

Notatka JavaScript

Autor: Oliwier Markiewicz

Definicja

JavaScript to wszechstronny, dynamiczny język programowania, który działa po stronie klienta (w przeglądarce internetowej) oraz po stronie serwera (dzięki platformie Node.js). Jest używany głównie do tworzenia interaktywnych elementów na stronach internetowych, takich jak animacje, dynamiczne aktualizowanie zawartości czy obsługa formularzy.

Cechy

- Jest językiem interpretowanym (kod jest wykonywany bez potrzeby wcześniejszej komplikacji).
- Posiada składnię zbliżoną do języków C i Java, co ułatwia naukę programistom.
- Obsługuje programowanie obiektowe, funkcyjne i zdarzeniowe.
- Jest podstawową częścią technologii webowych, obok HTML i CSS.

Powstanie

Data

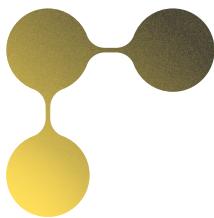
4 grudnia 1995 roku

Twórca

Brendan Eich



Cele



JavaScript stosuję się:

1. **Frontend (strona kliencka)** JavaScript jest używany do tworzenia dynamicznych i interaktywnych elementów na stronach internetowych,
2. **Backend (Node.js)** Dzięki platformie Node.js JavaScript może być używany po stronie serwera,

Jak Uruchomić kod JavaScript



**W przeglądarce
(konsola DevTools)**

**Za pomocą Node
js**

Dołączanie kodu JavaScript

Wstawienie kodu JS bezpośrednio w HTML

W tej metodzie kod JavaScript jest osadzony wewnątrz tagu `<script>` w sekcji `<head>` lub `<body>` dokumentu HTML.

Podłączenie pliku zewnętrznego



Jest to najlepsza praktyka, ponieważ pozwala oddzielić kod JavaScript od kodu HTML, co zwiększa czytelność i ułatwia konserwację.

Aby poprawić wydajność strony (skrypty JS mogą opóźniać renderowanie strony), możesz umieścić `<script>` tuż przed zamykającym tagiem `</body>`.

Möżesz kontrolować sposób wczytywania skryptu, dodając atrybuty `async` lub `defer`.

- **async:** Skrypt ładowany równolegle z innymi elementami strony i wykonywany natychmiast po załadowaniu.
- **defer:** Skrypt jest ładowany równolegle, ale uruchamiany dopiero po zakończeniu parsowania HTML.

W osobnym pliku

```
1 <script src="../app.js"></script>
```

W tym pliku HTML

```
1 <script>
2   console.log("Hello");
3 </script>
```

Async / Defer

```
1 <script async>
2   console.log("Hello");
3 </script>
```

```
1 <script defer>
2   console.log("Hello");
3 </script>
```

Zmienne

Właściwość	Const	Let	Var (niezalecany)
Przeznaczenie	Stałe wartości, których nie można zmienić	Zmienne o zasięgu blokowym	Zmienne o zasięgu funkcyjnym
Możliwość ponownej deklaracji	✗ Nie można	✗ Nie można	✓ Można
Możliwość ponownego przypisania	✗ Nie można	✓ Można	✓ Można
Zasięg	Blokowy ({})	Blokowy ({})	Funkcyjny (ignoruje zasięg blokowy)

```
1 const PI = 3.14;
2 // PI = 3.15; // Błąd: nie można zmienić wartości
```

```
1 let count = 0;  
2 count = 1; // Można zmienić wartość
```

```
1 var name = "Alice";  
2 var name = "Bob"; // Możliwa ponowna deklaracja
```

→ Typy Danych

Podstawowe:

- String
- Number
- Boolean
- null
- undefined

Złożone:

