

### Zmienne

#### Deklaracja zmiennej:

- **var** nazwa = wartość – deklaruje zmienną o zakresie funkcyjnym
- **let** nazwa = wartość – deklaruje zmienną o zakresie blokowym
- **const** nazwa = wartość – deklaruje stałą o zakresie blokowym, nie może ulec zmianie

### Deklaracja funkcji

```
function addition(a,b) {
  let result = a + b
  console.log( " Wynik dodawania: " + result)}
addition(1, 2)
addition(5, 2)
function multiply(a,b) {
  let result = a * b
  return result}
console.log( multiply(5, 10))
```

### Typy zmiennych - prymitywy

let num = 7	number
let str = " tekst	string
let flag = true	boolean
let info	null - specjalny prymityw, intencjonalnie nie zmienia wartości
let mySymbol = Symbol ("Sym ")	symbol
let bigint = 1n	bigint do przechowywania liczb większych niż Number.MAX_ SAFE_INTEGER

### Typy zmiennych - złożone

let obj = new Object()	Object
let arr = new Array()	obiekt Array
let date = new Date()	obiekt Date
let err = new Error()	obiekt Error

### Sprawdzanie typów danych - typeof

```
typeof 42; // "number"
typeof " Hello"; // " string "
typeof true; // " boolean"
typeof []; // " object "
typeof {}; // " object "
typeof null; // " object "
typeof undefined; // " undefined "
typeof function() {}; // " function"
```

### Sprawdzanie typów danych - instanceof

```
const arr = [];
arr instanceof Array; // true
const obj = {};
obj instanceof Object; // true
const today = new Date();
today instanceof Date; // true
```

### Sprawdzanie typów danych: porównywanie

```
typeof 42 === "number"; // true
typeof " Hello" === " string "; // true
typeof true === " boolean"; // true
```



### Sprawdzanie typów danych - Array.isArray()

```
Array.isArray([]); // true
Array.isArray({}); // false
Array.isArray("hello"); // false
```

### NaN - Not a Number

parseFloat("hello")	NaN
Math.sqrt(-1)	NaN
Math.sqrt(-1)	NaN
isNaN(NumberOfNaN)	true
isNaN(NaN)	true
isNaN(78)	false

### formatowanie z backtickami

Wstawianie wartości zmiennych:

```
let name = "John";
let age = 30;
let message = `My name is ${name} and I am ${age} years old.`;
console.log(message); // "My name is John and I am 30 years old."
```

Wyrażenia matematyczne:

```
let a = 5;
let b = 3;
let sum = `${a} + ${b} = ${a + b}`;
console.log(sum); // "5 + 3 = 8"
```

Wywołanie funkcji:

```
function greet(name) {
  return `Hello, ${name}!`;
}

let person = "Alice";
let greeting = greet(person);
```

### formatowanie z backtickami (cont)

```
> console.log(greeting); // "Hello, Alice!"

Warunkowe wyrażenia:

let num = 7;
let isEven = `${num} is ${num % 2 === 0 ? "even" : "odd"}`;
console.log(isEven); // "7 is odd"
```

### String

```
const s = 'string w apostrofach'
const str = "string w cudzysłowie"
const txt = `string
w wielu
liniach`
```

### Operacje na stringach

Właściwość length: Zwraca liczbę znaków w ciągu znaków.

```
let str = "Hello, World!";
console.log(str.length); // 13
```

Indeksowanie za pomocą nawiasów kwadratowych: Można uzyskać dostęp do poszczególnych znaków ciągu znaków, odwołując się do ich indeksu za pomocą nawiasów kwadratowych i liczby całkowitej reprezentującej pozycję znaku w ciągu (indeksowanie zaczyna się od zera).

```
let str = "Hello";
console.log(str[0]); // "H"
console.log(str[1]); // "e"
console.log(str[4]); // "o"
```

Metoda charAt(index): Zwraca znak znajdujący się na określonym indeksie w ciągu znaków.

```
let str = "Hello, World!";
console.log(str.charAt(7)); // "W"
```

Metoda charCodeAt(): Metoda charCodeAt() zwraca wartość kodu Unicode znaku na



### Operacje na stringach (cont)

> określonym indeksie w ciągu znaków.

```
let str = "Hello";
console.log(str.charCodeAt(0)); // 72
console.log(str.charCodeAt(1)); // 101
```

Metoda `indexOf(searchValue, fromIndex)`: Znajduje indeks pierwszego wystąpienia określonej wartości w ciągu znaków.

```
let str = "Hello World";
console.log(str.indexOf("o")); // 4
console.log(str.indexOf("o", 5)); // 7 (rozpoczyna wyszukiwanie od indeksu 5)
console.log(str.indexOf("o", 10)); // -1 (nie znaleziono, rozpoczyna wyszukiwanie //od indeksu 10)
console.log(str.indexOf("World")); // 6
console.log(str.indexOf("JavaScript")); // -1 (nie znaleziono)
```

Metoda `lastIndexOf()`: Metoda `lastIndexOf()` znajduje ostatnie wystąpienie podciągu w ciągu znaków i zwraca indeks pierwszego znaku tego podciągu. Jeśli podciąg nie zostanie znaleziony, metoda zwraca -1.

```
let str = "Hello World";
console.log(str.lastIndexOf("o")); // 7
console.log(str.lastIndexOf("o", 7)); // 7 (rozpoczyna wyszukiwanie od indeksu 7 wstecz)
console.log(str.lastIndexOf("o", 3)); // 2 (rozpoczyna wyszukiwanie od indeksu 3 wstecz)
console.log(str.lastIndexOf("World")); // 6
console.log(str.lastIndexOf("JavaScript")); // -1 (nie znaleziono)
```

Metoda `slice(startIndex, endIndex)` zwraca fragment tekstu między dwoma indeksami (`startIndex` i `endIndex`). `slice()` działa włączając `startIndex`, ale wyłączając `endIndex`. Jeśli `startIndex` jest większy niż `endIndex`, metoda zamienia te wartości. Można również podać tylko `startIndex`, wtedy `slice()` zwróci fragment od `startIndex` do końca tekstu.

Metoda `slice()` jest jedyną metodą, która obsługuje ujemne indeksy w celu wycinania fragmentów tekstu od końca. `substring()` i `substr()` nie

### Operacje na stringach (cont)

> obsługują indeksów ujemnych i zamieniają je na 0.

```
let str = 'Hello, world!';
let sliced = str.slice(7, 12);
console.log(sliced); // "world"
```

Metoda `substring(startIndex, endIndex)`: Zwraca fragment ciągu znaków od indeksu `startIndex` do indeksu `endIndex` (bez końcowego indeksu). Metoda `substring(startIndex, endIndex)` działa podobnie jak `slice()`, z tą różnicą, że automatycznie zamienia wartości, jeśli `startIndex` jest większy niż `endIndex`. Tak samo jak `slice()`, `substring()` zwraca fragment tekstu między indeksami, włączając `startIndex`, ale wyłączając `endIndex`. Można również podać tylko `startIndex`, wtedy `substring()` zwróci fragment od `startIndex` do końca tekstu.

```
let str = "Hello, World!";
console.log(str.substring(7, 12)); // "World"
```

Metoda `substr(startIndex, length)` zwraca fragment tekstu o określonej długości, zaczynając od `startIndex`. Drugim argumentem jest `length`, który określa, ile znaków ma zostać zwróconych. Jeśli `length` nie jest podane, `substr()` zwraca fragment tekstu od `startIndex` do końca tekstu.

```
let str = 'Hello, world!';
let substr = str.substr(7, 5);
console.log(substr); // "world"
```

Metoda `search()` szuka dopasowania wzorca w ciągu znaków i zwraca indeks pierwszego dopasowania.

```
let str = "Hello World";
console.log(str.search(/o/)); // 3 (znaleziono dopasowanie na indeksie 3)
console.log(str.search("World")); // 6 (znaleziono dopasowanie na indeksie 6)
console.log(str.search(/JavaScript/)); // -1 (nie znaleziono dopasowania)
```

Metoda `match()` znajduje wszystkie dopasowania wzorca w ciągu znaków i zwraca tablicę zawierającą dopasowania.



### Operacje na stringach (cont)

```
> let str = "Hello World";
console.log(str.match(/lo/)); // ["lo"] (znaleziono dopasowanie "lo")
console.log(str.match(/o/g)); // ["o", "o"] (znaleziono wszystkie dopasowania "o")
console.log(str.match(/JavaScript/)); // null (nie znaleziono dopasowania)
```

Metoda `trim()` jest używana do usuwania białych znaków (spacje, tabulatory, znaki nowej linii) z początku i końca ciągu znaków. Oto przykład użycia metody `trim()`:

```
let str = " Hello World ";
console.log(str.trim()); // "Hello World" (usunięcie białych znaków z początku i końca)
let str2 = " Hello \n World ";
console.log(str2.trim()); // "Hello \n World" (białe znaki w środku nie są usuwane)
```

Metoda `toLowerCase()`: Zmienia wszystkie znaki w ciągu znaków na małe litery.

```
let str = "Hello, World!";
console.log(str.toLowerCase()); // "hello, world!"
```

Metoda `toUpperCase()`: Zmienia wszystkie znaki w ciągu znaków na wielkie litery.

```
let str = "Hello, World!";
console.log(str.toUpperCase()); // "HELLO, WORLD!"
```

Metoda `split(separator)`:

```
let str = 'Hello, world!';
let splitted = str.split(', ');
console.log(splitted); // ["Hello", "world!"]
```

Metoda `replace(searchValue, replaceValue)`:

```
let str = 'Hello, world!';
let replaced = str.replace('world', 'universe');
console.log(replaced); // "Hello, universe!"
```

### Łączenie stringów

Operator `+`: Można użyć operatora `+` do konkatencji dwóch ciągów znaków.

```
let str1 = " Hel lo";
let str2 = " Wor ld";
let result = str1 + " " + str2;
console.log(result); // " Hello World"
```

Metoda `concat()`: Metoda `concat()` pozwala na łączenie wielu ciągów znaków.

```
let str1 = " Hel lo";
let str2 = " Wor ld";
let result = str1.concat(" ", str2);
console.log(result); // " Hello World"
```

Metoda `join()`: Metoda `join()` pozwala na łączenie elementów tablicy w jeden ciąg znaków, używając określonego separatora.

```
let arr = ["He llo ", " Wor ld"];
let result = arr.join(" ");
console.log(result); // " Hello World"
```

### Wyrażenia regularne

- Tworzenie wyrażeń regularnych: Wyrażenia regularne można utworzyć za pomocą dwóch sposobów: za pomocą literałów `/pattern/` lub przy użyciu konstruktora `RegExp` (`'pattern'`).  
Na przykład: `/hello/` lub `new RegExp('hello')`.  

```
let regex1 = /hello/; // za pomocą literału
let regex2 = new RegExp('hello'); // za pomocą konstruktora RegExp
```
- Metody do pracy z wyrażeniami regularnymi: Metody takie jak `test()`, `exec()`, `match()`, `search()`, `replace()`, `split()` itp. mogą być wykorzystywane do interakcji z wyrażeniami regularnymi i manipulacji tekstami.  
`test()`:  

```
let regex = /hello/;
let text = 'Hello, world!';
```

### Wyrażenia regularne (cont)

```
> let result = regex.test(text);
console.log(result); // false
exec():
let regex = /lorem/;
let text = 'Lorem ipsum dolor sit amet.';
let result = regex.exec(text);
console.log(result); // ['lorem', index: 0, input: 'Lorem ipsum dolor sit amet.',
// groups: undefined]
split():
let regex = /\s+/;
let text = 'Hello world!';
let result = text.split(regex);
console.log(result); // ['Hello', 'world!']
```

3. Składnia wzorców:  
Wzorce wyrażeń regularnych składają się z literałów, znaków specjalnych i zestawów znaków. Przykłady: /hello/ - dopasowuje bezpośrednio słowo "hello", /[aeiou]/ - dopasowuje dowolny samogłoskę.

```
let regex = /hello/;
let text = 'Hello, world!';
let result = regex.test(text);
console.log(result); // false
let regex = /[aeiou]/;
let text = 'Hello, world!';
let result = text.match(regex);
console.log(result); // ['e', 'o', 'o']
```

4. Znaki specjalne:  
Wyrażenia regularne wykorzystują znaki specjalne, takie jak ^, \$, ., \*, +, ?, |, \, (), [], {}, itd., które mają specjalne znaczenie i służą

### Wyrażenia regularne (cont)

> do definiowania różnych wzorców dopasowania.

Znaki specjalne:

Kropka .: Dopasowuje dowolny znak, z wyjątkiem nowej linii.

Znak zapytania ?: Oznacza, że poprzedni znak lub grupa może wystąpić zero lub jeden raz.

Gwiazdka \*: Oznacza, że poprzedni znak lub grupa może wystąpić zero lub więcej razy.

Plus +: Oznacza, że poprzedni znak lub grupa musi wystąpić co najmniej raz.

Nawiasy (): Tworzą grupę dopasowania.

5. Modifikatory:  
Modifikatory są dodawane do wzorca wyrażenia regularnego w celu wpływania na sposób dopasowywania.

Przykłady: g - dopasowanie globalne (wszystkie wystąpienia), i - dopasowanie bez względu na wielkość liter, m - wielolinijkowe dopasowanie.

### falsy values

```
" " // pusty string
0, -0
0n // 0 jako BigInt
NaN
null
undefined
false
```

To wartości, które poddane konwersji do boolean będą miały wartość false



### Tablica

kolekcja elementów – obiekt, zmienna, w której są inne zmienne

```
let a = [3, „tekst”, 99, 12]
```

```
let b = new Array(10, 4, " txt ")
```

```
let c = Array(10, 4, " txt ")
```

```
let d = Array.of(5) // pojedyncza wartość traktowana jest jako jeden element,
```

a nie ustawienie wartości length`

```
tab[1] // element z tablicy, indeksy liczone są od 0
```

Tablica (Array) jest jednym z podstawowych typów danych w języku JavaScript,

który służy do przechowywania kolekcji elementów w uporządkowany sposób. Oto kilka dodatkowych informacji o tablicach:

- Indeksowanie: Elementy w tablicy są indeksowane numerycznie, zaczynając od zera.

