **wycieków pamięci**.

Obecnie w JavaScript wykorzystywany jest algorytm **Mark and Sweep**. Sprawdza on czy do każdego obiektu można uzyskać dostęp zaczynąc od **głównego obiektu**, którym w przypadku przeglądarki jest obiekt **window**.

Dzięki temu poprawnie wykrywa obiekty które mają cykliczne zależności między sobą.

PERFORMANCE

Na czym polega Reference Counting Algorithm?

125

Algorytm liczenia odwołań do obiektu - **Reference Counting Algorithm** usuwa obiekt z pamięci, jeśli nie ma do niego żadnych odwołań. Na przykład po wykonaniu poniższego kodu, oryginalny obiekt `{ title: 'React' }` nie jest dostępny przez żadną referencję więc może zostać usunięty:

```
let reactCourse = { title: 'React' };
let angularCourse = { title: 'Anguar' };
reactCourse = {};
```

Jednak obiekty, które mają referencje do siebie nawzajem, **nie zostaną nigdy usunięte**.

```
function foo() {
    let reactCourse = { title: 'React' };
    let angularCourse = { title: 'Anguar' };
    reactCourse.prerequiste = angularCourse;
    angularCourse.prerequiste = reactCourse;
}
```

PERFORMANCE

Na czym polega algorytm Mark and Sweep?

126

Algorytm **Mark and Sweep** sprawdza czy do każdego obiektu można uzyskać dostęp zaczynąc od głównego obiektu, którym w przeglądarce jest obiekt **window**.

Podczas działania, algorytm **oznacza** wszystkie obiekty dostępne z **window**, następnie wszystkie obiekty dostępne z tych już oznaczonych itd.

W ten sposób oznaczy **(mark)** wszystkie *używane* i *dostępne* obiekty. Te, które nie zostały oznaczone, są przeznaczone do usunięcia **(sweep)**.

Dzięki temu poprawnie wykrywa obiekty które nie mają referencji z zewnątrz oraz takie, które mają cykliczne zależności między sobą.

PERFORMANCE

Co to jest *memory leak*?

127

Memory leak to pamięć, która jest aktualnie zajęta, ale nie jest potrzebna do działania aplikacji. Innymi słowy została zajęta przez obiekty, które nie są już potrzebne, ale nie zostały usunięte z pamięci.

W JavaScript do wycieków dochodzi najczęściej na skutek:

- przypadkowego utworzenia zmiennych globalnych, np. przez pominięcie słowa kluczowego `var`, `let` lub `const`
- niepoprawnego użycia `this` wewnątrz funkcji
- błędnego użycia własnoręcznie stworzonego *cache*, powodującego jego niekontrolowany rozrost
- pozostawienia działającej funkcji `setInterval`
- przechowywania referencji do elementów `DOM` - nawet gdy zostaną już usunięte z drzewa `DOM`

PERFORMANCE

Jak zoptymalizować zużycie zasobów?

128

Aby zoptymalizować zużycie zasobów przez stronę internetową można **zmniejszyć rozmiar pobieranych plików** lub **poprawić jakość transferu**. W tym celu należy:

- Zminifikować pliki `*.js`, `*.css`, `*.html`
- Włączyć kompresję **gzip** dla przesyłanych zasobów
- Zoptymalizować rozmiar, jakość i kompresję obrazów
- Zoptymalizować pliki `*.svg` poprzez usunięcie niepotrzebnych metadanych
- Ograniczyć rozmiar strony poprzez usunięcie białych znaków, komentarzy i nieużywanego kodu
- Ograniczyć rozmiar skryptów poprzez `tree shaking`
- Ładować skrypty **asynchronicznie** tam gdzie jest to możliwe poprzez `<script async>`
- Opóźnić wykonanie skryptów do czasu załadowania strony tam gdzie to możliwe przez `<script defer>`

PERFORMANCE

Jak zwiększyć performance strony?

129

Aby poprawić performance strony internetowej można:

- Wykorzystać **HTTP/2**, które korzysta z multipleksingu i pozwala na obsługę wielu zapytań jednocześnie
- Ograniczyć liczbę przesyłanych plików
- Ograniczyć liczbę requestów do serwisów znajdujących się na innych domenach
- Wykorzystać **cache** do przechowywania często używanych danych oraz referencji do elementów **DOM**
- Wykorzystać **lazy-loading** do ładowania skryptów
- Ładować tylko skrypty, które są wymagane a całą resztę **opóźnić**, np. poprzez użycie `<script defer>`
- Usunąć z kodu **skrypty *inline***, które wydłużają proces odczytu i przetworzenia strony
- Wykorzystać kompresję plików **gzip**
- Ograniczyć **liczbę zależności** do innych bibliotek JS

PERFORMANCE

Jak odnaleźć memory leak w aplikacji?