- Object
- Array
- Function

→ Operacje na typach danych

String:

```
1 const name = "Oliwer";
2 console.log(`Witaj, ${name}`);
3 // lub
4 console.log("Witaj, " + name);
```

```
1 // Najważniejsze operacje na stringach w JavaScript
2
3 // 1. Tworzenie stringów
4 const str1 = "Hello, World!"; // Podwójne cudzysłowy
5 const str2 = 'JavaScript jest super'; // Pojedyncze cudzysłowy
6 const str3 = `To jest szablon: ${str1}`; // Template string (z interpolacją)
7
8 console.log(str1, str2, str3);
9
10 // 2. Łączenie stringów
11 const combined = str1 + " " + str2;
12 const combinedTemplate = `${str1} ${str2}`;
13 console.log(combined);
14 console.log(combinedTemplate);
15
16 // 3. Dostęp do znaków
17 console.log(str1[0]); // 'H'
18 console.log(str1.charAt(1)); // 'e'
19
20 // 4. Długość stringa
21 console.log(str1.length); // 13
22
23 // 5. Wycinanie fragmentów stringa
24 console.log(str1.slice(0, 5)); // 'Hello'
25 console.log(str1.substring(0, 5)); // 'Hello'
26 console.log(str1.substr(0, 5)); // 'Hello'
    (niezalecane, przestarzałe)
27
28 // 6. Znajdowanie tekstu
29 console.log(str1.indexOf("World")); // 7
30 console.log(str1.includes("Hello")); // true
31 console.log(str1.startsWith("Hello")); // true
32 console.log(str1.endsWith("!")); // true
33
34 // 7. Zamiana tekstu
35 console.log(str1.replace("World", "JavaScript"));
    // 'Hello, JavaScript!'
```

```
36 console.log(str1.replace(/world/i, "JavaScript"));
    // 'Hello, JavaScript!' (z regexem)
37
38 // 8. Dzielenie stringa
39 const splitExample = "jabłko,banan,wiśnia";
40 console.log(splitExample.split(","));
    // ['jabłko',
     'banan', 'wiśnia']
41
42 // 9. Zmiana wielkości liter
43 console.log(str1.toUpperCase());
    // 'HELLO, WORLD!'
44 console.log(str1.toLowerCase());
    // 'hello, world!'
45
46 // 10. Usuwanie białych znaków
47 const strWithSpaces = "Hello, World! ";
48 console.log(strWithSpaces.trim());
    // 'Hello,
     World!'
49 console.log(strWithSpaces.trimStart());
    // 'Hello,
     World! '
50 console.log(strWithSpaces.trimEnd());
    // 'Hello,
     World!'
51
52 // 11. Powtarzanie stringa
53 console.log("Hi! ".repeat(3));
    // 'Hi! Hi! Hi!'
54
55 // 12. Podział na znaki
56 console.log(Array.from("ABC"));
    // ['A', 'B', 'C']
57
58 // 13. Tworzenie stringa z tablicy
59 const arr = ["Jeden", "Dwa", "Trzy"];
60 console.log(arr.join(", "));
    // 'Jeden, Dwa, Trzy'
61
62 // 14. Porównywanie stringów
63 console.log("abc" === "abc");
    // true
64 console.log("abc" > "abd");
    // false (porównanie
     leksykalne)
65
66 // 15. Unicode i kody znaków
67 console.log("A".charCodeAt(0));
    // 65
68 console.log(String.fromCharCode(65));
    // 'A'
69
70 // 16. Praca z wieloliniowym tekstem
71 const multilineString = `
72 To jest pierwszy wiersz
73 A to drugi wiersz
74 I trzeci wiersz
75 `;
```

```
76 console.log(multilineString);
77
78 // 17. Sprawdzanie pustego stringa
79 console.log("").length == 0); // true
80 console.log(!""); // true
81
82 // 18. Tworzenie stringa z liczby
83 console.log((123).toString()); // '123'
84
85 // 19. Weryfikacja typu
86 console.log(typeof str1 == "string"); // true
87
```