Możesz uzyskać dostęp do elementów za pomocą ich indeksów.

Przykład:

```
const fruits = ['apple', 'banana', 'orange'];
```

```
console.log(fruits[0]); // "apple"
```

```
console.log(fruits[2]); // "orange"
```

- Długość tablicy: Właściwość length pozwala określić liczbę elementów w tablicy.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
```

```
console.log(numbers.length); // 5
```

- Dodawanie i usuwanie elementów: Tablice mają wiele metod, które umożliwiają dodawanie i usuwanie elementów. Niektóre z tych metod to: push(), pop(), shift(), unshift(), splice() itp.

Przykład (dodawanie elementu na koniec tablicy):

```
const fruits = ['apple', 'banana'];
```

```
fruits.push('orange');
```

```
console.log(fruits); // ["apple", "banana", "orange"]
```

Przykład (usuwanie elementu z początku tablicy):

```
const numbers = [1, 2, 3, 4, 5];
```

### Tablica (cont)

```
numbers.shift();
```

```
console.log(numbers); // [2, 3, 4, 5]
```

- Iteracja po tablicy: Możesz iterować po elementach tablicy za pomocą pętli, takich jak pętla for lub metody forEach(), map(), filter(), reduce() itp.

Przykład (użycie metody forEach()):

```
const fruits = ['apple', 'banana', 'orange'];
```

```
fruits.forEach(fruit => {
```

```
  console.log(fruit);
```

```
});
```

- Metoda fill(value, start, end): Metoda fill() wypełnia wszystkie elementy tablicy określoną wartością. Można również określić opcjonalne parametry start i end, które wskazują zakres indeksów tablicy, które mają być wypełnione.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
```

```
numbers.fill(0);
```

```
console.log(numbers); // [0, 0, 0, 0, 0]
```

```
const fruits = ['apple', 'banana', 'orange'];
```

```
fruits.fill('pear', 1, 2);
```

```
console.log(fruits); // ['apple', 'pear', 'orange']
```

- Metoda find(callback): Metoda find() zwraca pierwszy element tablicy, który spełnia warunek zdefiniowany przez funkcję zwrtną (callback).

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
```

```
const found = numbers.find(num => num > 3);
```

```
console.log(found); // 4
```

- Metoda findIndex(callback): Metoda findIndex() zwraca indeks pierwszego elementu

tablicy, który spełnia warunek zdefiniowany przez funkcję zwrtną (callback).

Przykład:



By sigeurd

[cheatography.com/signeur/](https://cheatography.com/signeur/)

Not published yet.

Last updated 2nd July, 2023.

Page 6 of 49.

Sponsored by [Readable.com](https://readable.com)

Measure your website readability!

<https://readable.com>

### Tablica (cont)

```
const numbers = [1, 2, 3, 4, 5];
const index = numbers.findIndex(num => num > 3);
console.log(index); // 3
```

- Metoda `copyWithin(target, start, end)`: Metoda `copyWithin()` kopiuje elementy z określonego zakresu tablicy i wkleja je na pozycję początkową wewnątrz tej samej tablicy.

Parametr `target` wskazuje pozycję docelową, a parametry `start` i `end` określają zakres elementów do skopiowania.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
numbers.copyWithin(0, 3, 5);
console.log(numbers); // [4, 5, 3, 4, 5]
```

- Metoda `filter(callback)`: Metoda `filter()` tworzy nową tablicę zawierającą elementy, dla których funkcja zwrotna (`callback`) zwraca wartość `true`.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

- Metoda `reduce(callback, initialValue)`: Metoda `reduce()` wykonuje funkcję zwrotną (`callback`) dla każdego elementu tablicy, redukując ją do pojedynczej wartości. Parametr `initialValue` jest opcjonalny i określa początkową wartość akumulatora.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) =>
  accumulator + currentValue, 0);
console.log(sum); // 15
```

- Metoda `map(callback)`: Metoda `map()` tworzy nową tablicę, zawierającą wyniki działania funkcji zwrotnej (`callback`) na każdym elemencie tablicy.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
```

### Tablica (cont)

```
const squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers); // [1, 4, 9, 16, 25]
```

- Metoda `every(callback)`: Metoda `every()` sprawdza, czy wszystkie elementy tablicy spełniają warunek zdefiniowany przez funkcję zwrotną (`callback`). Zwraca wartość `true`, jeśli wszystkie elementy spełniają warunek, w przeciwnym razie zwraca `false`.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // false
```

- Metoda `some(callback)`: Metoda `some()` sprawdza, czy przynajmniej jeden element tablicy spełnia warunek zdefiniowany przez funkcję zwrotną (`callback`). Zwraca wartość `true`, jeśli przynajmniej jeden element spełnia warunek, w przeciwnym razie zwraca `false`.

Przykład:

```
const numbers = [1, 2, 3, 4, 5];
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // true
```

- Inne metody tablicowe: Tablice mają wiele innych przydatnych metod, które ułatwiają manipulację i przetwarzanie danych. Niektóre z tych metod to: `join()`, `slice()`, `concat()`, `sort()`, `reverse()` itp.

Przykład (użycie metody `join()`):

```
const fruits = ['apple', 'banana', 'orange'];
const joinedFruits = fruits.join(', ');
console.log(joinedFruits); // "apple, banana, orange"
```



### Operacje na tablicach

```
length: Zwraca liczbę elementów w tablicy. - można też w ten sposób zmniejszyć wielkość tablicy
let fruits = ["apple", "banana", "orange"];
console.log(fruits.length); // Wyświetli: 3
let a = [1,2,3,4,5,6,7,8,9]
a.length = 3
console.log(a) // [1, 2, 3]
push(): Dodaje jeden lub więcej elementów na koniec tablicy i zwraca jej nową długość.
let fruits = ["apple", "banana"];
fruits.push("orange");
console.log(fruits); // Wyświetli: ["apple", "banana", "orange"]
pop(): Usuwa ostatni element z tablicy i zwraca ten element.
let fruits = ["apple", "banana", "orange"];
let removedFruit = fruits.pop();
console.log(removedFruit); // Wyświetli: "orange"
console.log(fruits); // Wyświetli: ["apple", "banana"]
delete: Usuwa określony element z tablicy, pozostawiając na jego miejscu pustą wartość (undefined).
let fruits = ["apple", "banana", "orange"];
delete fruits[1];
console.log(fruits); // Wyświetli: ["apple", undefined, "orange"]
concat(): Łączy dwie tablice lub więcej tablic i zwraca nową tablicę.
let fruits1 = ["apple", "banana"];
let fruits2 = ["orange", "grape"];
let combinedFruits = fruits1.concat(fruits2);
console.log(combinedFruits); // Wyświetli: ["apple", "banana", "orange", "grape"]
join(): Łączy wszystkie elementy tablicy w jeden ciąg znaków, używając
```

### Operacje na tablicach (cont)

```
> podanego separatora.
let fruits = ["apple", "banana", "orange"];
let joinedString = fruits.join(",");
console.log(joinedString); // Wyświetli: "apple, banana, orange"
indexOf(): Znajduje indeks pierwszego wystąpienia podanego elementu w tablicy. Zwraca -1, jeśli element nie jest obecny.
let fruits = ["apple", "banana", "orange"];
let index = fruits.indexOf("banana");
console.log(index); // Wyświetli: 1
fill: Wypełnia wszystkie elementy tablicy określoną wartością.
let numbers = [1, 2, 3, 4, 5];
numbers.fill(0);
console.log(numbers); // Wyświetli: [0, 0, 0, 0, 0]
forEach: Wykonuje podaną funkcję dla każdego elementu tablicy.
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(number) {
  console.log(number);
});
// Wyświetli:
// 1
// 2
// 3
// 4
// 5
reverse: Odwraca kolejność elementów w tablicy.
let fruits = ["apple", "banana", "orange"];
fruits.reverse();
console.log(fruits); // Wyświetli: ["orange", "banana", "apple"]
```



### Operacje na tablicach (cont)

```
> shift: Usuwa pierwszy element z tablicy i zwraca ten element.
let fruits = ["apple", "banana", "orange"];
let shiftedFruit = fruits.shift();
console.log(shiftedFruit); // Wyświetli: "apple"
console.log(fruits); // Wyświetli: ["banana", "orange"]
unshift: Dodaje jeden lub więcej elementów na początek tablicy i zwraca jej nową długość.
let fruits = ["banana", "orange"];
fruits.unshift("apple");
console.log(fruits); // Wyświetli: ["apple", "banana", "orange"]
splice(start, deleteCount, item1, item2...): Modyfikuje zawartość tablicy, usuwając, zamieniając lub dodając elementy.
let fruits = ["apple", "banana", "orange"];
fruits.splice(1, 1, "kiwi");
console.log(fruits); // Wyświetli: ["apple", "kiwi", "orange"]
```

### Typy zdarzeń DOM

Przeglądarka wywołuje wiele zdarzeń. Pełna lista jest dostępna na MDN, ale poniżej możesz znaleźć te najpopularniejsze:

zdarzenia myszki (MouseEvent): mousedown, mouseup, click, dblclick, mousemove, mouseover, mousewheel, mouseout, contextmenu

zdarzenia dotykowe (TouchEvent): touchstart, touchmove, touchend, touchcancel

zdarzenia klawiatury (KeyboardEvent): keydown, keypress, keyup

zdarzenia formularzy: focus, blur, change, submit

zdarzenia okna przeglądarki: scroll, resize, hashchange, load, unload

Khan Academy

### Fetch - pobieranie danych z serwera

```
fetch("https://swapi.dev/api/people/3/")
  .then( response => response.json() )
  .then( processData )
function processData( data ) {
    console.log( data );
    console.log( " Imię -> ", data.name );
    console.log( " Wysokość -> ", data.height );
    console.log( " Masa -> ", data.mass );
    console.log( " Kolor oczu -> ", data.eye_color );
}
```

### Obiekt window i event handler

```
onload:
window.onload = function() {
    // Kod wykonywany po załadowaniu strony
};
Ten event handler jest wywoływany po załadowaniu całej strony.
onresize:
window.onresize = function() {
    // Kod wykonywany po zmianie rozmiaru okna przeglądarki
};
Ten event handler jest wywoływany po zmianie rozmiaru okna przeglądarki.
onscroll:
window.onscroll = function() {
    // Kod wykonywany podczas przewijania strony
};
Ten event handler jest wywoływany podczas przewijania strony.
onkeydown:
window.onkeydown = function( event ) {
```

### Obiekt window i event handler (cont)

```
> // Kod wykonywany po wciśnięciu klawisza
if (event.key === 'Enter') {
  // Wykonaj pewną akcję po wciśnięciu klawisza Enter
}
};
```

Ten event handler jest wywoływany po wciśnięciu klawisza na klawiaturze.

```
onbeforeunload:
window.onbeforeunload = function(event) {
  // Kod wykonywany przed zamknięciem strony
  return 'Czy na pewno chcesz opuścić tę stronę?';
};
```

Ten event handler jest wywoływany przed zamknięciem strony.

### Obiekt window i zdarzenia

Zdarzenie load:

```
window.addEventListener('load', function() {
  // Kod wykonywany po załadowaniu strony
});
```

Ten event listener jest wywoływany po załadowaniu całej strony.

Zdarzenie resize:

```
window.addEventListener('resize', function() {
  // Kod wykonywany po zmianie rozmiaru okna przeglądarki
});
```

Ten event listener jest wywoływany po zmianie rozmiaru okna przeglądarki.

Zdarzenie scroll:

```
window.addEventListener('scroll', function() {
  // Kod wykonywany podczas przewijania strony
});
```

Ten event listener jest wywoływany podczas przewijania strony.

### Obiekt window i zdarzenia (cont)

Zdarzenie keydown:

```
window.addEventListener('keydown', function(event) {
  // Kod wykonywany po wciśnięciu klawisza
  if (event.key === 'Enter') {
    // Wykonaj pewną akcję po wciśnięciu klawisza Enter
  }
});
```

Ten event listener jest wywoływany po wciśnięciu klawisza na klawiaturze.

Zdarzenie beforeunload:

```
window.addEventListener('beforeunload', function(event) {
  // Kod wykonywany przed zamknięciem strony
  event.returnValue = 'Czy na pewno chcesz opuścić tę stronę?';
});
```

Ten event listener jest wywoływany przed zamknięciem strony.

### Zdarzenia myszy

```
<!DOCTYPE html>
<html>
<head>
  <title>Zdarzenia myszy</title>
</head>
<body>
  <p id="my-paragraph">Najedź na ten
  paragraf i kliknij go!</p>
  <script>
    const paragraph = document.getElementById('my-paragraph');
    // Obsługa zdarzenia najechania myszą
    paragraph.addEventListener('mouseover', function(event) {
      paragraph.style.backgroundColor = 'yellow';
    });
```



### Zdarzenia myszy (cont)

```
> // Obsługa zdarzenia opuszczenia myszy
paragraph.addEventListener('mouseleave', function(event) {
  paragraph.style.backgroundColor = 'white';
});
// Obsługa zdarzenia kliknięcia myszą
paragraph.addEventListener('click', function(event) {
  paragraph.textContent = 'Kliknięto!';
});
// Obsługa zdarzenia poruszania myszą
paragraph.addEventListener('mousemove', function(event) {
  const x = event.clientX;
  const y = event.clientY;
  paragraph.textContent = `X: ${x}, Y: ${y}`;
});
</script>
</body>
</html>
```

### Obiekt window

`alert(message)`: Wyświetla okno dialogowe z podaną wiadomością.

```
window.alert("Hello!");
```

`confirm(message)`: Wyświetla okno dialogowe z pytaniem i zwraca wartość logiczną w zależności od wyboru użytkownika.

```
const result = window.confirm("Czy jesteś pewien?");
console.log(result); // true jeśli kliknięto OK, false jeśli kliknięto Anuluj
```

`prompt(message, default)`: Wyświetla okno dialogowe z pytaniem o wprowadzenie danych i zwraca wprowadzoną wartość jako ciąg znaków.