130

Aby odnaleźć memory leak w kodzie JavaScript można się posłużyć zestawem narzędzi **Chrome Devtools**, na przykład:

- Za pomocą **Chrome Task Manager** (`Shift + Esc`) dodaj do tabeli kolumnę z *JavaScript memory* aby zobaczyć zużycie pamięci
- Przeanalizuj zużycie pamięci w czasie za pomocą **Performance recordings** - otwórz Devtools (`F12`), panel *Performance* i zaznacz *Memory*. Możesz zaobserwować jak rośnie zużycie pamięci *JS Heap*, liczba elementów DOM lub liczba event listenerów. Na wykresie będzie również widać działanie **garbage collectora**.
- Wyszukaj elementy **detached DOM** - czyli takie, które zostały odłączone od głównego drzewa - w panelu *Memory* wykonaj *Heap Snapshot* a następnie wyszukaj elementy o nazwie zaczynającej się od *Detached...*

JAVASCRIPT

Co to jest WeakSet?

131

WeakSet to struktura danych przechowująca **tylko obiekty** posiadające *słabe referencje*. Jeżeli nie istnieje referencja wskazująca na obiekt w kolekcji, to może on zostać usunięty z pamięci przez **garbage collector**. Zapobiega w ten sposób powstawaniu wycieków pamięci.

Ta cecha sprawia, że nadaje się doskonale do przechowywania informacji o dużej liczbie często tworzonych obiektów.

WeakSet oferuje zestaw operacji zbliżony do tego ze zwykłego Set:

```
const weakSet = new WeakSet();
const course = { title: 'React' };
weakSet.add(course);
weakSet.has(course);    // true
weakSet.delete(course); // usuwa kurs ze zbioru
weakSet.has(course);    // false
```

JAVASCRIPT

Co odróżnia WeakSet od Set?

132

Najważniejszą różnicą miedzy Set a WeakSet polega na tym, że Set przechowuje ***zwykłe referencje*** do obiektów a WeakSet ***słabe referencje***.

Oznacza to, że obiekt w WeakSet może zostać usunięty z pamięci przez **garbage collector** jeśli nie ma do niego żadnych innych referencji spoza kolekcji.

Dodatkowo:

- Set mogą przechowywać dowolne wartości, natomiast WeakSet **tylko obiekty**
- WeakSet nie posiada właściwości `size` - nie można sprawdzić liczby obiektów w zbiorze
- WeakSet nie posiada metod `clear`, `keys`, `values`, `entries`, `forEach` - **uniemożliwia iterowanie** po obiektach kolekcji

JAVASCRIPT

Co to jest WeakMap?

WeakMap to struktura danych przechowująca kolekcję danych w postaci par klucz-wartość, gdzie kluczami sa **obiekty** posiadające *słabe referencje* a wartościami dane dowolnego typu.

Jeżeli jakiś klucz w mapie nie posiada referencji wskazujących na siebie, to może on zostać usunięty z pamięci przez **garbage collector**.

Zapobiega w ten sposób powstawaniu wycieków pamięci.

```
const weakMap = new WeakMap();
const course = { title: 'React' };
weakMap.set(course, 12);
weakMap.has(course);    // true
weakMap.delete(course); // usuwa kurs
weakMap.has(course);    // false, kurs usunięty
```

JAVASCRIPT

Co odróżnia WeakMap od Map?

134

Najważniejszą różnicą miedzy Map a WeakMap polega na tym, że w WeakMap przechowywane klucze mają ***słabe referencje***.

Oznacza to, że jeżeli do takiego klucza nie ma zewnętrznych referencji, może on zostać usunięty z kolekcji i z pamięci przez **garbage collector**.

Dodatkowo:

- Map może przechowywać klucze dowolnego typu a WeakMap tylko obiekty
- WeakMap nie posiada właściwości `size` - nie można sprawdzić liczby obiektów w mapie
- WeakMap nie posiada metod `clear`, `keys`, `values`, `entries`, `forEach` - **uniemożliwia iterowanie** po kluczach lub wartościach mapy

JAVASCRIPT

Czym jest *barrel* w ES6?