Tablice:

```
1 // Najważniejsze operacje na tablicach w JavaScript
2
3 // 1. Tworzenie tablic
4 const arr1 = [1, 2, 3, 4, 5]; // Tablica liczb
5 const arr2 = ["jabłko", "banan", "wiśnia"]; // Tablica stringów
6 const arr3 = new Array(5).fill(0); // Tablica wypełniona zerami
7 console.log(arr1, arr2, arr3);
8
9 // 2. Dostęp do elementów
10 console.log(arr1[0]); // 1
11 console.log(arr2[arr2.length - 1]); // 'wiśnia'
12
13 // 3. Dodawanie i usuwanie elementów
14 arr1.push(6); // Dodaje na końcu
15 arr2.pop(); // Usuwa ostatni element
16 arr1.unshift(0); // Dodaje na początku
17 arr2.shift(); // Usuwa pierwszy element
18 console.log(arr1, arr2);
19
20 // 4. Długość tablicy
21 console.log(arr1.length); // 6
22
23 // 5. Sprawdzanie, czy to tablica
24 console.log(Array.isArray(arr1)); // true
25
26 // 6. Iteracja po tablicy
27 arr1.forEach((el, index) => console.log(`Element
${index}: ${el}`));
28
29 // 7. Mapowanie (tworzenie nowej tablicy)
30 const mapped = arr1.map(el => el * 2);
31 console.log(mapped); // [0, 2, 4, 6, 8, 10, 12]
32
33 // 8. Filtrowanie
34 const filtered = arr1.filter(el => el % 2 === 0);
35 console.log(filtered); // [0, 2, 4, 6]
36
37 // 9. Znajdowanie elementów
```

```
38 console.log(arr1.find(el => el > 3)); // 4
    (pierwszy spełniający warunek)
39 console.log(arr1.findIndex(el => el > 3)); // 4
    (indeks pierwszego spełniającego warunek)
40
41 // 10. Sprawdzanie obecności elementu
42 console.log(arr1.includes(3)); // true
43 console.log(arr1.indexOf(3)); // 3
44 console.log(arr1.lastIndexOf(3)); // 3
45
46 // 11. Sortowanie
47 const unsorted = [3, 1, 4, 1, 5];
48 unsorted.sort(); // Sortuje leksykalnie
49 console.log(unsorted); // [1, 1, 3, 4, 5]
50 unsorted.sort((a, b) => a - b); // Sortowanie
    numeryczne
51 console.log(unsorted); // [1, 1, 3, 4, 5]
52
53 // 12. Łączenie tablic
54 const combined = arr1.concat(arr2);
55 console.log(combined);
56
57 // 13. Dzielenie tablicy
58 const sliced = arr1.slice(2, 5); // Nowa tablica od
    indeksu 2 do 4
59 console.log(sliced);
60
61 // 14. Usuwanie lub zamiana elementów (splice)
62 const spliced = arr1.splice(2, 2, 99, 100); //
    Usuwa 2 elementy od indeksu 2 i wstawia 99, 100
63 console.log(arr1, spliced); // arr1 zmodyfikowane
64
65 // 15. Odwracanie i odwrócone tablice
66 console.log(arr1.reverse()); // Odwraca tablicę w
    miejscu
67
68 // 16. Płaskie tablice
69 const nested = [1, [2, 3], [4, [5, 6]]];
70 console.log(nested.flat()); // [1, 2, 3, 4, [5, 6]]
71 console.log(nested.flat(2)); // [1, 2, 3, 4, 5, 6]
72
73 // 17. Redukcja
74 const sum = arr1.reduce((acc, el) => acc + el, 0);
75 console.log(sum); // Suma elementów tablicy
76
77 // 18. Klonowanie tablic
```

```
78 const clone = [...arr1];
79 console.log(clone);
80
81 // 19. Usuwanie duplikatów
82 const duplicates = [1, 2, 2, 3, 3, 4];
83 const unique = [...new Set(duplicates)];
84 console.log(unique); // [1, 2, 3, 4]
85
86 // 20. Tworzenie tablic z zakresu (od 1 do 10)
87 const range = Array.from({ length: 10 }, (_, i) =>
  i + 1);
88 console.log(range);
89
90 // 21. Zamiana tablicy na string
91 console.log(arr2.join(", ")); // 'banan'
92
93 // 22. Praca z indeksami
94 arr1.forEach((value, index) => console.log(`Index
${index}: ${value}`));
95
96 // 23. Sprawdzanie warunków
97 console.log(arr1.every(el => el > 0)); // false
  (czy wszystkie elementy są większe od 0)
98 console.log(arr1.some(el => el > 0)); // true (czy
  jakiś element jest większy od 0)
99
```

Obiekty:

```
1 // Najważniejsze operacje na obiektach w JavaScript
2
3 // 1. Tworzenie obiektu
4 const person = {
5     name: "John",
6     age: 30,
7     isEmployed: true
8 };
9 console.log(person);
10
11 // 2. Dostęp do właściwości obiektu
12 console.log(person.name); // 'John'
13 console.log(person["age"]); // 30
14
15 // 3. Zmiana wartości właściwości
16 person.age = 31;
17 person["isEmployed"] = false;
18 console.log(person);
19
20 // 4. Dodawanie nowych właściwości
21 person.city = "New York"; // Dodanie właściwości
22 // 'city'
23 console.log(person);
24
25 // 5. Usuwanie właściwości
26 delete person.city;
27 console.log(person);
28
29 // 6. Sprawdzanie, czy właściwość istnieje
30 console.log("name" in person); // true
31 console.log("city" in person); // false
32
33 // 7. Iteracja po obiekcie (for...in)
34 for (let key in person) {
35     console.log(key, person[key]);
36 }
37
38 // 8. Zastosowanie Object.keys(), Object.values(),
39 // Object.entries()
40 console.log(Object.keys(person)); // ['name',
```

```
'age', 'isEmployed']  
39 console.log(Object.values(person)); // ['John', 31,  
false]  
40 console.log(Object.entries(person)); // [['name',  
'John'], ['age', 31], ['isEmployed', false]]  
41  
42 // 9. Kopiowanie obiektów  
43 const copyPerson = { ...person };  
44 console.log(copyPerson);  
45  
46 // 10. Łączenie obiektów  
47 const address = { city: "New York", zipCode:  
"10001" };  
48 const fullPerson = { ...person, ...address }; //  
Łączenie dwóch obiektów  
49 console.log(fullPerson);  
50  
51 // 11. Destrukturyzacja obiektów  
52 const { name, age } = person;  
53 console.log(name, age); // 'John', 31  
54  
55 // 12. Funkcja jako właściwość obiektu (metoda)  
56 const car = {  
57   brand: "Toyota",  
58   model: "Corolla",  
59   greet: function() {  
60     return `Hello, I am a ${this.brand}  
${this.model}`;  
61   }  
62 };  
63 console.log(car.greet()); // 'Hello, I am a Toyota  
Corolla'  
64  
65 // 13. Przekształcanie obiektu na string  
66 const jsonString = JSON.stringify(person);  
67 console.log(jsonString); //  
'{"name": "John", "age": 31, "isEmployed": false}'  
68  
69 // 14. Parsowanie stringa na obiekt  
70 const parsedObject = JSON.parse(jsonString);  
71 console.log(parsedObject); // {name: "John", age:  
31, isEmployed: false}  
72
```



Instrukcje Warunkowe

Instrukcje warunkowe w [JavaScript](#) pozwalają na wykonywanie określonych bloków kodu w zależności od spełnienia warunków logicznych. W JavaScript mamy kilka podstawowych typów instrukcji warunkowych, które umożliwiają kontrolowanie przepływu programu.