```
const name = window.prompt("Jak masz na imię?");
console.log(name); // Wprowadzona wartość zostanie wyświetlona w konsoli
```

### Obiekt window (cont)

`> setTimeout(callback, delay)`: Wywołuje funkcję (callback) po określonym opóźnieniu (w milisekundach).

```
function greet() {
  console.log("Hello!");
}
window.setTimeout(greet, 2000); // Wywołuje funkcję greet po 2 sekundach
```

`setInterval(callback, interval)`: Wywołuje funkcję (callback) co określony interwał (w milisekundach), powtarzając ją w nieskończoność.

```
function printTime() {
  console.log(new Date());
}
window.setInterval(printTime, 1000); // Wywołuje funkcję printTime co sekundę
```

`window.location`: Obiekt dostarcza informacje o aktualnym adresie URL strony.

```
console.log(window.location.href); // Zwraca pełny adres URL strony
console.log(window.location.pathname); // Zwraca ścieżkę URL
console.log(window.location.host); // Zwraca nazwę hosta (np. www.example.com)
```

`innerWidth` i `innerHeight`: Właściwości `innerWidth` i `innerHeight` zwracają szerokość i wysokość obszaru widocznego w oknie przeglądarki, bez uwzględniania paska przewijania i innych elementów interfejsu przeglądarki.

```
console.log(window.innerWidth); // Zwraca szerokość obszaru widocznego w pikselach
console.log(window.innerHeight); // Zwraca wysokość obszaru widocznego w pikselach
```

`pageXOffset` i `pageYOffset`: Właściwości `pageXOffset` i `pageYOffset` zwracają przesunięcie strony w poziomie i pionie względem początku dokumentu. Są one przydatne do określania aktualnego położenia przewijania strony.

```
console.log(window.pageXOffset); // Zwraca przesunięcie strony w poziomie
console.log(window.pageYOffset); // Zwraca przesunięcie strony w pionie
```

`prompt(message, default)`: Metoda `prompt` wyświetla okno dialogowe z pytaniem o wprowadzenie danych. Przyjmuje dwa argumenty: wiadomość wyświetlaną w oknie



### Obiekt window (cont)

> dialogowym i wartość domyślną. Zwraca wprowadzoną wartość jako ciąg znaków lub

null, jeśli użytkownik kliknie Anuluj.

```
const result = window.prompt("Podaj swoje imię:", "Anonim");
```

```
console.log(result); // Zawiera wprowadzoną wartość lub null
```

### Przedrostek new

Przedrostek new jest używany w JavaScript do tworzenia nowych instancji obiektów na podstawie funkcji konstruktora. Gdy używasz słowa kluczowego new przed wywołaniem funkcji konstruktora, oznacza to, że chcesz utworzyć nowy obiekt na podstawie tego konstruktora.

Kiedy używamy new przed funkcją konstruktora, następuje kilka ważnych rzeczy:

1. Tworzony jest nowy pusty obiekt.
2. this wewnątrz funkcji konstruktora odnosi się do tego nowego obiektu.
3. Właściwości i metody są dodawane do tego obiektu przy użyciu składni this.propertyName lub this.methodName.
4. Jeśli funkcja konstruktora nie zwraca żadnej wartości, to automatycznie zwracany jest ten nowy obiekt.
5. Jeśli funkcja konstruktora zwraca inny obiekt (np. poprzez użycie instrukcji return), to zamiast nowego obiektu zwracany jest ten inny obiekt.

Bez przedrostka new funkcja konstruktora będzie po prostu zwykłą funkcją,

a nie zostanie utworzony nowy obiekt. W takim przypadku, this wewnątrz funkcji konstruktora odnosić się będzie do globalnego obiektu (np. window w przypadku przeglądarki) lub do kontekstu wywołania, w zależności od tego, w jakim kontekście zostanie wywołana.

### Obiekty

1. Object: Reprezentuje ogólny obiekt i jest podstawowym typem dla wszystkich innych obiektów JavaScript.

```
const obj = { name: 'John', age: 30 };
```

2. Array: Przechowuje dane w uporządkowanej sekwencji i zapewnia wiele funkcji do manipulacji tablicami.

```
const arr = [1, 2, 3, 4, 5];
```

3. String: Reprezentuje sekwencję znaków tekstowych i zapewnia wiele funkcji do manipulacji tekstami.

```
const str = 'Hello, World!';
```

4. Number: Reprezentuje wartość liczbową i zapewnia wiele funkcji matematycznych.

```
const num = 42;
```

5. Boolean: Reprezentuje wartość logiczną true lub false.

```
const bool = true;
```

6. Date: Reprezentuje datę i czas.

```
const currentDate = new Date();
```

```
console.log(currentDate); // Wyświetli aktualną datę i czas
```

7. Math: Dostarcza funkcje matematyczne i stałe.

```
const result = Math.sqrt(25);
```

8. RegExp: Reprezentuje wyrażenie regularne, używane do dopasowywania i manipulowania tekstami.

```
const regex = /hello/gi;
```

9. Function: Reprezentuje funkcję w JavaScript.

```
function greet(name) {
  console.log('Hello, ' + name!);
}
```

10. Error: Reprezentuje błąd wykonania programu.

```
const error = new Error('Something went wrong.');
```

11. JSON: Zapewnia funkcje do przekształcania danych JavaScript na format



By sigeud

[cheatography.com/sigeud/](https://cheatography.com/sigeud/)

Not published yet.

Last updated 2nd July, 2023.

Page 12 of 49.

Sponsored by [Readable.com](https://readable.com)

Measure your website readability!

<https://readable.com>

### Obiekty (cont)

> JSON (JavaScript Object Notation) i vice versa.  
`const data = JSON.parse({"name":"John","age":30});`

12. Map: Przechowuje pary klucz-wartość i zapewnia różne metody do manipulacji mapą.  
`const map = new Map();`  
`map.set('key', 'value');`

13. Set: Przechowuje unikalne wartości i zapewnia różne metody do manipulacji zbiorami.  
`const set = new Set();`  
`set.add(1);`

14. Promise: Reprezentuje asynchroniczne wykonanie operacji i obsługę wyników lub błędów.  
`const promise = new Promise((resolve, reject) => {`  
`setTimeout(() => {`  
`resolve('Done');`  
`}, 1000);`  
`});`

15. ArrayBuffer: Przechowuje sekwencję bajtów i zapewnia interfejs do manipulacji binarnymi danymi.  
`const buffer = new ArrayBuffer(16);`

16. DataView: Pozwala na odczyt i zapis danych binarnych w ArrayBuffer.  
`const buffer = new ArrayBuffer(16);`  
`const view = new DataView(buffer);`

17. XMLHttpRequest: służy do wykonywania asynchronicznych żądań HTTP w języku JavaScript. Umożliwia komunikację z serwerem i pobieranie danych w tle bez konieczności przeładowywania strony. Pozwala na wysyłanie żądań HTTP, odbieranie odpowiedzi, zarządzanie nagłówkami, przesyłanie danych i obsługę różnych typów odpowiedzi (np. tekst, JSON, XML).

### Obiekty (cont)

> `const xhr = new XMLHttpRequest();`

18. Window: reprezentuje globalne okno przeglądarki, w którym wyświetlana jest strona internetowa. Dostarcza wiele funkcji i właściwości związanych z interakcją użytkownika, manipulacją strukturą dokumentu, zarządzaniem historią przeglądarki itp. Udostępnia interfejs do operacji na elementach DOM, obsługę zdarzeń, manipulację historią przeglądarki, tworzenie nowych okien i ram, zarządzanie czasem (np. `setTimeout`, `setInterval`) oraz wiele innych funkcjonalności.  
`window.alert('Hello, World!');`

19. Document: reprezentuje aktualnie załadowany dokument HTML w przeglądarce. Zapewnia dostęp do elementów DOM na stronie, takich jak elementy HTML, style, zdarzenia itp. Udostępnia metody do manipulacji zawartością dokumentu, takie jak tworzenie, usuwanie i modyfikowanie elementów HTML, manipulowanie stylami, dodawanie i usuwanie zdarzeń. Pozwala na dostęp do różnych kolekcji elementów, takich jak elementy po tagu, po klasie, po identyfikatorze itp.  
`const element = document.createElement('div');`

### Serializacja danych

Serializacja danych to proces konwersji struktur danych, takich jak obiekty, tablice, liczby, ciągi znaków itp., na format, który można przesłać lub przechować, na przykład w formie ciągu znaków. Najczęściej używane formaty serializacji danych to JSON (JavaScript Object Notation) i XML (eXtensible Markup Language).

Serializacja danych jest użyteczna w wielu przypadkach, takich jak:

Przesyłanie danych przez sieć: Serializacja danych umożliwia przesłanie struktur danych między aplikacjami lub serwerami za pomocą protokołów komunikacyjnych, takich jak HTTP. Dane mogą być zamieniane na tekstowe reprezentacje, które są łatwe do przesłania i odczytania.

Przechowywanie danych: Serializacja danych umożliwia zapisanie struktur danych na dysku lub w bazie danych w celu późniejszego odczytania. Przechowywanie danych w formie serializowanej jest przydatne do utrwalania stanu aplikacji lub



### Serializacja danych (cont)

> przechowywania danych persistentnych.

Współpraca z innymi językami programowania: Serializacja danych pozwala na wymianę danych między aplikacjami napisanymi w różnych językach programowania.

Dzięki temu można zintegrować aplikacje działające na różnych platformach.

W JavaScript jednym z najczęściej używanych formatów serializacji danych jest

JSON. JavaScript dostarcza funkcje `JSON.stringify()` do konwersji obiektów na

JSON i `JSON.parse()` do konwersji JSON z powrotem na obiekty JavaScript. Przykład użycia:

```
const obj = { name: 'John', age: 30 };
// Serializacja obiektu do formatu JSON
const json = JSON.stringify(obj);
console.log(json); // {"name":"John","age":30}
```

// Deserializacja JSON do obiektu JavaScript

```
const parsedObj = JSON.parse(json);
console.log(parsedObj); // { name: 'John', age: 30 }
```

Serializacja danych jest powszechnie stosowaną techniką w programowaniu,

która umożliwia efektywną wymianę i przechowywanie danych w różnych formatach.

### Error

W JavaScript obiekty typu `Error` są wykorzystywane

do reprezentowania błędów i wyjątków. Obiekt `Error` zawiera informacje o błędzie, takie jak wiadomość opisująca rodzaj błędu oraz stos wywołań.

Przykład użycia obiektu `Error`:

```
try {
  // Kod, który może generować błąd
  throw new Error('To jest jakiś błąd');
} catch (error) {
  // Obsługa błędu
  console.log(error.message); // 'To jest jakiś błąd'
}
```

### Error (cont)

> W powyższym przykładzie używamy słowa kluczowego `throw`, aby wygenerować nowy

obiekt `Error` z określoną wiadomością. Następnie używamy bloku `catch` do

przechwycenia błędu i obsługi go. W tym przypadku po prostu wypisujemy

wiadomość błędu na konsoli.

Obiekty `Error` mogą być dziedziczone przez konkretne typy błędów, takie jak

`TypeError`, `ReferenceError`, `SyntaxError` itp. Na przykład:

```
try {
  // Kod, który może generować błąd typu TypeError
  throw new TypeError('To jest błąd typu TypeError');
} catch (error) {
  // Obsługa błędu
  console.log(error instanceof TypeError); // true
  console.log(error.message); // 'To jest błąd typu TypeError'
}
```

W powyższym przykładzie używamy obiektu `TypeError` do reprezentowania błędu

typu `TypeError`. Możemy sprawdzić typ błędu za pomocą operatora `instanceof` oraz

uzyskać jego wiadomość za pomocą właściwości `message`.

Obiekty `Error` posiadają również inne właściwości, takie jak `name` (nazwa błędu) i

`stack` (stos wywołań), które zawierają dodatkowe informacje diagnostyczne dotyczące

błędu.

Błędy i wyjątki są ważnym elementem w zarządzaniu błędami w JavaScript,

pozwalając na łatwiejszą identyfikację i obsługę różnych sytuacji błędnych w kodzie.

W języku JavaScript istnieje kilka wbudowanych typów błędów (`errors`), które dziedziczą

o klasie `Error`. Oto niektóre z najczęściej spotykanych typów błędów:

- `Error`: Podstawowy typ błędu. Może być używany jako ogólny typ błędu.

- `SyntaxError`: Występuje, gdy w kodzie występuje błąd składniowy. Na przykład,

nieprawidłowe użycie składni języka JavaScript.

- `ReferenceError`: Występuje, gdy odwołujemy się do nieistniejącej zmiennej lub obiektu.



### Error (cont)

> Na przykład, próba odwołania się do niezadeklarowanej zmiennej.

- **TypeError**: Występuje, gdy wykonywane jest nieprawidłowe działanie na typie danych.

Na przykład, próba wykonania działania matematycznego na wartości niebędącej liczbą.

- **RangeError**: Występuje, gdy wartość znajduje się poza zakresem, który jest dozwolony.

Na przykład, próba utworzenia tablicy o zbyt dużym rozmiarze.

- **EvalError**: Występuje, gdy wystąpi błąd w funkcji eval().

- **URIError**: Występuje, gdy URI (Uniform Resource Identifier) nie jest poprawnie zakodowany.

- **AggregateError**: Występuje w przypadku wielu błędów zgromadzonych w jednym

obiekcie. Jest używany w przypadku obsługi wielu promisów.

Warto zauważyć, że możemy również tworzyć własne typy błędów, dziedzicząc po

klasie Error. Tworzenie niestandardowych typów błędów może być przydatne w celu

lepszego zorganizowania obsługi błędów w naszym kodzie.

Przykład użycia niestandardowego typu błędu:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = 'CustomError';
  }
}
try {
  throw new CustomError('To jest niestandardowy błąd');
} catch (error) {
  console.log(error instanceof CustomError); // true
  console.log(error.message); // 'To jest niestandardowy błąd'
}
```

W powyższym przykładzie tworzymy niestandardowy typ błędu o nazwie

CustomError, który dziedziczy po klasie Error. Możemy go obsługiwać podobnie

### Error (cont)

> jak inne wbudowane typy błędów.

### XMLHttpRequest

XMLHttpRequest jest obiektem wbudowanym w przeglądarki internetowe, które umożliwia komunikację z serwerem za pomocą protokołu HTTP lub HTTPS.