135

Począwszy od ES6 **barrel** to sposób na połączenie danych eksportowanych z różnych modułów w jeden moduł, z którego można wygodnie korzystać w kodzie aplikacji.

Barrel jest zwykłym **modułem ES6**, który eksportuje dane eksportowane wcześniej przez inne moduły. Przykładowo:

```
// courses/index.js (plik barell)
export * from './js';
export * from './react';
export * from './vue';

// app.js
import {
    JsCourse,
    ReactCourse,
    VueCourse
} from '../courses';
```

JAVASCRIPT

Czym jest *TypedArray*?

136

TypedArray to obiekt przypominający tablicę i przechowujący dane binarne. Co prawda nie istnieje globalna property o nazwie TypedArray, ale istnieją **odmiany TypedArray** przystosowane do obsługi różnych typów danych, przykładowo:

- `Int8Array`, `Int16Array`, `Int32Array`
  tablice liczb ze znakiem
- `Uint8Array`, `Uint16Array`, `Uint32Array`
  tablice liczb bez znaku
- `Float32Array`, `Float64Array`
  tablice liczb zmiennoprzecinkowych

Przykładowo, aby utworzyć tablicę liczb 32-bitowych:

```
const bytes = 1024;
const arr = new Int32Array(bytes);
```

JAVASCRIPT

Jaka jest różnica między obiektami typu *host* i *native*?

137

Obiekty typu **native** to obiekty, których semantyka, działanie i zachowanie jest określone przez **specyfikację ECMAScript**. Do takich obiektów należą przykładowo: `Object`, `Date`, `Math`, `RegExp`, `Function`, itp.

Z drugiej strony obiekty typu **host** to obiekty **dostarczane przez środowisko**, w którym działa JavaScript - **przeglądarka**, lub silnik nodeJS **V9**. Do takich obiektów należą (w środowisku przeglądarki) `window`, `document`, `location`, `history`, `XmlHttpRequest`, itp.

Istnieje również trzecia kategoria obiektów - **user**, czyli obiekty użytkownika. Należa do niej wszystkie pozostałe obiekty tworzone w kodzie.

JAVASCRIPT

Jakie wartości może przyjmować *this*?

138

W zależności od bieżącego **kontekstu wywołania funkcji**, **this** może przyjmować następujące wartości:

- W **global scope** lub wewnątrz funkcji wskazuje na `window`,
- Wewnątrz IIFE, gdy używany jest `"use strict"`, **this** jest `undefined`.
- Podczas wywołania metody w obiekcie wskazuje na ten obiekt.
- Wewnątrz `setTimeout` wskazuje na obiekt **window**
- W konstruktorze wskazuje na nowo tworzony obiekt.
- Podczas wywołania `bind`, `call` lub `apply` wskazuje na obiekt przekazany jako pierwszy parametr
- Podczas obsługi **zdarzeń DOM** wskazuje na element DOM, który był źródłem zdarzenia.

JAVASCRIPT

Do czego służy *Iterator*?

139

Iterator jest to obiekt, który wie jak uzyskać dostęp do elementów kolekcji. Dostarcza metodę `next()`, która zwraca następny obiekt w sekwencji, który posiada dwie właściwości: `done` i `value`. Przykładami obietków które implementują **Iterator** są: Array, String, Map, Set.

Iteratorów można używać w kontekście:

- *spread operator*, np. `[...courses]`
- pętli `for...of`
- tworzenia tablicy za pomocą `Array.from()`
- konstruktorów `new Map([iterable])`, `new Set([iterable])`, `new WeakMap([iterable])`, `new WeakSet([iterable])`
- metod Promise, np. `Promise.all([iterable])`, `Promise.race([iterable])`

JAVASCRIPT

Do czego służy *Generator*?

140

Generator to funkcja, która działa jak fabryka iteratorów. Tworzy się je za pomocą konstrukcji `function*` oraz z wykorzystaniem operatora `yield`.

Generatory obliczają zwracaną wartość **na żądanie**, co jest przydane szczególnie, gdy operacje są **kosztowne** lub potencjalnie **nieskończone**, na przykład:

```
function* idGenerator() {
    let id = 0;
    while(true) {
        yield ++id;
    }
}
let generator = idGenerator();
generator.next() // {value: 1, done: false}
generator.next() // {value: 2, done: false}
generator.next() // {value: 3, done: false}
```

SECURITY

Jakie znasz sposoby na poprawę bezpieczeństwa aplikacji JavaScript?