Podstawowe instrukcje warunkowe:

`if, else if, else`

Służą do wykonywania kodu w zależności od spełnienia warunku logicznego.

```
1 const age = 18;
2
3 if (age >= 18) {
4   console.log("Możesz głosować.");
5 } else if (age >= 16) {
6   console.log("Możesz głosować z opiekunem.");
7 } else {
8   console.log("Nie możesz głosować.");
9 }
```

`switch`

Dobre rozwiązanie, gdy sprawdzamy jedną zmienną w wielu przypadkach.

```
1  const day = "Monday";
2
3  switch (day) {
4      case "Monday":
5          console.log("Początek tygodnia!");
6          break;
7      case "Friday":
8          console.log("Prawie weekend!");
9          break;
10     default:
11         console.log("Zwykły dzień.");
12 }
```

Skrócone instrukcje warunkowe:

Skrócona wersja **if-else**. Idealny do prostych decyzji.

```
1 const age = 18;
2 const message = age >= 18 ? "Pełnoletni" :
  "Niepełnoletni";
3 console.log(message); // "Pełnoletni"
```



Pętle

Pętle w JavaScript pozwalają na wielokrotne wykonanie fragmentu kodu. Są używane do iteracji po tablicach, obiektach, lub powtarzania działań aż do spełnienia określonego warunku.

Rodzaje pętli:

for

Klasyczna pętla z kontrolą liczby iteracji.

```
1  for (let i = 0; i < 5; i++) {  
2      console.log(`Iteracja ${i}`);  
3  }  
4 // Wynik: Iteracja 0, Iteracja 1, ..., Iteracja 4
```

while

Wykonuje blok kodu, dopóki warunek jest spełniony.

```
1  let count = 0;  
2  
3  while (count < 5) {  
4      console.log(`Licznik: ${count}`);  
5      count++;  
6  }
```

do...while

Podobna do while, ale kod wykonuje się przynajmniej raz (warunek sprawdzany na końcu).

```
1 let count = 0;
2
3 do {
4     console.log(`Licznik: ${count}`);
5     count++;
6 } while (count < 5);
```

for...of

Idealna do iteracji po elementach tablicy, ciągach znaków lub innych iterowalnych strukturach danych.

```
1 const colors = ["red", "green", "blue"];
2
3 for (const color of colors) {
4     console.log(color);
5 }
6 // Wynik: red, green, blue
```

for...in

Służy do iteracji po kluczach obiektu (niezalecane dla tablic).

```
1 const user = { name: "John", age: 25 };
2
3 for (const key in user) {
4     console.log(` ${key}: ${user[key]}`);
5 }
6 // Wynik: name: John, age: 25
```

+ oczywiście pętle tablicowe ;)

Data i Czas

JavaScript oferuje wbudowane mechanizmy do pracy z datami i czasem za pomocą obiektu **Date** oraz dodatkowych metod i bibliotek. Możesz tworzyć, manipulować i formatować daty oraz wykonywać obliczenia czasowe.

```
1 // Tworzenie dat
2 const now = new Date(); // Obecna data i czas
3 const specificDate = new Date(2023, 10, 30); //  
Konkretna data: 30 listopada 2023
4 const fromString = new Date("2024-12-25T10:00:00");  
// Data z ciągu znaków
5 const fromTimestamp = new Date(1672531200000); //  
Data ze znacznika czasu
6
7 console.log("Obecna data:", now);
8 console.log("Konkretna data:", specificDate);
9 console.log("Data z ciągu:", fromString);
10 console.log("Data z timestampu:", fromTimestamp);
11
12 // Pobieranie elementów daty
13 console.log("Rok:", now.getFullYear());
14 console.log("Miesiąc (0-11):", now.getMonth());
15 console.log("Dzień miesiąca:", now.getDate());
16 console.log("Godzina:", now.getHours());
17 console.log("Minuta:", now.getMinutes());
18 console.log("Sekunda:", now.getSeconds());
19 console.log("Dzień tygodnia (0 = niedziela):",
now.getDay());
20
21 // Ustawianie elementów daty
22 const modifiedDate = new Date();
23 modifiedDate.setFullYear(2025);
24 modifiedDate.setMonth(5); // Czerwiec
25 modifiedDate.setDate(15); // 15 dzień
26 console.log("Zmodyfikowana data:", modifiedDate);
27
28 // Dodawanie/odejmowanie czasu
29 const addedDays = new Date();
30 addedDays.setDate(addedDays.getDate() + 7); //  
Dodaj 7 dni
31 console.log("Data po dodaniu 7 dni:", addedDays);
32
33 // Różnica między datami
34 const startDate = new Date("2023-11-30");
35 const endDate = new Date("2023-12-25");
```

```
36 const diffInMs = endDate - startDate; // Różnica w  
37 milisekundach  
38 const diffInDays = diffInMs / (1000 * 60 * 60 *  
39 24); // Przelicz na dni  
40 console.log("Różnica w dniach:", diffInDays);  
41  
42 // Formatowanie dat  
43 console.log("ISO 8601:", now.toISOString());  
44 console.log("Lokalna data:",  
45 now.toLocaleDateString("pl-PL"));  
46 console.log("Lokalny czas:",  
47 now.toLocaleTimeString("pl-PL"));  
48 console.log("Data i czas lokalnie:",  
49 now.toLocaleString("pl-PL"));  
50  
51 // Formatowanie za pomocą Intl.DateTimeFormat  
52 const formatter = new Intl.DateTimeFormat("pl-PL",  
53 {  
54 year: "numeric",  
55 month: "long",  
56 day: "numeric",  
57 weekday: "long",  
58 hour: "2-digit",  
59 minute: "2-digit",  
60 second: "2-digit"  
61 });  
62 console.log("Sformatowana data (Intl):",  
63 formatter.format(now));  
64  
65 // Wyciąganie timestampu  
66 console.log("Timestamp (milisekundy od 1970):",  
67 now.getTime());  
68  
69 // Tworzenie zegara czasu rzeczywistego  
70 setInterval(() => {  
71     console.log("Aktualny czas:", new  
72 Date().toLocaleTimeString("pl-PL"));  
73 }, 1000);  
74  
75
```