Służy do wysyłania asynchronicznych żądań HTTP i odbierania odpowiedzi w formacie tekstowym, XML lub JSON.

Główne kroki w użyciu XMLHttpRequest obejmują:

Tworzenie obiektu XMLHttpRequest:

```
const xhr = new XMLHttpRequest();
```

Konfiguracja żądania HTTP:

```
xhr.open(method, url, async);
```

```
method: Metoda HTTP, np. "GET", "POST", "PUT", "DELETE".
```

```
url: Adres URL, do którego wysyłane jest żądanie.
```

```
async (opcjonalne): Określa, czy żądanie ma być asynchroniczne (domyślnie true) lub synchroniczne (false).
```

Ustalenie funkcji obsługi zdarzenia readyStateChange:

```
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4 && xhr.status === 200) {
    // Obsługa zakończonego żądania
  }
};
```

readyState reprezentuje stan obiektu XMLHttpRequest i może przyjmować różne

wartości od 0 do 4. Wartość 4 oznacza, że żądanie zostało zakończone i odpowiedź

jest gotowa do odczytu.

Wysłanie żądania:

```
xhr.send(data);
```

data (opcjonalne) reprezentuje dane, które mogą być przesłane wraz z żądaniem,

np. w przypadku metod POST lub PUT.





### XMLHttpRequest (cont)

> Obsługa odpowiedzi:

```
xhr.responseText; // Odpowiedź jako tekst
xhr.responseXML; // Odpowiedź jako dokument XML
xhr.status; // Kod stanu odpowiedzi HTTP
```

Odpowiedź z serwera jest dostępna za pomocą właściwości `responseText` lub `responseXML`. Właściwość `status` zawiera kod stanu odpowiedzi HTTP, np. 200 oznacza sukces.

XMLHttpRequest dostarcza również inne funkcje i metody, takie jak `setRequestHeader()`, która umożliwia ustawianie nagłówków żądania HTTP, oraz `abort()`, która służy do anulowania żądania w trakcie trwania.

Warto również zaznaczyć, że wraz z pojawieniem się nowszych standardów w JavaScript, takich jak Fetch API i XMLHttpRequest zostało zastąpione przez te bardziej nowoczesne metody obsługi żądań HTTP.

### Iteracja po elementach kolekcji

W języku JavaScript istnieje kilka sposobów iteracji po elementach kolekcji, takich jak tablice lub obiekty. Oto kilka popularnych sposobów iteracji:

- **Pętla for:** Tradycyjna pętla `for` pozwala na iterację po indeksach kolekcji. Jest szczególnie przydatna do iteracji po tablicach.

Przykład:

```
const tablica = [1, 2, 3, 4];
for (let i = 0; i < tablica.length; i++) {
  console.log(tablica[i]);
}
```

- **Pętla for...of:** Pętla `for...of` umożliwia iterację po elementach kolekcji, takich jak tablice lub obiekty iterowalne, bez konieczności odwoływania się do indeksów.

Przykład:

```
const tablica = [1, 2, 3, 4];
for (let element of tablica) {
  console.log(element);
}
```

### Iteracja po elementach kolekcji (cont)

```
> }
```

- **Metoda forEach:** Metoda `forEach` jest dostępna dla tablic i pozwala na wykonanie określonej funkcji dla każdego elementu tablicy.

Przykład:

```
const tablica = [1, 2, 3, 4];
tablica.forEach((element) => {
  console.log(element);
});
```

- **Metoda map:** Metoda `map` również jest dostępna dla tablic i pozwala na wykonanie określonej funkcji dla każdego elementu tablicy, a następnie zwrócenie nowej tablicy zawierającej wyniki.

Przykład:

```
const tablica = [1, 2, 3, 4];
const nowaTablica = tablica.map((element) => {
  return element * 2;
});
console.log(nowaTablica);
```

- **Pętla for...in:** Pętla `for...in` umożliwia iterację po nazwach właściwości obiektu.

Jest szczególnie przydatna do iteracji po obiektach.

Przykład:

```
const obiekt = { a: 1, b: 2, c: 3 };
for (let key in obiekt) {
  console.log(key, obiekt[key]);
}
```

### Klasa

Klasy są wprowadzone w języku JavaScript w standardzie ECMAScript 2015 (ES6) jako syntaktyczny cukier nad prototypami, umożliwiający definiowanie obiektów i ich właściwości oraz metody w bardziej deklaracyjny sposób. Klasy pozwalają na tworzenie obiektów o podobnej strukturze i zachowaniu. Oto kilka kluczowych cech klas w JavaScript: Deklaracja klasy: Klasa może być zadeklarowana za pomocą słowa kluczowego `class`, po którym podajemy nazwę klasy.

Przykład:

```
class Klasa {
  // Ciało klasy
}
```

Konstruktor: Konstruktor jest specjalną metodą w klasie, która jest wywoływana podczas tworzenia nowego obiektu na podstawie klasy. Może być używany do inicjalizacji właściwości obiektu.

Przykład:

```
class Klasa {
  constructor(parametr) {
    this.wlasnosc = parametr;
  }
}
```

```
const obiekt = new Klasa('właśc');
```

Metody: Metody to funkcje zdefiniowane wewnątrz klasy, które wykonują określone operacje na obiekcie. Mogą być używane do manipulacji właściwościami obiektu lub wykonywania innych operacji.

Przykład:

```
class Klasa {
  metoda() {
    console.log('Wywołanie metody');
  }
}
```

### Klasa (cont)

```
> }
```

```
}
```

```
const obiekt = new Klasa();
```

```
obiekt.metoda();
```

Dziedziczenie: Klasa może dziedziczyć właściwości i metody z innej klasy za pomocą

słowa kluczowego `extends`. Dziedziczenie pozwala na tworzenie hierarchii klas i ponowne

wykorzystanie kodu.

Przykład:

```
class KlasaPodstawowa {
  metoda() {
    console.log("Metoda z klasy podstawowej");
  }
}
```

```
class KlasaPochodna extends KlasaPodstawowa {
  // Klasa pochodna dziedziczy metody z klasy podstawowej
}
```

```
const obiekt = new KlasaPochodna();
```

```
obiekt.metoda(); // Wywołanie metody z klasy podstawowej
```

Słowo kluczowe `super` w języku JavaScript jest używane w kontekście dziedziczenia klas,

aby odwoływać się do właściwości i metod z klasy nadrzędnej (klasy bazowej). Pozwala to

na rozszerzanie funkcjonalności klasy nadrzędnej w klasie pochodnej oraz wywoływanie

konstruktora klasy nadrzędnej.

Oto kilka zastosowań słowa kluczowego `super`:

Wywołanie konstruktora klasy nadrzędnej: Przy dziedziczeniu klas, konstruktor klasy

pochodnej może wywołać konstruktor klasy nadrzędnej za pomocą `super()`. Pozwala to

na inicjalizację właściwości klasy nadrzędnej przed zainicjalizowaniem właściwości klasy

pochodnej.

Przykład:



### Klasa (cont)

```
> class KlasaNadrzedna {
  constructor(parametr) {
    this.wlasowoscNadrzedna = parametr;
  }
}
class KlasaPochodna extends KlasaNadrzedna {
  constructor(parametr1, parametr2) {
    super(parametr1); // Wywołanie konstruktora klasy nadrzędnej
    this.wlasowoscPochodna = parametr2;
  }
}
const obiekt = new KlasaPochodna('wartość1', 'wartość2');
console.log(obiekt.wlasowoscNadrzedna); // wartość1
console.log(obiekt.wlasowoscPochodna); // wartość2
```

Odwoływanie się do metody klasy nadrzędnej: Słowo kluczowe `super` może być używane do odwoływania się do metody z klasy nadrzędnej w klasie pochodnej. Umożliwia to rozszerzanie funkcjonalności metody z klasy nadrzędnej lub nadpisywanie jej zachowania.

Przykład:

```
class KlasaNadrzedna {
  metoda() {
    console.log('Metoda z klasy nadrzędnej');
  }
}
class KlasaPochodna extends KlasaNadrzedna {
  metoda() {
    super.metoda(); // Wywołanie metody klasy nadrzędnej
    console.log('Metoda z klasy pochodnej');
  }
}
```

### Klasa (cont)

```
> }
}
const obiekt = new KlasaPochodna();
obiekt.metoda();
// Wyjście:
// Metoda z klasy nadrzędnej
// Metoda z klasy pochodnej
```

Słowo kluczowe `super` jest używane do odwoływania się do elementów klasy nadrzędnej w kontekście dziedziczenia. Dzięki temu możliwe jest rozszerzanie i dostęp do funkcjonalności klasy nadrzędnej w klasie pochodnej. Klasy w JavaScript zapewniają bardziej zwięzłą i deklaratywną składnię do tworzenia obiektów i zarządzania ich zachowaniem. Mogą być używane do implementacji obiektowo zorientowanego programowania w JavaScript.

```
let Animal = class BasicAnimal {
  constructor(name) {
    this.name = name
    this._age = 1;

    if (Animal.count === undefined) Animal.count = 0;
    Animal.count++; // emulujemy statyczne property w ES6
  }
  getName = () => {
    return this.name;
  }
  set age(value) {
    if (value > 0) this._age = value;
  }
  get age() {
    return this._age;
  }
}
```



### Klasa (cont)

```
> }
  static getNewAnimal() {
    return new Animal("Default animal");
  }
  static getAnimalCount() {
    return Animal.count;
  }
}
const animal1 = new Animal("Tiger");
console.log( animal1.getName() ); // "Tiger"
console.log( Animal.name ); // BasicAnimal
animal1.age = 10;
// animal1._age = 22; // można zmodyfikować, nie ma private w
ES6
console.log( animal1.age ); // 10
const animal2 = Animal.getNewAnimal();
console.log(animal2.getName()); // Default animal
console.dir(Animal);
console.log( Animal.getAnimalCount() ); // 2
```

### Closures

Closures (domknięcia) są ważnym konceptem w języku JavaScript. Oznaczają one zdolność funkcji do zapamiętania i uzyskiwania dostępu do zmiennych ze swojego otoczenia, nawet po opuszczeniu tego otoczenia. Oto prosty przykład, który demonstrowa closures:

```
function outer() {
  var outerVariable = 'Hello';
  function inner() {
    var innerVariable = 'World';
    console.log( outerVariable + ' ' +
innerVariable );
```

### Closures (cont)

```
> }
  return inner;
}
var closureFn = outer();
closureFn(); // Wynik: "Hello World"
```

W tym przykładzie funkcja inner jest zagnieżdżona w funkcji outer. Funkcja inner ma dostęp do zmiennej outerVariable z funkcji outer, nawet po tym jak funkcja outer zakończyła swoje działanie i została wywołana. Closures są przydatne w wielu sytuacjach, takich jak tworzenie prywatnych zmiennych, implementacja funkcji zwrrotnych (callback) i zarządzanie stanem w programowaniu asynchronicznym. Pozwalają one na utrzymanie stanu i zapamiętywanie danych między różnymi wywołaniami funkcji. Warto zauważyć, że closures w języku JavaScript mogą mieć znaczący wpływ na zarządzanie pamięcią. Jeśli closure nadal utrzymuje referencję do obiektów lub zmiennych, które nie są już potrzebne, może to prowadzić do wycieków pamięci.

### Currying

Currying to technika programistyczna, która polega na przekształceniu funkcji o wielu argumentach w sekwencję funkcji o pojedynczych argumentach. Pozwala to na tworzenie bardziej elastycznych i modułowych funkcji. W curryingu funkcja przyjmuje jeden argument i zwraca nową funkcję, która również przyjmuje jeden argument, a tak dalej, aż wszystkie argumenty zostaną dostarczone i funkcja zostanie ostatecznie wywołana. Oto prosty przykład:

```
function add(a) {
  return function(b) {
    return a + b;
  }
}
```



### Currying (cont)

```
> var add5 = add(5); // Zwraca funkcję, która dodaje 5 do przekazanego argumentu
console.log(add5(2)); // Wynik: 7
console.log(add5(10)); // Wynik: 15
```

W powyższym przykładzie funkcja `add` jest funkcją currying. Przyjmuje ona argument `a` i zwraca funkcję, która dodaje przekazany argument `b` do `a`. Wywołanie `add(5)` zwraca funkcję, która dodaje 5 do przekazanego argumentu. Następnie możemy wielokrotnie wywoływać zwróconą funkcję, przekazując różne argumenty. Currying jest przydatne w wielu sytuacjach, szczególnie przy tworzeniu funkcji cząstkowych (partial functions) i kompozycji funkcji. Umożliwia tworzenie bardziej elastycznych funkcji, które można łatwo dostosować do różnych przypadków użycia. Warto również wspomnieć, że w języku JavaScript można używać metod takich jak `bind()` do osiągnięcia efektu podobnego do currying. Metoda `bind()` pozwala na utworzenie nowej funkcji, która jest związana z określonym kontekstem i zestawem argumentów.

### ECMAScript 2021 (ES12)

- Private Fields and Methods (Pola i metody prywatne):

ES12 wprowadza możliwość definiowania pól i metod prywatnych w klasach za pomocą prefixu `#`. Przykład:

```
class MyClass {
  #privateField = 10;
  #privateMethod() {
    return this.#privateField;
  }
  get PrivateFieldValue() {
    return this.#privateField;
  }
}
```

```
const myObject = new MyClass();
console.log(myObject.getPrivateFieldValue()); // 10
```

### ECMAScript 2021 (ES12) (cont)

```
> console.log(myObject.privateField); // undefined
console.log(myObject.privateMethod()); // Error: privateMethod is not a function
```

- Logical Assignment Operators (Operatorzy logiczne przypisania):

ES12 wprowadza skrócone zapisy operacji logicznych z przypisaniem, takie jak  `||=` ,  `&&=` ,  `??=` . Pozwalają one na wykonanie operacji logicznej i przypisania wartości w jednym kroku. Przykład:

```
let x = 5;
x ||= 10; // Jeśli x jest fałszywe (falsy), przypisz mu wartość 10
console.log(x); // 5
let y = null;
y ??= 'Hello'; // Jeśli y jest null lub undefined, przypisz mu wartość 'Hello'
console.log(y); // 'Hello'
```

- Numeric Separator (Separator numeryczny):

ES12 wprowadza możliwość stosowania podkreślenia `_` jako separatora w liczbach, aby ułatwić czytanie i zrozumienie dużych liczb. Przykład:

```
const bigNumber = 1_000_000;
console.log(bigNumber); // 1000000
```

- Promise.any():

ES12 wprowadza funkcję `Promise.any()`, która przyjmuje tablicę obiektów `Promise` i zwraca pierwszy rozwiązany `Promise`. Przykład:

```
const promises = [
  fetch('https://api.example.com/data1'),
  fetch('https://api.example.com/data2'),
  fetch('https://api.example.com/data3')
];
Promise.any(promises)
  .then(response => console.log(response))
  .catch(error => console.error(error));
```

### ECMAScript 2021 (ES12) (cont)

> - WeakRef to nowa funkcjonalność wprowadzona w ECMAScript 2021 (ES12).