141

Do poprawy bezpieczeństwa aplikacji JavaScript można wykorzystać następujące techniki:

- Hashowanie haseł za pomocą silnego algorytmu, np. **SHA-256** lub **bcrypt** wraz z dodaną wartością ***salt***
- Dodawanie do formularzy losowej wartości, aby zapobiec atakom **CSRF**
- Używanie nagłówków CORS do definiowania dopuszczalnych zachowań **Cross-Origin** (CORS)
- **Szyfrowanie** zawartości plików cookie oraz innych danych zapisanych po stronie klienta
- Stosowania flagi **HttpOnly** dla ciasteczek
- Walidowanie danych wprowadzanych przez użytkownika pod kątem ataków **Cross Site Scripting**
- Wyłączenie **autouzupełniania** najbardziej wrażliwych danych użytkownika
- Unikanie funkcji **eval**

SECURITY

Jak zabezpieczyć cookie?

142

Najważniejsze poczas obsługi ciasteczek jest zapewnienie, aby dane były przesyłane w sposób **zaszyfrowany**.

W tym celu należy ograniczyć użycie ciasteczek tylko do stron *bezpiecznych,* używających odpowiedniego **szyfrowania**.

Oznacza to, że dane z ciasteczek nie będą przesyłane do strony, która nie używa protokołu **SSL/HTTP**.

Aby oznaczyć ciasteczko jako ***Secure*** należy skorzystać z:

```
document.cookie = "secret=test;secure";
```

Dodatkowo należy ustawić następujące parametry:

- HttpOnly – cookie nie może być odczytane w JS,
- Domain– nazwa domeny,
- Path – dokładna nazwa ścieżki na domenie,
- Expires – określa do kiedy cookie jest ważne
- Max-Age - określa po jakim czasie cookie traci ważność

SECURITY

Na czym polega atak Cross-site Scripting?

143

Atak **Cross-site Scripting** (XSS) polega na **wstrzyknięciu kodu** do prawidłowo działającej strony internetowej lub aplikacji webowej, co umożliwia atakującemu wykonanie złośliwych akcji w zaatakowanym systemie oraz uzyskanie dostępu do `localStorage`, `sessionStorage` oraz `cookies`.

Najpopularniejszą formą tego ataku jest znalezienie pola `<input>`, którego wartość nie jest w żaden sposób walidowana ani czyszczona po stronie aplikacji. Atakujący wpisuje w pole `<input>` fragment kodu JavaScript, który następnie zostanie zapisany w bazie danych i wyświetlony w innych miejscach systemu jako dane użytkownika.

Aby zabezpieczyć stronę przed atakiem XSS należy zawsze **czyścić** i **sprawdzać zawartość formularzy** wprowadzanych przez użytkownika.

SECURITY

Na czym polega atak Cross-site Request Forgery?

144

Atak **Cross-site Request Forgery** (CSRF) jest atakiem, w którym atakujący wykonuje w imieniu ofiary żądanie HTTP **korzystając z jej uprawnień**, np. zapisanych w cookie.

Najpopularniejsza forma tego ataku polega na znalezieniu niechronionego **formularza,** z którego korzysta ofiara. Atakujący **preparuje żądanie HTTP** pod ten sam adres co formularz, ale z wykorzystaniem złośliwych danych. Gdy użytkownik wyśle spreparowane żądanie, przeglądarka **dołączy do niego zawartość plików cookie**, co może spowodować wykonanie w systemie akcji, o której użytkownik nie wiedział.

Aby zabezpieczyć się przed atakiem CSRF należy do każdego formularza dodać losową wartość - **token CSRF**, którego obecnosc i poprawność będzie zweryfikowana po stronie serwera po otrzymaniu żądania.

SECURITY

Na czym polega atak Server-side JavaScript Injection?

145

Atak **Server-side JavaScript Injection** jest atakiem wymierzonym w **część backendową** aplikacji, najczęściej napisaną w Node.js.

Umożliwia on atakującemu **wykonanie złośliwego kodu** na serwerze, co może okazać się bardzo poważne w skutkach, głównie z powodu uzyskania bezpośredniego dostępu do bazy danych.

Aby uniknąć tego ataku należy pozbyć się wywołań funkcji **eval**, która jest bardzo podatna na tego typu ataki i umożliwia wykonanie dowolnego kodu JavaScript.