Funkcje

Funkcje w JavaScript to podstawowy sposób organizacji i ponownego użycia kodu. Mogą przyjmować argumenty, zwracać wyniki i istnieją w różnych formach, od klasycznych deklaracji po nowoczesne funkcje strzałkowe.

Rodzaje funkcji:

Funkcje deklarowane (Function Declaration):

Funkcje definiowane za pomocą słowa kluczowego function. Można ich używać przed ich deklaracją (hoisting).

```
1  function greet(name) {  
2      return `Cześć, ${name}!`;  
3  }  
4  
5  console.log(greet("Jan")); // "Cześć, Jan!"
```

Funkcje wyrażone (Function Expression):

Funkcja przypisana do zmiennej. Nie są poddawane hoistingowi.

```
1 const greet = function(name) {  
2   return `Cześć, ${name}!`;  
3 };  
4  
5 console.log(greet("Jan")); // "Cześć, Jan!"
```

Funkcje strzałkowe (Arrow Functions):

Krótsza składnia funkcji, szczególnie przydatna w wyrażeniach lub metodach tablicowych.

```
1 const greet = (name) => `Cześć, ${name}!`;  
2  
3 console.log(greet("Jan")); // "Cześć, Jan!"
```

Domyślne wartości parametrów

Parametry funkcji mogą mieć domyślne wartości, jeśli nie zostaną przekazane.

```
1  function greet(name = "Gość") {  
2      return `Cześć, ${name}!`;  
3  }  
4  
5  console.log(greet()); // "Cześć, Gość!"  
6  console.log(greet("Anna")); // "Cześć, Anna!"
```

```
1  const greet = (name = "Gość") => `Cześć, ${name}!`;
```

Funkcje anonimowe

Anonimowe funkcje są często używane jako callbacki.

```
1  setTimeout(function() {  
2      console.log("Minęło 2 sekundy!");  
3  }, 2000);
```

setTimeout - wykona funkcje za 2 sekundy

Funkcje natychmiastowo wywoływane (IIFE)

Funkcje, które są wywoływane natychmiast po ich zdefiniowaniu.

```
1  (function() {  
2      console.log("To działa od razu!");  
3  })();
```



Klasy

Klasy w JavaScript to sposób na definiowanie struktur obiektowych, które łączą dane (właściwości) z funkcjami (metodami). Klasy zostały wprowadzone w ES6 (ECMAScript 2015) i są uproszczonym

sposobem pracy z prototypami.

Tworzenie klasy

Klasy są definiowane za pomocą słowa kluczowego class.

```
1  class Person {
2      constructor(name, age) {
3          this.name = name; // Właściwość klasy
4          this.age = age;   // Właściwość klasy
5      }
6
7      // Metoda klasy
8      greet() {
9          return `Cześć, jestem ${this.name} i mam
10         ${this.age} lat.`;
11     }
12
13    // Tworzenie instancji klasy
14    const person1 = new Person("Jan", 30);
15    console.log(person1.greet()); // "Cześć, jestem Jan
16         i mam 30 lat."
```

Konstruktor (constructor)

constructor to specjalna metoda wywoływana automatycznie przy tworzeniu obiektu klasy. Służy do inicjalizacji właściwości obiektu.

```
1     constructor(name, age) {
2         this.name = name; // Właściwość klasy
3         this.age = age;   // Właściwość klasy
4     }
```