Służy do tworzenia słabych referencji do obiektów, co umożliwia bardziej elastyczne

zarządzanie pamięcią w JavaScript.

Słabe referencje są szczególnie przydatne w sytuacjach, gdy chcemy monitorować

obiekty, ale jednocześnie pozwolić na ich automatyczne usuwanie przez mechanizm

zarządzania pamięcią, jeśli nie są już używane. Słabe referencje nie zatrzymują

automatycznie obiektów przed usunięciem przez garbage collector.

Oto prosty przykład użycia WeakRef:

```
let obj = { data: "Hello" };
```

```
const ref = new WeakRef(obj);
```

```
// Sprawdzanie, czy obiekt wciąż istnieje
```

```
console.log(ref.deref()); // { data: "Hello" }
```

```
obj = null; // Usunięcie referencji do obiektu
```

```
// Sprawdzanie, czy obiekt został usunięty
```

```
console.log(ref.deref()); // null
```

W powyższym przykładzie tworzymy obiekt obj, a następnie

tworzymy słabą

referencję ref do tego obiektu za pomocą konstruktora WeakRef.

Możemy użyć

metody deref() na obiekcie ref, aby sprawdzić, czy obiekt wciąż istnieje.

Po usunięciu referencji do obiektu poprzez przypisanie null do zmiennej obj,

ponowne wywołanie deref() zwraca null, co oznacza, że obiekt został usunięty.

Słabe referencje są szczególnie przydatne w przypadku tworzenia pamięciochłonnych

obektów, które mogą być dynamicznie usuwane, gdy nie są już potrzebne.

### ECMAScript 2019 (ES10) (cont)

```
> const nestedArray = [1, [2, 3], [4, [5, 6]]];
```

```
const flattenedArray = nestedArray.flat();
```

```
console.log(flattenedArray); // [1, 2, 3, 4, [5, 6]]
```

```
const numbers = [1, 2, 3];
```

```
const doubledArray = numbers.flatMap(num => [num, num * 2]);
```

```
console.log(doubledArray); // [1, 2, 2, 4, 3, 6]
```

- String.trimStart() i String.trimEnd(): Metody trimStart() i trimEnd()

służą do usuwania

spacji na początku i końcu łańcucha znaków. Zastępują one metody trimLeft() i trimRight()

używane wcześniej. Przykład:

```
const text = ' Hello World ';
```

```
console.log(text.trimStart()); // 'Hello World '
```

```
console.log(text.trimEnd()); // ' Hello World'
```

- Optional Catch Binding: Pozwala na użycie pustego identyfikatora w bloku catch,

jeśli nie jest potrzebny dostęp do błędu. Przykład:

```
try {
```

```
  // kod generujący błąd
```

```
} catch {
```

```
  console.log("Wystąpił błąd");
```

```
}
```

- Object.fromEntries(): Metoda fromEntries() przekształca listę par klucz-wartość

na obiekt. Przykład:

```
const entries = [['a', 1], ['b', 2], ['c', 3]];
```

```
const obj = Object.fromEntries(entries);
```

```
console.log(obj); // { a: 1, b: 2, c: 3 }
```

- Symbol Description Accessor: Umożliwia uzyskanie dostępu do opisu symbolu za

pomocą metody description. Przykład:

```
const symbol = Symbol('My Symbol');
```

```
console.log(symbol.description); // 'My Symbol'
```

### ECMAScript 2019 (ES10)

Oto lista niektórych nowych funkcji i zasad wprowadzonych w ECMAScript 2019 (ES10):

- Array.flat() i Array.flatMap(): Metoda flat()

służy do spłaszczenia tablicy, usuwając

zagnieżdżone tablice i łącząc je w jedną płaską

tablicę. Metoda flatMap() pozwala na

jednocześnie spłaszczenie tablicy i zastosowanie

funkcji do każdego elementu.

Przykład:



### ECMAScript 2017 (ES8)

Oto kilka nowości wprowadzonych w ECMAScript 2017 (ES8) wraz z przykładami:

- Async Functions: Wprowadzono asynchroniczne funkcje, które ułatwiają pracę z asynchronicznym kodem. Wykorzystuje się do tego słowo kluczowe `async` oraz `await`.

Przykład:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error:', error);
  }
}
```

- `Object.values()`: Metoda `Object.values()` zwraca tablicę zawierającą wartości wszystkich właściwości obiektu. Przykład:

```
const obj = { a: 1, b: 2, c: 3 };
const values = Object.values(obj);
console.log(values); // [1, 2, 3]
```

- `Object.entries()`: Metoda `Object.entries()` zwraca tablicę zawierającą tablice składające się z klucza i wartości każdej właściwości obiektu. Przykład:

```
const obj = { a: 1, b: 2, c: 3 };
const entries = Object.entries(obj);
console.log(entries); // [["a", 1], ["b", 2], ["c", 3]]
```

- String padding: Dodano możliwość wyrównywania łańcucha znaków poprzez

uzupełnienie go odpowiednią ilością znaków.

Wykorzystuje się do tego metody

`padStart()` i `padEnd()`. Przykład:

```
const str = 'Hello';
console.log(str.padStart(10, ' '); // "Hello"
```

### ECMAScript 2017 (ES8) (cont)

```
> console.log(str.padEnd(10, '-')); // "Hello-----"
```

- `Object.getOwnPropertyDescriptors()`: Metoda `Object.getOwnPropertyDescriptors()`

zwraca wszystkie deskryptory właściwości obiektu. Przykład:

```
const obj = {
  name: 'John',
  age: 30
};
const descriptors = Object.getOwnPropertyDescriptors(obj);
console.log(descriptors);
/*
{
  name: { value: 'John', writable: true, enumerable: true, configurable: true },
  age: { value: 30, writable: true, enumerable: true, configurable: true }
}
*/
```

- `SharedArrayBuffer` i `Atomics`: Wprowadzono nowe obiekty `SharedArrayBuffer` i `Atomics`

do obsługi współdzielonych buforów pomiędzy wątkami w środowisku wielowątkowym.

- Async Iteration: Wprowadzono możliwość iteracji asynchronicznej przy użyciu pętli

`for-await-of`, co ułatwia pracę z asynchronicznymi operacjami.

Przykład:

```
async function* asyncGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
(async () => {
  for await (const num of asyncGenerator()) {
    console.log(num);
  }
})
```



### ECMAScript 2017 (ES8) (cont)

```
> });
```

### ECMAScript 2016 (ES7)

ES7, również znany jako ECMAScript 2016, wprowadził niewiele nowych funkcji w porównaniu do poprzednich wersji języka JavaScript. Poniżej znajduje się lista kilku zmian wprowadzonych w ES7 wraz z przykładami:

- Includes:

Metoda `includes` została dodana do prototypu tablic w celu sprawdzenia, czy dany element istnieje w tablicy. Zwraca wartość logiczną `true`, jeśli element istnieje, lub `false`, jeśli nie istnieje. Przykład:

```
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.includes(3)); // true
console.log(numbers.includes(6)); // false
```

- Potęgowanie:

Operator potęgowania `**` został wprowadzony w ES7 do obliczania potęgi liczby.

Przykład:

```
console.log(2 ** 3); // 8
console.log(5 ** 2); // 25
```

- Rest Properties:

W ES7 dodano obsługę reszty właściwości w destrukcji rzyzacji obiektów. Umożliwia to zbieranie pozostałych właściwości obiektu do jednej zmiennej. Przykład:

```
const { name, age, ...rest } = { name: 'John',
age: 30, gender: 'male', city: 'New York' };
console.log(name); // 'John'
console.log(age); // 30
console.log(rest); // { gender: 'male', city: 'New York' }
```

- Async Functions:

W ES7 wprowadzono słowo kluczowe `async`, które umożliwia tworzenie funkcji asynchronicznych. Funkcje asynchroniczne obsługują `Promise` i umożliwiają korzystanie z `await` wewnątrz funkcji, co ułatwia programowanie asynchroniczne. Przykład:

### ECMAScript 2016 (ES7) (cont)

```
> async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
}
```

- Shared Memory i Atomics (Atomics API):

ES7 wprowadza obsługę pamięci współdzielonej i wprowadza nowe API `Atomics` do

wykonywania operacji atomowych na pamięci współdzielonej. Ta funkcjonalność jest

szczególnie przydatna w programowaniu równoległym i wielowątkowym, ale wymaga

bardziej zaawansowanych technik programistycznych.

Warto zauważyć, że ES7 nie wprowadził dużej liczby nowych funkcji, ale skupił się głównie

na poprawie i rozszerzeniu istniejących funkcjonalności języka JavaScript.

### Instrukcje warunkowe: if else if

```
if (temperatura < 0) console.log("Bierzemy kurtkę");
else if (temperatura == 0) console.log("Bierzemy rękaw");
else if (temperatura >= 0) console.log("Bierzemy rękaw");
else console.log("Zostajemy w domu");
```

### switch case

```
switch (data3) {
  case 0:
    console.log("data równe 0");
    break;
  case 1:
    console.log("data równe 1");
    break;
  case 3:
    console.log("data równe 3");
}
```



### switch case (cont)

```
> break
case 5:
  console.log("data równe 5")
  break
default:
  console.log("data równe jest dokładnie:", data3)
}
```

### while

```
while ( i < 3) {console.log("while oraz wartość i: " + i)
                i = i + 1 }
```

### do while

```
do {console.log("do while oraz wartość b:", b)
    b++ // to samo co b = b + 1
} while (b < 4)
```

### for

```
for (let c = 0; c < 5; c++) {
  console.log(" pętla c, wartość c:", c)}
```

### Number - bezpieczne zakresy

Number.MAX_SAFE_INTEGER	900719 925 474099
Number.MIN_SAFE_INTEGER	-90071 992 547 40991
Number.isSafeInteger( Math.pow( 2,53) - 1)	true
Number.isSafeInteger( Math.pow( 2,53) )	false

### Infinity

console.log(1 / 0)	Infinity
console.log(-1 / 0)	-Infinity
Number.isFinite(35)	true
Number.isFinite(1/0)	false

### Numbers - właściwości i funkcje

```
console.log(Number.MAX_VALUE); //
1.7976931348623157e+308
console.log(Number.MIN_VALUE); // 5e-324
Funkcja Number.parseInt(): Konwertuje ciąg
znaków na liczbę całkowitą.
console.log(Number.parseInt("42")); // 42
console.log(Number.parseInt("3.14")); // 3
console.log(Number.parseInt("abc")); // NaN
Funkcja Number.parseFloat(): Konwertuje ciąg
znaków na liczbę zmiennoprzecinkową.
console.log(Number.parseFloat("3.14"));
// 3.14
console.log(Number.parseFloat("42.0"));
// 42
console.log(Number.parseFloat("abc")); //
NaN
Funkcja Number.isInteger(): Sprawdza, czy
wartość jest liczbą całkowitą.
console.log(Number.isInteger(42)); // true
console.log(Number.isInteger(3.14)); //
false
console.log(Number.isInteger("42")); //
false
console.log(Number.isInteger(NaN)); //
false
Metoda Number.toFixed(): Określa liczbę miejsc
dziesiętnych po przecinku dla liczby
zmiennoprzecinkowej i zwraca wynik jako ciąg
znaków.
let num = 3.14159;
console.log(num.toFixed(2)); // "3.14"
console.log(num.toFixed(0)); // "3"
console.log(num.toFixed(5)); // "3.14159"
Uwaga: Number.toFixed() zwraca wynik jako string,
a nie jako liczbę.
Jeśli chcesz uzyskać liczbę z ustaloną ilością
miejsc dziesiętnych, można
```

### Numbers - właściwości i funkcje (cont)

```
> użyć parseFloat() lub Number().
let num = 3.14159;
console.log(parseFloat(num.toFixed(2))); // 3.14
console.log(Number(num.toFixed(2))); // 3.14
```

### Zaokrąglenie liczb

Metoda `Math.round()`: Zaokrągla liczbę do najbliższej wartości całkowitej.

```
console.log(Math.round(3.7)); // 4
console.log(Math.round(3.2)); // 3
```

Metoda `Math.floor()`: Zaokrągla liczbę w dół do najbliższej wartości całkowitej mniejszej lub równej danej liczbie.

```
console.log(Math.floor(3.7)); // 3
console.log(Math.floor(3.2)); // 3
```

Metoda `Math.ceil()`: Zaokrągla liczbę w górę do najbliższej wartości całkowitej większej lub równej danej liczbie.

```
console.log(Math.ceil(3.7)); // 4
console.log(Math.ceil(3.2)); // 4
```

Metoda `Math.trunc()`: Zwraca tylko całkowitą część liczby, odrzucając jej część dziesiętną.

```
console.log(Math.trunc(3.7)); // 3
console.log(Math.trunc(3.2)); // 3
```

Metoda `Number.toFixed()`: Zaokrągla liczbę do określonej liczby miejsc dziesiętnych i zwraca wynik jako ciąg znaków.

```
console.log((3.14159).toFixed(2)); // "3.14"
console.log((3.14159).toFixed(0)); // "3"
```

Metody `Math.floor()`, `Math.ceil()`, `Math.trunc()` można także wykorzystać w połączeniu z mnożeniem i dzieleniem, aby zaokrąglić liczbę do określonej dokładności miejsc dziesiętnych.