Dodatkowo należy pamiętać, żeby nigdy nie używać wartości wprowadzonych do systemu przez użytkownika **zanim nie zostaną zwalidowane i oczyszczone**.

POJĘCIA

Jaka jest różnica między *Obfuscation* a *Encryption*?

146

Obfuskacja - *zaciemnianie* kodu - to proces, w trakcie którego zmieniany jest wygląd kodu, w taki sposób, aby **utrudnić jego odczytanie** i analizowanie **przez człowieka.** Z punktu widzenia parsera JavaScript jest to nadal poprawny kod. Do przeprowadzenia procesu nie jest potrzebny klucz.

Obfuskację przeprowadza się, aby:

- zmniejszyć rozmiar kodu i przyśpieszyć jego transfer
- ukryć logikę biznesową zapisaną w kodzie
- utrudnić analizę kodu osobom trzecim

Szyfrowanie - *encryption* - polega na zamianie formatu kodu na format niemożliwy do oczytania **bez posiadania klucza** szyfrującego. Przeprowadza się je, aby zabezpieczyć kod przed oczytaniem przez osoby nie posiadające uprawnień.

POJĘCIA

Czym jest *event table*?

147

Event Table - tabela zdarzeń - jest to struktura danych, która przechowuje informacje na temat **zarejestrowanych zdarzeń** asynchronicznych, które zostaną wykonane po zakończeniu pewnych akcji, na przykład po upłynięciu zdanego czasu, lub odebraniu odpowiedzi z serwera HTTP.

Przykładowo, podczas każdego wywołania operacji asynchronicznej takiej jak `setTimeout` do **Event Table** dodawany jest wpis zawierający informację o funkcji, która ma zostać wykonana po upłynięciu czasu określonego przez parametr `setTimeout`.

Gdy to nastapi, funkcja jest przekazywana do **Event Queue** w celu kontynuowania przetwarzania.

POJĘCIA

Co to jest *Temporal Dead Zone*?

148

Temporal Dead Zone to sytuacja w kodzie, w którym zmienne są wykorzystywane przed ich deklaracją. Próba odczytu lub zapisu takich zmiennych powoduje błąd **ReferenceError**.

Temporal Dead Zone dotyczy tylko zmiennych deklarowanych za pomocą słów kluczowych `let` i `const`. Zmienne deklarowane przez `var` są **hoistowane i inicjalizowane** wartością undefined nawet przed deklaracją

Czas pomiędzy odwołaniem się do zmiennej a jej deklaracją to właśnie **Temporal Dead Zone**, przykładowo:

```
function printCourseDetails() {
    console.log(maxStudents);  // undefined
    console.log(courseLength); // ReferenceError
    var maxStudents = 12;
    let courseLength = 30;
}
```

POJĘCIA

Czym jest *service worker*?

149

Service worker to skrypt JavaScript, który jest wykonywany w tle, tj. oddzielnie od kodu JavaScript wykonywanego na stronie. Dostarcza funkcjonalności, które nie wymagają dostępu do DOM ani interakcji z użytkownikiem.

Wykorzystuje się je do

- obsługi *offline* strony i implementacji **PWA**,
- synchronizacji danych z zewnętrznymi serwisami,
- wysyłania **notyfikacji *push***,
- przechwytywania i obługi żądań HTTP oraz
- zarządzania ***cache***, przydatnych w przypadku częstego wykonywania takich samych żądań HTTP.

Service worker **nie ma bezpośredniego dostępu do DOM**, ale może się komunikować ze stroną poprzez metodę `postMessage` i w ten sposób sterować wyświetlaniem strony.

POJĘCIA

Do czego służą server-sent events?

150

Server-sent events (SSE) to technologia pozwalająca serwerowi na **wysyłanie informacji do przeglądarki** w dowolnym momencie. SSE tworzy **jednokierunkowy kanał komunikacji**, co oznacza, że zdarzenia mogą płynąć tylko od serwera do przeglądarki.

Takie rozwiązanie jest przydatne przy implementowaniu interfejsów użytkownika, które wyświetlają **często zmieniające się dane**, np. notowania giełdowe, wiadomości na Twitterze lub informacje na stonie użytkownika Facebooka.

Implementacja SSE opiera się na interfejsie `EventSource`, który pozwala na otwarcie kanału komunikacji z serwerem HTTP, za pomocą którego serwer będzie wysyłał zdarzenia w formacie `text/event-stream`. Połączenie pozostanie otwarte do momentu wywołania `EventSource.close()`.