Dodawanie metod

Metody są funkcjami definiowanymi wewnątrz klasy. Można ich używać na instancjach klasy.

```
1     // Metoda klasy
2     greet() {
3         return `Cześć, jestem ${this.name} i mam
4             ${this.age} lat.`;
```

Właściwości instancji

Właściwości to dane przypisane do konkretnej instancji klasy.

```
1 // Tworzenie instancji klasy
2 const person1 = new Person("Jan", 30);
3 console.log(person1.greet()); // "Cześć, jestem Jan
i mam 30 lat."
```

W JavaScript, pola prywatne w klasach zostały wprowadzone w **ES2022** za pomocą symbolu **#**. Pola prywatne są dostępne tylko wewnątrz klasy i nie mogą być bezpośrednio dostępne spoza niej.

```
1  class Person {
2      #name; // Prywatne pole
3
4      constructor(name, age) {
5          this.#name = name; // Inicjalizacja pola
6          // prywatnego
7          this.age = age;    // Publiczne pole
8      }
9
10     // Metoda klasy odczytująca pole prywatne
11     getName() {
12         return this.#name;
13     }
14
15     // Metoda klasy modyfikująca pole prywatne
16     setName(newName) {
17         this.#name = newName;
18     }
19
20     const person = new Person("Jan", 30);
21     console.log(person.getName()); // "Jan"
22     person.setName("Anna");
23     console.log(person.getName()); // "Anna"
24
25     // Próba dostępu do pola prywatnego z zewnątrz
26     // (błąd):
27     console.log(person.#name); // Błąd: Private field
28     '#name' must be declared in an enclosing class
```



DOM (Document Object Model)

DOM (Document Object Model) to interfejs programistyczny dla dokumentów [HTML](#) i [XML](#).

Pozwala na dynamiczny dostęp do zawartości, struktury i stylów stron internetowych oraz ich modyfikację w czasie rzeczywistym za pomocą JavaScript.

Jak działa DOM ?

DOM reprezentuje strukturę strony jako **drzewo węzłów**. Każdy element HTML (np. [`<div>`](#), [`<p>`](#), [`<h1>`](#)) jest węzłem w tym drzewie. Dzięki temu JavaScript może manipulować:

- elementami strony (dodawać, usuwać, zmieniać),
- tekstem (modyfikować zawartość tekstową),
- atrybutami (np. id, class, src).

Kluczowe metody DOM:

Pobieranie elementów:

- `document.getElementById('id')` - pobiera element o określonym id.
- `document.querySelector('selector')` - pobiera pierwszy element pasujący do selektora CSS.
- `document.querySelectorAll('selector')` - pobiera wszystkie elementy pasujące do selektora CSS.

Tworzenie i modyfikacja elementów:

- `document.createElement('tag')` - tworzy nowy element HTML.
- `element.appendChild(node)` - dodaje nowy element jako dziecko.
- `element.innerHTML` - ustawia lub zwraca zawartość HTML wewnątrz elementu.
- `element.textContent` - ustawia lub zwraca tekst wewnątrz elementu (bez HTML).

Usuwanie elementów:

- `element.removeChild(node)` - usuwa dziecko określonego elementu.
- `element.remove()` - usuwa element z DOM.

Manipulacja atrybutami:

- `element.setAttribute('attribute', 'value')` - ustawia nowy atrybut lub modyfikuje istniejący.
- `element.getAttribute('attribute')` - zwraca wartość atrybutu.
- `element.classList.add('class') / .remove('class')` - dodaje lub usuwa klasy CSS.

Przykłady:

Zmienianie tekstu nagłówka:

```
1 const heading = document.getElementById('my-  
heading');  
2 heading.textContent = 'Nowy tekst nagłówka!';
```

Dodawanie nowego elementu:

```
1 const newDiv = document.createElement('div');  
2 newDiv.textContent = 'To jest nowy element!';  
3 document.body.appendChild(newDiv);
```

Zmiana koloru tła przy kliknięciu:

```
1 const button = document.querySelector('button');
2 button.addEventListener('click', () => {
3     document.body.style.backgroundColor =
4         'lightblue';
5});
```

Dlaczego DOM jest ważny?

- Umożliwia dynamiczne zmiany na stronie bez jej przeładowywania.
- Jest podstawą interaktywności w nowoczesnych aplikacjach webowych.
- Dzięki DOM JavaScript może reagować na zdarzenia (np. kliknięcia, przewijanie).

Asynchroniczność

Asynchroniczność w JavaScript pozwala wykonywać kod w tle, nie blokując głównego wątku programu (tzw. **Event Loop**). Jest to kluczowe dla aplikacji webowych, które muszą reagować na zdarzenia użytkownika, ładować dane z serwera czy obsługiwać animacje bez opóźnień.

Dlaczego asynchroniczność jest ważna?