```
let num = 3.14159;
```

### Zaokrąglenie liczb (cont)

```
> let roundedNum = Math.floor(num * 100) / 100;
console.log(roundedNum); // 3.14
```

### Operatory przypisania

`+=` - Dodawanie i przypisanie  
Przykład: `x += 2;` // równoważne zapisowi: `x = x + 2;`

`-=` - Odejmowanie i przypisanie  
Przykład: `x -= 3;` // równoważne zapisowi: `x = x - 3;`

`*=` - Mnożenie i przypisanie  
Przykład: `x = 4;` // równoważne zapisowi: `x = x * 4;`

`/=` - Dzielenie i przypisanie  
Przykład: `x /= 2;` // równoważne zapisowi: `x = x / 2;`

`%=` - Reszta z dzielenia i przypisanie  
Przykład: `x %= 3;` // równoważne zapisowi: `x = x % 3;`

`**=` - Potęgowanie i przypisanie  
Przykład: `x = 2;` // równoważne zapisowi: `x = x ** 2;`

`<<=` - Przesunięcie bitowe w lewo i przypisanie  
Przykład: `x <<= 3;` // równoważne zapisowi: `x = x << 3;`

`>>=` - Przesunięcie bitowe w prawo (ze znakiem) i przypisanie  
Przykład: `x >>= 2;` // równoważne zapisowi: `x = x >> 2;`

`>>>=` - Przesunięcie bitowe w prawo (bez znaku) i przypisanie  
Przykład: `x >>>= 4;` // równoważne zapisowi: `x = x >>> 4;`

`&=` - Bitowe AND i przypisanie  
Przykład: `x &= 5;` // równoważne zapisowi: `x = x & 5;`

`|=` - Bitowe OR i przypisanie  
Przykład: `x |= 8;` // równoważne zapisowi: `x = x | 8;`

`^=` - Bitowe XOR i przypisanie  
Przykład: `x ^= 3;` // równoważne zapisowi: `x = x ^ 3;`



### Operatory arytmetyczne

+ - Dodawanie  
 Przykład: 2 + 3 // wynik: 5

- - Odejmowanie  
 Przykład: 5 - 2 // wynik: 3

\* - Mnożenie  
 Przykład: 2 \* 3 // wynik: 6

/ - Dzielenie  
 Przykład: 6 / 2 // wynik: 3

% - Reszta z dzielenia (modulo)  
 Przykład: 7 % 3 // wynik: 1

\*\* - Potęgowanie  
 Przykład: 2 \*\* 3 // wynik: 8

++ - Inkrementacja (zwiększenie o 1)  
 Przykład: let x = 5; x++; // x będzie miało wartość 6

-- - Dekrementacja (zmniejszenie o 1)  
 Przykład: let x = 5; x--; // x będzie miało wartość 4

### Operatory porównania (relacyjne)

== - Równy wartości (konwertuje typy)  
 Przykład: 2 == '2' // wynik: true

=== - Równy wartości i typu (nie konwertuje typów)  
 Przykład: 2 === '2' // wynik: false

!= - Nie równy wartości (konwertuje typy)  
 Przykład: 2 != '3' // wynik: true

!== - Nie równy wartości i typu (nie konwertuje typów)  
 Przykład: 2 !== '3' // wynik: true

> - Większy niż  
 Przykład: 5 > 3 // wynik: true

< - Mniejszy niż

### Operatory porównania (relacyjne) (cont)

> Przykład: 3 < 5 // wynik: true

>= - Większy lub równy  
 Przykład: 5 >= 5 // wynik: true

<= - Mniejszy lub równy  
 Przykład: 3 <= 5 // wynik: true

Operatory porównania z obiektami (== i ===) porównują referencje obiektów, a nie ich wartości. Dlatego dla dwóch różnych obiektów o identycznym zewnętrznym wyglądzie, operator porównania zwróci false, chyba że obiekt jest bezpośrednio porównywany z samym sobą.

```
let obj1 = { value: 5 };
let obj2 = { value: 5 };
let obj3 = obj1;
console.log(obj1 == obj2); // false
console.log(obj1 === obj2); // false
console.log(obj1 === obj3); // true
```

### Operatory logiczne

&& - Operator logiczny "i" (AND)  
 Zwraca true, jeśli oba wyrażenia są prawdziwe, w przeciwnym razie zwraca false.  
 Przykład: true && false // wynik: false

|| - Operator logiczny "lub" (OR)  
 Zwraca true, jeśli przynajmniej jedno z wyrażeń jest prawdziwe, w przeciwnym razie zwraca false.  
 Przykład: true || false // wynik: true

! - Operator logiczny "nie" (NOT)  
 Zwraca wartość przeciwną do wartości wyrażenia, czyli true zamienia na false i vice versa.  
 Przykład: !true // wynik: false

Operatory logiczne && i || wykorzystują "short-circuit evaluation" (ocenie



### Operatory logiczne (cont)

> krótkiego obwodu). Oznacza to, że jeśli wynik można określić na podstawie

pierwszego wyrażenia, drugie wyrażenie nie jest oceniane.

```
let a = 5;
let b = 10;
if (a > 0 && b > 0) {
  console.log("Oba warunki są spełnione");
} else {
  console.log("Co najmniej jeden warunek nie jest spełniony");
}
```

W tym przypadku, jeśli a i b są większe od zera, to oba warunki są spełnione i

zostanie wyświetlony odpowiedni komunikat. Jeśli przynajmniej jeden z warunków

nie jest spełniony, zostanie wyświetlony inny komunikat.

### Operatory bitowe

& - Operator bitowy AND

Wykonuje operację logicznego AND bit po bicie na dwóch operandach.

Przykład: `5 & 3 // wynik: 1 (binarnie: 0101 & 0011 = 0001)`

| - Operator bitowy OR

Wykonuje operację logicznego OR bit po bicie na dwóch operandach.

Przykład: `5 | 3 // wynik: 7 (binarnie: 0101 | 0011 = 0111)`

^ - Operator bitowy XOR (exclusive OR)

Wykonuje operację logicznego XOR bit po bicie na dwóch operandach.

Przykład: `5 ^ 3 // wynik: 6 (binarnie: 0101 ^ 0011 = 0110)`

~ - Operator bitowy NOT (negacja)

Wykonuje negację bitową na jednym operandzie.

Odwraca bity.

Przykład: `~5 // wynik: -6 (binarnie: ~0101 = 1010)`

<< - Operator bitowy przesunięcia w lewo

Przesuwa bity w lewo o określoną ilość pozycji.

Przykład: `5 << 2 // wynik: 20 (binarnie: 0101 << 2 = 10100)`

### Operatory bitowe (cont)

>>> - Operator bitowy przesunięcia w prawo ze znakiem

Przesuwa bity w prawo o określoną ilość pozycji, zachowując znak liczby.

Przykład: `5 >> 1 // wynik: 2 (binarnie: 0101 >> 1 = 0010)`

>>> - Operator bitowy przesunięcia w prawo bez znaku

Przesuwa bity w prawo o określoną ilość pozycji, wypełniając zerami z lewej strony.

Przykład: `5 >>> 1 // wynik: 2 (binarnie: 0101 >>> 1 = 0010)`

### Inne operatory

Operator warunkowy (ternarny, trójargumentowy, trójelementowy):

Przykład: `let result = condition ? value1 : value2;`

Operator konkatencji (dla łączenia stringów):

Przykład: `let fullName = "John" + " " + "Doe";`

Operator dostępu do właściwości:

Przykład: `let length = array.length;`

Operator przecinka ,:

Operator przecinka w JavaScript służy do oddzielenia wielu wyrażeń i wykonywania

ich sekwencyjnie. Wykonuje się wyrażenie po lewej stronie przecinka, a następnie

wyrażenie po prawej stronie przecinka. Operator przecinka zwraca wartość

ostatniego wyrażenia.

`let a = 1, b = 2, c = 3;`

`console.log(a, b, c); // wynik: 1 2 3`

Operator jednoargumentowy (unary):

Operator jednoargumentowy działa na jednym

argumentach. Przykładowymi operatorami

jednoargumentowymi w JavaScript są:

+ - Konwersja na liczbę dodatnią

- - Negacja liczby

++ - Inkrementacja

-- - Dekrementacja

! - Negacja logiczna (NOT)



### Inne operatory (cont)

```
> let a = 5;
console.log(-a); // wynik: -5
console.log(++a); // wynik: 6
console.log(!true); // wynik: false
```

### Pierwszeństwo operatorów

W języku JavaScript istnieje hierarchia lub priorytet operatorów, która określa kolejność wykonywania operacji. Poniżej przedstawiam przykładową listę operatorów według priorytetu, od najwyższego do najniższego:

Nawiasy: ()

Operator indeksowania: []

Operator wywołania funkcji: ()

Postinkrementacja (++) i postdekrementacja (--)

Preinkrementacja (++) i predekrementacja (--)

Unarne operatory arytmetyczne: +, -

Unarne operatory logiczne: !, typeof, void, delete

Potęgowanie: \*\*

Mnożenie: \*, dzielenie: /, reszta z dzielenia: %

Dodawanie: +, odejmowanie: -

Operatory porównania: >, <, >=, <=, instanceof, in

Operatory równości: ==, !=, ===, !==

Operatory logiczne: &&, ||

Operator warunkowy (ternary): ? :

Operator przypisania: =, +=, -=, \*=, /=, %=, \*=,

<<=, >>=, >>>=, &=, |=, ^=

W przypadku wyrażeń z operatorem o tym samym poziomie priorytetu, wykonywane są od lewej do prawej.

### Konwersja typów

W języku JavaScript istnieją dwa główne rodzaje konwersji typów: niejawna (automatyczna) konwersja typów i jawna (ręczna) konwersja typów.

Niejawna (automatyczna) konwersja typów:

Niejawna konwersja typów w JavaScript zachodzi automatycznie w pewnych sytuacjach, gdy wartości są używane w kontekście, który oczekuje innych typów danych. Np. przy dodawaniu liczby do stringa, JavaScript automatycznie zamienia liczbę na string i wykonuje konkatenację.

Przykład niejawnej konwersji:

```
let a = 5;
let b = " 10";
let result = a + b; // Wartość result będzie " 510", ponieważ liczba 5 zostanie skonwertowana na string i nastąpi konkatenacja.
```

Jawna (ręczna) konwersja typów:

Jawna konwersja typów w JavaScript polega na ręcznym zmienianiu typu wartości za pomocą odpowiednich funkcji i operacji.

Przykłady jawnej konwersji:

Konwersja na liczbę:

```
let a = " 5";
let b = Number(a); // jawna konwersja na liczbę
console.log(typeof b); // " number "
```

### Funkcje

Deklaracja funkcji:

Funkcje w JavaScript mogą być deklarowane za pomocą słowa kluczowego `function`.

Przykład deklaracji funkcji:

```
function greet( name) {
    console.log( " Hello, " + name + " !");
}
```

Wyrażenie funkcyjne:

### Funkcje (cont)

> Funkcje w JavaScript mogą być przypisywane do zmiennych lub wartości.

Przykład wyrażenia funkcyjnego:

```
const greet = function(name) {
  console.log("Hello, " + name + "!");
};
```

Wywołanie funkcji:

Funkcje są wywoływane poprzez podanie nazwy funkcji i nawiasów () z odpowiednimi argumentami.

Przykład wywołania funkcji:

```
greet("John"); // Wywołanie funkcji greet z argumentem "John"
```

Funkcje strzałkowe:

Funkcje strzałkowe to skrócona składnia do tworzenia funkcji anonimowych.

Przykład funkcji strzałkowej:

```
const greet = (name) => {
  console.log("Hello, " + name + "!");
};
```

Zasięg (scope) zmiennych:

Funkcje w JavaScript mają własne zasięgi zmiennych. Zmienne zdefiniowane wewnątrz

funkcji są lokalne dla funkcji i nie są dostępne poza nią. Zmienne zdefiniowane poza

funkcją są globalne lub mają zasięg zdefiniowany przez blok kodu (w przypadku bloków

takich jak if czy for).

Wartość zwracana:

Funkcje w JavaScript mogą zwracać wartość za pomocą słowa kluczowego return.

Zwrócona wartość może być przypisana do zmiennej lub używana bezpośrednio w

innym kontekście.

Przykład zwracania wartości:

```
function add(a, b) {
  return a + b;
```

### Funkcje (cont)

```
> }
```

```
const sum = add(2, 3); // Przypisanie zwracanej wartości do zmiennej
```

```
console.log(sum); // Wyświetlenie wartości 5
```

Metoda call():

Metoda call() pozwala na wywołanie funkcji i przekazanie innego obiektu jako jej

kontekstu (this).

Przykład użycia metody call():

```
const person = {
  name: "John",
  greet: function() {
    console.log("Hello, " + this.name + "!");
  }
};
```

```
const anotherPerson = {
```

```
  name: "Jane"
```

```
};
```

```
person.greet.call(anotherPerson); // Wywołanie metody greet z kontekstem anotherPerson
```

Metoda apply():

Metoda apply() działa podobnie do metody call(), ale argumenty są przekazywane

jako tablica.

Przykład użycia metody apply():

```
const person = {
  name: "John",
  greet: function(greeting) {
    console.log(greeting + ", " + this.name + "!");
  }
};
```

```
const greetings = ["Hello", "Bonjour", "Hola"];
```

### Funkcje (cont)

> `person.greet.apply(person, greetings);` // Wywołanie metody `greet` z kontekstem `person`

i przekazanie tablicy `greetings` jako argumentów

Rekurencja:

Rekurencja to technika polegająca na wywoływaniu funkcji samej siebie. Jest często

używana do rozwiązywania problemów, które można zdefiniować w sposób rekurencyjny.

Przykład rekurencji:

```
function countdown(n) {
```

```
  if (n === 0) {
    console.log("Lift off!");
```

```
  } else {
```

```
    console.log(n);
```

```
    countdown(n - 1);
```

```
  }
```

```
}
```

```
countdown(5);
```

 // Wywołanie funkcji `countdown` z argumentem 5

Funkcje przekazywane jako argument do funkcji (funkcje zwrotne lub funkcje

wyższego rzędu) pozwalają na wykonanie pewnych działań wewnątrz funkcji

na podstawie logiki zawartej w przekazanej funkcji.

Oto przykład, który ilustruje przekazywanie funkcji jako argumentów:

```
function sayHello() {
```

```
  console.log("Hello!");
```

```
}
```

```
function runCallback(callback) {
```

```
  console.log("Running callback...");
```

```
  callback(); // Wywołanie przekazanej funkcji
```

```
}
```

```
runCallback(sayHello);
```

 // Przekazanie funkcji `sayHello` jako argument do funkcji

```
runCallback
```

### Funkcje (cont)

> W powyższym przykładzie funkcja `sayHello` jest przekazywana jako argument do funkcji

`runCallback`. Gdy funkcja `runCallback` jest wywoływana, przekazana funkcja `sayHello` jest

wywoływana wewnątrz funkcji `runCallback`. To umożliwia wykonanie dodatkowych działań

wewnątrz `runCallback`, takich jak logowanie, przed lub po wywołaniu przekazanej funkcji.

Funkcje anonimowe w języku JavaScript to funkcje, które nie posiadają nazwy.

Są one definiowane bez podawania nazwy funkcji i mogą być przypisywane do

zmiennych lub przekazywane jako argumenty do innych funkcji.

Funkcje anonimowe

są często wykorzystywane w przypadkach, gdy potrzebujemy jednorazowej funkcji lub

gdy chcemy przekazać logikę funkcji bez konieczności nadawania jej nazwy.

Przypisanie funkcji anonimowej do zmiennej:

```
const sayHello = function() {
```

```
  console.log("Hello!");
```

```
};
```

```
sayHello();
```

 // Wywołanie funkcji anonimowej

Przekazanie funkcji anonimowej jako argumentu:

```
function runCallback(callback) {
```

```
  console.log("Running callback...");
```

```
  callback(); // Wywołanie przekazanej funkcji anonimowej
```

```
}
```

```
runCallback(function() {
```

```
  console.log("Hello from anonymous function!");
```

```
});
```

Wykorzystanie funkcji anonimowej bezpośrednio w wyrażeniu:

```
setTimeout(function() {
```

```
  console.log("Delayed message");
```

```
}, 2000);
```



By sigeurd

[cheatography.com/signeur/](https://cheatography.com/signeur/)

Not published yet.

Last updated 2nd July, 2023.

Page 30 of 49.

Sponsored by **Readable.com**

Measure your website readability!

<https://readable.com>

### Obiekt document

```
getElementById():
const element = document.getElementById('myElement');
Ta funkcja zwraca element DOM o określonym identyfikatorze. Przykład pokazuje pobranie elementu o identyfikatorze "myElement".

getElementsByClassName():
const elements = document.getElementsByClassName('myClass');
Ta funkcja zwraca kolekcję elementów DOM, które mają określoną klasę. Przykład pokazuje pobranie elementów o klasie "myClass".

getElementsByTagName():
const elements = document.getElementsByTagName('p');
Ta funkcja zwraca kolekcję elementów DOM, które mają określony tag HTML. Przykład pokazuje pobranie wszystkich elementów <p>.

querySelector():
const element = document.querySelector('#myElement.myClass');
Ta funkcja zwraca pierwszy element DOM, który pasuje do podanego selektora CSS. Przykład pokazuje pobranie pierwszego elementu, który ma identyfikator "myElement" i klasę "myClass".

querySelectorAll():
const elements = document.querySelectorAll('.myClass');
Ta funkcja zwraca wszystkie elementy DOM, które pasują do podanego selektora CSS. Przykład pokazuje pobranie wszystkich elementów o klasie "myClass".

createElement():
const element = document.createElement('div');
Ta funkcja tworzy nowy element DOM o podanym tagu HTML. Przykład pokazuje utworzenie nowego elementu <div>.

appendChild():
parentElement.appendChild(newElement);
Ta funkcja dodaje nowy element jako dziecko do określonego elementu DOM.
```

### Obiekt document (cont)

```
> Przykład pokazuje dodanie nowego elementu jako dziecka do elementu nadrzędnego.

removeChild():
parentElement.removeChild(childElement);
Ta funkcja usuwa określony element potomny z danego elementu DOM.
Przykład pokazuje usunięcie określonego elementu potomnego z elementu nadrzędnego.

prepend():
parentElement.prepend(newElement);
Ta funkcja dodaje nowy element jako pierwsze dziecko do określonego elementu DOM.
Przykład pokazuje dodanie nowego elementu jako pierwszego dziecka do elementu nadrzędnego.

const parentElement = document.getElementById('myElement');
const newElement = document.createElement('div');
newElement.textContent = 'Hello, World!';
parentElement.prepend(newElement);

createElement():
const element = document.createElement('div');
Ta funkcja tworzy nowy element DOM o podanym tagu HTML.
Można również użyć tej funkcji do tworzenia innych rodzajów elementów, takich jak <p>, <span>, <img>, itp.
Przykład pokazuje tworzenie nowego elementu <div>.

const element = document.createElement('p');
element.textContent = 'Hello, World!';
Funkcja createElement() jest używana w połączeniu z innymi funkcjami, takimi jak appendChild() lub prepend(), aby dodać nowo utworzony element do drzewa DOM.
```





### Manipulowanie elementami html i css

```
setAttribute():
element.setAttribute('attributeName', 'attributeValue');
Ta funkcja ustawia wartość określonego atrybutu
na elemencie. Może być używana
do ustawienia dowolnego atrybutu, na przykład
class, id, src, href itp.
style:
element.style.property = 'value';
Można użyć właściwości style na elemencie, aby
zmieniać jego style CSS. Właściwości
style odzwierciedlają style inline dla danego
elementu.
classList:
element.classList.add('className');
element.classList.remove('className');
element.classList.toggle('className');
Właściwość classList umożliwia manipulowanie
klasami elementu. Można użyć metod
add(), remove() i toggle() do dodawania, usuwania i
przełączania klas na elemencie.
innerHTML obiektu document służy do odczytywania
i modyfikowania zawartości HTML
danego elementu. Przechowuje ona kod HTML
znajdujący się wewnątrz danego elementu.
Przykład użycia innerHTML do odczytania zawartości
HTML elementu:
const element = document.getElementById('myElement');
const htmlContent = element.innerHTML;
console.log(htmlContent);
Przykład użycia innerHTML do modyfikowania
zawartości HTML elementu:
const element = document.getElementById('myElement');
element.innerHTML = '<p>Nowa zawartość </p>';
```

### Własne funkcje konstruujące

W JavaScript możemy tworzyć własne funkcje konstruktorskie do tworzenia nowych obiektów na podstawie określonego wzorca. Funkcje te są używane do definiowania "klas" obiektów i tworzenia instancji obiektów.

Aby zdefiniować własną funkcję konstruktora, stosujemy CamelCase dla nazwy funkcji (zaczynamy wielką literą) i definiujemy jej konstruktor może zawierać parametry, które są przekazywane przy tworzeniu instancji obiektu.

Wewnątrz funkcji konstruktora możemy definiować właściwości obiektu, które będą różne dla każdej instancji, oraz metody, które będą wspólne dla wszystkich instancji.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        console.log('Hello, my name is ${this.name} and I'm ' +
            age + ' years old.');
```

### call apply bind

call, apply i bind to metody, które pozwalają na manipulację kontekstem (this) funkcji w JavaScript. Oto ich krótkie wyjaśnienie:

- call: Metoda call pozwala na wywołanie funkcji z określonym kontekstem (this) oraz przekazanie argumentów osobno.

```
function sayHello() {
    console.log('Hello, ${this.name}!');
```

### call apply bind (cont)

```
> }
const person = { name: 'John' };
sayHello.call(person); // Wywołanie funkcji sayHello z kontekstem person
- apply: Metoda apply działa podobnie do call, ale przekazuje argumenty jako tablicę.
function sayHello(greeting) {
  console.log(`${greeting}, ${this.name}!`);
}
const person = { name: 'John' };
sayHello.apply(person, ['Hello']); // Wywołanie funkcji sayHello z kontekstem person i argumentem 'Hello'
- bind: Metoda bind tworzy nową funkcję, która ma przypisany określony kontekst (this), ale nie jest natychmiast wywoływana. Zamiast tego zwraca funkcję, która może być wywołana później.
function sayHello() {
  console.log(Hello, ${this.name}!);
}
const person = { name: 'John' };
const greetPerson = sayHello.bind(person); // Utworzenie nowej funkcji greetPerson z przypisanym kontekstem person
greetPerson(); // Wywołanie funkcji greetPerson
Metody call i apply są używane, gdy chcemy natychmiastowo wywołać funkcję z określonym kontekstem i argumentami. Metoda bind jest przydatna, gdy chcemy utworzyć nową funkcję, która będzie miała przypisany kontekst i będzie mogła być wywołana w przyszłości.
Te trzy metody są często używane do manipulacji kontekstem (this) w JavaScript i pozwalają na większą elastyczność w programowaniu.
```

### Object

Właściwości obiektu:

- Object.prototype: Zwraca referencję do funkcji konstruktora, która została użyta do utworzenia obiektu.
- Object.prototype.hasOwnProperty(): Sprawdza, czy dany obiekt posiada własność o podanej nazwie.
- Object.prototype.isPrototypeOf(): Sprawdza, czy dana własność obiektu jest wyliczalna.

Funkcje obiektu:

- Object.prototype.constructor: Kopiuje wartości wszystkich wyliczalnych własności z jednego lub więcej obiektów źródłowych do obiektu docelowego.
- Object.prototype.keys(): Zwraca tablicę zawierającą nazwy wszystkich wyliczalnych własności obiektu.
- Object.prototype.values(): Zwraca tablicę zawierającą wartości wszystkich wyliczalnych własności obiektu.
- Object.prototype.entries(): Zwraca tablicę zawierającą tablice [klucz, wartość] dla wszystkich wyliczalnych własności obiektu.
- Object.freeze(): Zamraża obiekt, uniemożliwiając zmianę jego własności.
- Object.seal(): Zapieczętuje obiekt, uniemożliwiając dodawanie i usuwanie własności, ale umożliwia modyfikację istniejących własności.
- Object.create(): Tworzy nowy obiekt z określonym prototypem i opcjonalnymi własnościami.

Metody prototypu obiektu:

- toString(): Zwraca ciąg znaków reprezentujący obiekt.
  - hasOwnProperty(): Sprawdza, czy obiekt ma daną własność jako własną, a nie odziedziczoną.
  - valueOf(): Zwraca wartość pierwotną obiektu.
- Poniżej znajduje się przykład użycia niektórych z tych właściwości i funkcji:
- ```
const obj = { name: 'John', age: 30 };
```



### Object (cont)

```
> console.log(Object.keys(obj)); // Output: ['name', 'age']
console.log(Object.values(obj)); // Output: ['John', 30]
console.log(Object.entries(obj)); // Output: [['name', 'John'], ['age', 30]]
const newObj = Object.assign({}, obj);
console.log(newObj); // Output: { name: 'John', age: 30 }
const sealedObj = Object.seal(obj);
sealedObj.name = 'Jane';
delete sealedObj.age;
console.log(sealedObj); // Output: { name: 'Jane', age: 30 }
const frozenObj = Object.freeze(obj);
frozenObj.name = 'Jane';
console.log(frozenObj); // Output: { name: 'John', age: 30 }
console.log(obj.hasOwnProperty('name')); // Output: true
console.log(obj.hasOwnProperty('toString')); // Output: false
```

Gettery i settery to specjalne metody używane w języku JavaScript do kontroli dostępu do właściwości obiektów. Pozwalają na bardziej elastyczne zarządzanie właściwościami obiektu i umożliwiają wykonanie dodatkowej logiki podczas odczytu i zapisu wartości.

Getter jest metodą, która zwraca wartość właściwości obiektu. Jest definiowany za pomocą słowa kluczowego `get` i nazwy właściwości. Kiedy odwołujemy się do tej właściwości, getter jest automatycznie wywoływany i zwraca oczekiwany wynik.

Setter jest metodą, która ustawia wartość właściwości obiektu. Jest definiowany za pomocą słowa kluczowego `set` i nazwy właściwości. Kiedy przypisujemy wartość do tej właściwości, setter jest automatycznie wywoływany, co pozwala na wykonanie określonych operacji lub walidację przed zapisaniem wartości.

Oto przykład użycia gettera i settera:

```
const person = {
  firstName: "",
```

### Object (cont)

```
> lastName: "",

get fullName() {
  return this.firstName + ' ' + this.lastName;
},

set fullName(name) {
  const [firstName, lastName] = name.split(' ');
  this.firstName = firstName;
  this.lastName = lastName;
}
};
person.fullName = 'John Doe'; // Wywołanie settera
console.log(person.firstName); // Wyświetli "John"
console.log(person.lastName); // Wyświetli "Doe"
console.log(person.fullName); // Wyświetli "John Doe", getter automatycznie wywołany
```

`Object.defineProperty()` jest metodą wbudowaną w JavaScript, która służy do definiowania nowej właściwości na obiekcie lub modyfikowania istniejącej właściwości na obiekcie. Metoda ta umożliwia dokładne kontrolowanie konfiguracji właściwości, takich jak możliwość zapisu (`writable`), możliwość konfiguracji (`configurable`), możliwość wyliczania (`enumerable`) itd.

Przykład użycia `Object.defineProperty()`:

```
const obj = {};
// Definiowanie nowej właściwości "name" na obiekcie
Object.defineProperty(obj, 'name', {
  value: 'John',
  writable: false, // Wartość nie może być zmieniana
  enumerable: true, // Właściwość jest wyliczalna
  configurable: true // Właściwość może być modyfikowana lub usunięta
});
```

### Object (cont)

```
> console.log(obj.name); // John
obj.name = 'Jane'; // Nie spowoduje zmiany, ponieważ writable jest
ustawione na false
console.log(obj.name); // John
// Modyfikowanie istniejącej właściwości
Object.defineProperty(obj, 'name', {
  value: 'Jane' // Zmiana wartości
});
console.log(obj.name); // Jane
```

### Enumeracja

Enumeracja obiektu w języku JavaScript polega na przejściu przez wszystkie jego właściwości. Istnieją różne metody umożliwiające iterację po właściwościach obiektu i wykonanie określonych operacji dla każdej z nich.