JavaScript działa w jednym wątku (tzw. **single-threaded**). Gdyby każda operacja trwała zbyt długo (np. oczekивание на odpowiedź z serwera), strona przestawałaby odpowiadać. Dzięki asynchroniczności można:

- wykonywać długotrwałe operacje (np. pobieranie danych) w tle,
- reagować na interakcje użytkownika bez opóźnień,
- poprawić wydajność i wrażenia użytkownika.

Mechanizmy asynchroniczności w JavaScript

Funkcje zwrotne (Callbacks): Funkcje przekazywane jako argumenty do innych funkcji i wywoływane później, gdy zadanie jest zakończone.

```
1  setTimeout(() => {  
2      console.log('To wyświetli się po 2  
3      sekundach!');  
4  }, 2000);
```

! Problem: Gdy wiele funkcji zwrotnych zależy od siebie, może dojść do tzw. **callback hell** (głębokie zagnieżdżenie kodu).

Obietnice (Promises): Obiekt reprezentujący wartość, która będzie dostępna w przyszłości (z sukcesem lub błędem). Promise ma trzy stany:

- **Pending** - w trakcie wykonywania.
- **Resolved** - zakończone sukcesem.
- **Rejected** - zakończone błędem.

```
1  const fetchData = new Promise((resolve, reject) =>
2  {
3      const success = true;
4      if (success) {
5          resolve('Dane zostały pobrane!');
6      } else {
7          reject('Błąd podczas pobierania danych.');
8      }
9
10     fetchData
11         .then(result => console.log(result))
12         .catch(error => console.error(error));
```

Async/Await: Syntaktyczny cukier (syntactic sugar) dla pracy z obietnicami, który sprawia, że kod wygląda bardziej jak synchroniczny.

```
1  async function fetchData() {  
2      try {  
3          const response = await fetch('https://  
api.example.com/data');  
4          const data = await response.json();  
5          console.log(data);  
6      } catch (error) {  
7          console.error('Błąd:', error);  
8      }  
9  }  
10  
11  fetchData();
```

Event Loop i Web API

JavaScript korzysta z mechanizmu **Event Loop**, który obsługuje asynchroniczne operacje. Proces wygląda tak:

1. Kod synchroniczny wykonywany jest najpierw.
2. Operacje asynchroniczne (np. `setTimeout`, `fetch`) trafiają do Web API i wykonują się w tle.
3. Gdy operacja asynchroniczna się zakończy, jej callback trafia do **kolejki zadań** (Task Queue).
4. Event Loop sprawdza, czy główny wątek jest wolny, i wykonuje zadania z kolejki.

Diagram (w skrócie):

1. Kod synchroniczny → 2. Web API → 3. Task Queue → 4. Event Loop.

Praktyczne zastosowania asynchroniczności:

- Pobieranie danych z API (`fetch`).
- Obsługa zdarzeń (np. kliknięcia).

- Opóźnienia i animacje (`setTimeout`, `setInterval`).
- Operacje na dużych zbiorach danych bez blokowania interfejsu użytkownika (Web Workers).

Zadania do ćwiczenia

1. Zmiana tekstu po kliknięciu

Napisz kod, który po kliknięciu przycisku zmienia tekst w nagłówku na "Tekst zmieniony!".

2. Tworzenie elementów

Stwórz przycisk, który po kliknięciu doda nowy akapit z losowym tekstem do sekcji na stronie.

3. Zegar na stronie

Napisz funkcję, która wyświetla aktualny czas na stronie i aktualizuje go co sekundę.

4. Zmiana koloru tła

Stwórz stronę, gdzie po kliknięciu przycisku tło zmienia kolor na losowy.

5. Filtr listy

Stwórz formularz z polem tekstowym. Po wpisaniu tekstu lista elementów powinna być filtrowana, aby pokazywała tylko te, które zawierają podany tekst.

6. Kalkulator dodawania

Napisz prosty kalkulator, który przyjmuje dwie liczby od użytkownika i wyświetla ich sumę po kliknięciu przycisku.

7. Obsługa API

Pobierz dane z dowolnego publicznego API (np. JSONPlaceholder) i wyświetl na stronie listę elementów.

8. Licznik kliknięć

Stwórz przycisk, który liczy, ile razy został kliknięty, i wyświetla tę liczbę obok siebie.

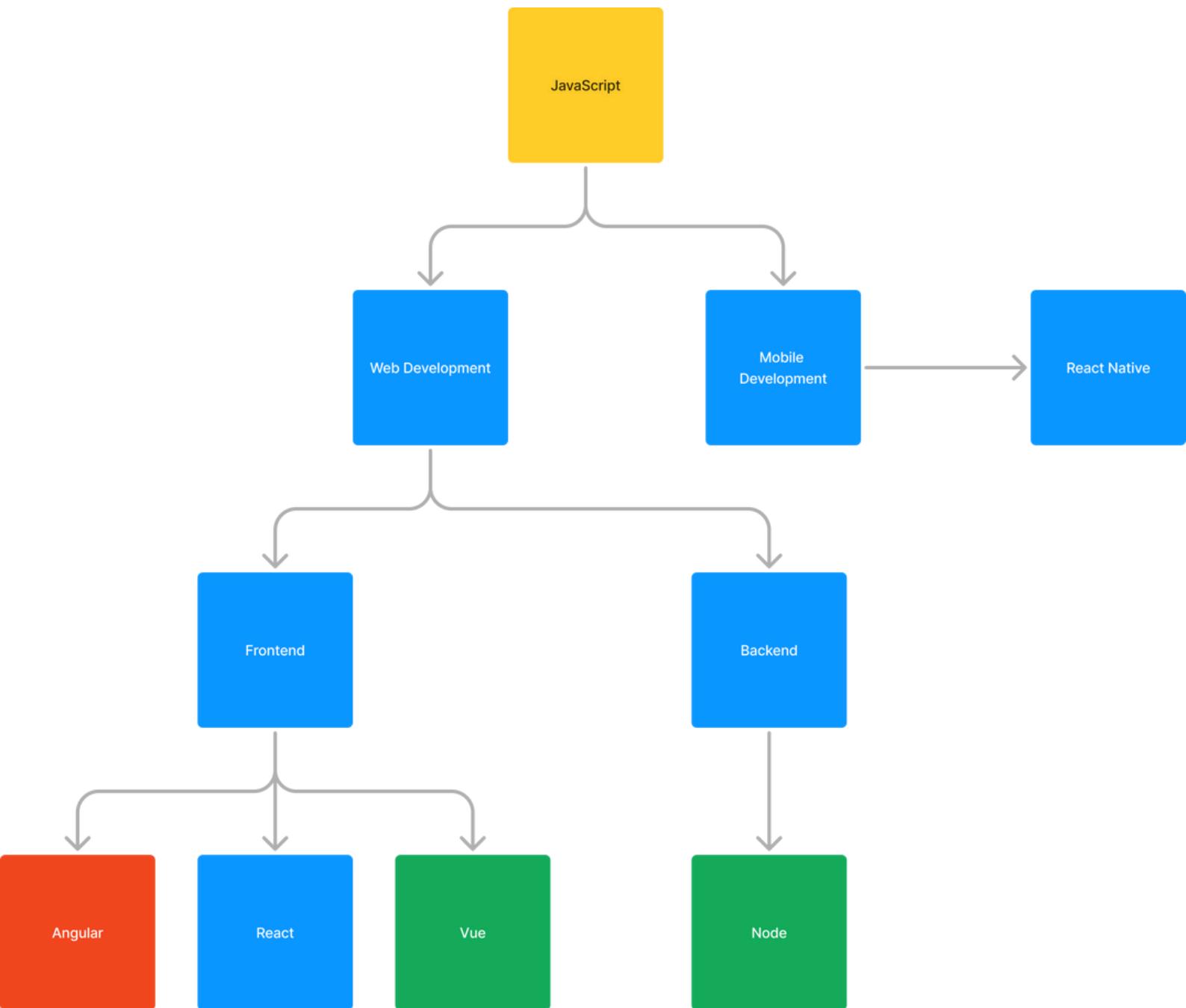
9. Ukrywanie i pokazywanie elementów

Napisz funkcję, która po kliknięciu przycisku ukrywa lub pokazuje element (np. obrazek lub tekst).

10. Weryfikacja formularza

Stwórz formularz z polem tekstowym i przyciskiem. Dodaj walidację, która wyświetli komunikat błędu, jeśli pole jest puste po kliknięciu przycisku.

Roadmap JavaScript Developer



Podstawy JavaScript

- HTML, CSS (podstawy)
- Vanilla JavaScript (DOM, Events, ES6+)
- Git i GitHub

Narzędzia developerskie

- Konsola przeglądarki (DevTools)
- Node.js (uruchamianie JS poza przeglądarką)
- npm/yarn (zarządzanie pakietami)

Frontend

- Frameworki CSS: TailwindCSS, Bootstrap

- Frameworki JS: React, Vue.js, Angular
- Bundlery i buildery: Vite, Webpack, Parcel
- TypeScript (rozszerzenie JS o typowanie)

Backend

- Node.js (środowisko backendowe)
- Frameworki: Express.js, NestJS
- Bazy danych:
 - Relacyjne: PostgreSQL, MySQL
 - NoSQL: MongoDB, Firebase
- ORM: Sequelize, Mongoose, Prisma

Testing

- Testy jednostkowe: Jest, Vitest
- Testy E2E: Cypress, Playwright

Narzędzia i DevOps

- Konteneryzacja: Docker
- CI/CD: GitHub Actions, Jenkins
- Hosting: Vercel, Netlify, Heroku

Zaawansowane technologie i koncepty

- WebSockets (real-time): Socket.IO
- GraphQL (alternatywa dla REST)
- Serverless: AWS Lambda, Google Cloud Functions
- Microfrontendy

Optymalizacja i wydajność

- Lighthouse, Web Vitals
- Progressive Web Apps (PWA)
- Service Workers