Najpopularniejszymi metodami iteracji po właściwościach obiektu są:

**Pętla for...in:** Pętla for...in iteruje po wszystkich wyliczalnych właściwościach obiektu (w tym również w właściwościach dziedziczonych). Dla każdej właściwości można wykonać określone operacje.

```
const obj = { a: 1, b: 2, c: 3 };
for (const prop in obj) {
  console.log(prop + ': ' + obj[prop]);
}
// Wynik:
// a: 1
// b: 2
// c: 3
```

**Metoda Object.keys():** Metoda Object.keys() zwraca tablicę zawierającą wszystkie własne właściwości wyliczalne obiektu. Można na niej iterować za pomocą pętli for...of.

```
const obj = { a: 1, b: 2, c: 3 };
for (const prop of Object.keys(obj)) {
```

### Enumeracja (cont)

```
> console.log(prop + ': ' + obj[prop]);
}
// Wynik:
// a: 1
// b: 2
// c: 3
Metoda Object.entries(): Metoda Object.entries() zwraca tablicę
zawierającą
tablice [klucz, wartość] dla każdej wyliczalnej własnej właściwości
obiektu. Można
na niej iterować za pomocą pętli for...of.
const obj = { a: 1, b: 2, c: 3 };
for (const [key, value] of Object.entries(obj)) {
  console.log(key + ': ' + value);
}
// Wynik:
// a: 1
// b: 2
// c: 3
Te metody iteracji pozwalają na dostęp do właściwości obiektu i
wykonanie operacji
na nich. Warto zauważyć, że kolejność iteracji nie jest gwarantowana,
więc nie należy
polegać na porządku, w jakim właściwości są zwracane.
```

### Proxy API

Proxy API to funkcjonalność dostępna w języku JavaScript, która umożliwia przechwytywanie i dostosowywanie zachowań podstawowych operacji na obiektach. API to dostarcza możliwość tworzenia proxy (osłonki) wokół istniejących obiektów i definiowania niestandardowych operacji, które mają być wykonywane przy różnych interakcjach z tymi obiektami. Oto kilka cech i przykładów użycia Proxy API:

- Przechwytywanie dostępu do właściwości:



### Proxy API (cont)

> Proxy pozwala przechwycić próby odczytu (get) i zapisu (set) właściwości obiektu oraz reagować na nie. Przykład:

```
const obj = {
  name: "John",
  age: 30
};
const proxy = new Proxy(obj, {
  get(target, property) {
    console.log(Odczytano wartość właściwości " ${property} ");
    return target[property];
  },
  set(target, property, value) {
    console.log(Ustawiono wartość właściwości " ${property} " na "${value}");
    target[property] = value;
  }
});
console.log(proxy.name); // Odczytano wartość właściwości "name", wynik: "John"
proxy.age = 40; // Ustawiono wartość właściwości "age" na "40"
```

- Przechwytywanie wywołań metod:

Proxy pozwala przechwycić wywołania metod na obiekcie i dostosować ich zachowanie.

Przykład:

```
const obj = {
  greet(name) {
    console.log(Witaj, ${name}!);
  }
};
const proxy = new Proxy(obj, {
  apply(target, thisArg, argumentsList) {
```

### Proxy API (cont)

```
console.log(Wywołano metodę " ${argumentsList[0]} ");
    return target.apply(thisArg, argumentsList);
  }
});
proxy.greet("John"); // Wywołano metodę "greet", wynik: "Witaj, John!"
```

- Inne operacje i przechwytywanie zdarzeń:

Proxy umożliwia przechwytywanie innych operacji, takich jak sprawdzanie istnienia właściwości (has), iteracja (enumerate), usuwanie właściwości (deleteProperty) itp.

Można również przechwycić zdarzenia dotyczące proxy za pomocą specjalnych metod,

takich jak getPrototypeOf, setPrototypeOf, isExtensible, preventExtensions, getOwnPropertyDescriptor, defineProperty, ownKeys itp.

Proxy API dostarcza potężny mechanizm do tworzenia niestandardowych zachowań dla

obiektów w języku JavaScript. Można go wykorzystać do wprowadzania walidacji, filtrowania,

zależności, śledzenia zmian i wielu innych operacji na obiektach.

### Reflect API

Reflect API to zestaw metod i funkcji, które dostarczają interfejsu programistycznego do operacji na obiektach w języku JavaScript. API to umożliwia wykonywanie różnych operacji na obiektach, takich jak tworzenie, modyfikacja, usuwanie właściwości, wywoływanie funkcji itp.

Oto kilka cech i przykładów użycia Reflect API:

1. Metody do tworzenia i manipulowania obiektami:

- Reflect.construct(target, argumentsList):  
Tworzy nowy obiekt na podstawie danego konstruktora i argumentów.

- Reflect.getPrototypeOf(object):  
Zwraca prototyp danego obiektu.

- Reflect.setPrototypeOf(object, prototype):  
Ustawia nowy prototyp dla danego obiektu.

2. Metody do manipulacji właściwościami obiektów:

- Reflect.get(object, propertyKey):  
Zwraca wartość właściwości danego obiektu.

- Reflect.set(object, propertyKey, value):  
Ustawia wartość właściwości danego obiektu.



### Reflect API (cont)

- `Reflect.has(object, propertyKey)`: Sprawdza, czy dany obiekt zawiera właściwość o podanym kluczu.
- `Reflect.deleteProperty(object, propertyKey)`: Usuwa właściwość z danego obiektu.

3. Metody do wywoływania funkcji:

- `Reflect.apply(target, thisArgument, argumentsList)`: Wywołuje daną funkcję z podanym kontekstem (`this`) i listą argumentów.
- `Reflect.construct(target, argumentsList, newTarget)`: Wywołuje konstruktor z podanymi argumentami i opcjonalnie nowym obiektem docelowym.

4. Obsługa zdarzeń:

- `Reflect.get(target, propertyKey, receiver)`: Obsługuje pobieranie wartości właściwości, umożliwiając działanie na pośredniku (`receiver`).
- `Reflect.set(target, propertyKey, value, receiver)`: Obsługuje ustawianie wartości właściwości, umożliwiając działanie na pośredniku (`receiver`).

Reflect API dostarcza bardziej jednolitej i przejrzystej składni do operacji na obiektach w porównaniu do tradycyjnych operacji, takich jak `obj.property` lub `obj["property"]`. Zapewnia również większą elastyczność i kontrolę nad operacjami na obiektach, umożliwiając manipulację właściwościami, wywoływanie funkcji i zarządzanie prototypami.

### Generatory

Generatory w języku JavaScript to specjalny rodzaj funkcji, które pozwalają na wznawialne wykonywanie się kodu. Generator jest tworzony przy użyciu słowa kluczowego `function`, ale zamiast zwykłego `return`, używa się instrukcji `yield`, która zwraca wartość, ale nie kończy działania funkcji.

Oto kilka cech generatorów:

Wznawialne wykonanie: Główną cechą generatorów jest możliwość wznawialnego wykonania kodu. Po wywołaniu generatora, nie jest on uruchamiany od początku do końca, ale zatrzymuje się na każdej instrukcji `yield`, zwracając wartość i zapamiętując swój stan. Kolejne wywołanie generatora wznawia jego wykonanie od momentu, w którym został zatrzymany.

### Generatory (cont)

> Wydajne zarządzanie pamięcią: Generatory pozwalają na leniwe generowanie wartości, co oznacza, że nie muszą generować wszystkich wartości od razu. Mogą generować wartości na żądanie, co jest szczególnie przydatne w przypadku generowania dużych kolekcji danych.

Iterator: Generatory są jednocześnie iteratorami. Implementują protokół iteratora, co oznacza, że można ich używać w pętlach `for...of` lub wykorzystywać do iteracji po danych.

Potokowe przetwarzanie danych: Generatory mogą być łączone w tzw. potoki, co umożliwia przetwarzanie danych w sposób sekwencyjny. Wynik jednego generatora może być przekazywany do innego generatora, co pozwala na składanie różnych operacji na danych.

Przykład użycia generatora:

```
function* generateNumbers() {
  let number = 1;
  while (true) {
    yield number;
    number++;
  }
}
```

```
const numberGenerator = generateNumbers();
console.log(numberGenerator.next().value); // 1
console.log(numberGenerator.next().value); // 2
console.log(numberGenerator.next().value); // 3
```

W powyższym przykładzie mamy generator `generateNumbers`, który generuje kolejne liczby naturalne. Po każdym wywołaniu `numberGenerator.next()`, otrzymujemy obiekt z wartością generowaną przez generator. Dzięki temu możemy generować kolejne liczby w nieskończoność, zatrzymując się i wznawiając wykonanie generatora w dowolnym momencie.

Generatory są użytecznym narzędziem do manipulacji danymi i zarządzania ich

### Generatory (cont)

> przepływem. Pozwalają na bardziej elastyczną kontrolę nad generowaniem i przetwarzaniem danych w języku JavaScript.

### Moduły

Moduły w języku JavaScript są mechanizmem, który umożliwia podział kodu na mniejsze, samodzielne jednostki. Pozwala to na lepszą organizację i zarządzanie kodem, a także ułatwia jego ponowne wykorzystanie.

W JavaScript istnieją różne metody implementacji modułów, takie jak moduły CommonJS, moduły AMD (Asynchronous Module Definition) i moduły ES6 (ECMAScript 6). Moduły ES6 są wbudowanym mechanizmem języka od wersji ES6, który zapewnia prostą i wydajną składnię do definiowania i eksportowania modułów.

Główne cechy modułów ES6 to:

**Deklaracja modułu:** Moduł jest tworzony poprzez utworzenie pliku z rozszerzeniem ".js", który zawiera kod JavaScript. Moduł jest oddzielną jednostką, która posiada swoje własne zakresy i niezależność od innych modułów.

**Eksportowanie danych:** Eksportowanie danych z modułu odbywa się za pomocą słowa kluczowego `export`. Możemy eksportować funkcje, zmienne, stałe, klasy i wiele innych.

**Przykład eksportu funkcji:**

```
export function sayHello() {
  console.log('Hello!');
}
```

**Importowanie danych:** Importowanie danych z innego modułu odbywa się za pomocą słowa kluczowego `import`. Importujemy określone elementy z modułu i przypisujemy je do zmiennych.

**Przykład importu funkcji:**

```
import { sayHello } from './module.js';
sayHello(); // Wywołanie funkcji z innego modułu
```

**Domyślne eksportowanie:** Możemy także domyślnie eksportować jedno element z modułu. Może to być funkcja, klasa lub inny obiekt.

### Moduły (cont)

> Przykład domyślnego eksportu klasy:

```
export default class MyClass {
  constructor() {
    // Konstruktor klasy
  }
}
```

Moduły w JavaScript umożliwiają tworzenie modularnej struktury kodu, gdzie poszczególne części aplikacji są łatwo odizolowane i mogą być używane i modyfikowane niezależnie od siebie.

Jest to przydatne w większych projektach, które wymagają modularności i skalowalności.

### isin

Jeśli chodzi o sprawdzenie, czy określona wartość jest obecna w danym tekście, obiekcie lub innym zbiorze danych, istnieje kilka sposobów:

- **Metoda `includes()`:** Metoda `includes()` jest dostępna dla typów danych takich jak ciągi znaków (stringi) i tablice. Służy do sprawdzenia, czy dana wartość jest zawarta w danym ciągu znaków lub tablicy. Zwraca wartość logiczną `true` lub `false`.

**Przykład (użycie na stringu):**

```
const text = "Lorem ipsum dolor sit amet";
console.log(text.includes("ipsum")); // true
console.log(text.includes("foo")); // false
```

- **Operator `in`:** Operator `in` jest używany do sprawdzenia, czy dany klucz istnieje w obiekcie. Może być stosowany na obiektach i tablicach. Zwraca wartość logiczną `true` lub `false`.

**Przykład (użycie na obiekcie):**

```
const person = { name: "John", age: 30 };
console.log("name" in person); // true
console.log("address" in person); // false
```

- **Metoda `indexOf()`:** Metoda `indexOf()` jest dostępna dla ciągów znaków (stringów) i

### isin (cont)

> tablic. Służy do znalezienia indeksu pierwszego wystąpienia danej wartości w ciągu

znaków lub tablicy. Jeśli wartość nie jest znaleziona, zwraca -1.

Przykład (użycie na stringu):

```
const text = "Lorem ipsum dolor sit amet";
```

```
console.log(text.indexOf("dolor")); // 12
```

```
console.log(text.indexOf("foo")); // -1
```

- Metoda `hasOwnProperty()`: Metoda `hasOwnProperty()` jest dostępna dla obiektów.

Służy do sprawdzenia, czy obiekt posiada określony klucz jako własność. Zwraca wartość logiczną `true` lub `false`.

Przykład (użycie na obiekcie):

```
const person = { name: "John", age: 30 };
```

```
console.log(person.hasOwnProperty("name")); // true
```

```
console.log(person.hasOwnProperty("address")); // false
```

- Metoda `match()`: Metoda `match()` jest dostępna dla ciągów znaków (stringów) i służy do znalezienia dopasowań do określonego wzorca za pomocą wyrażenia regularnego.

Jeśli dopasowanie jest znalezione, zwraca tablicę dopasowanych wartości; w przeciwnym razie zwraca `null`.

Przykład (użycie na stringu):

```
const text = "Lorem ipsum dolor sit amet";
```

```
console.log(text.match(/ipsum/)); // ["ipsum"]
```

```
console.log(text.match(/foo/)); // null
```

- Metoda `find()`: Metoda `find()` jest dostępna dla tablic i służy do znalezienia pierwszego

elementu, który spełnia określony warunek w funkcji zwrotnej.

Zwraca wartość znalezionej elementu lub `undefined`, jeśli nie znaleziono pasującego elementu.

Przykład (użycie na tablicy):

```
const numbers = [1, 2, 3, 4, 5];
```

```
const found = numbers.find(num => num > 3);
```

```
console.log(found); // 4
```

### isin (cont)

> - Operator `some()` (dla tablic): Operator `some()` działa na tablicach i sprawdza, czy

przynajmniej jeden element w tablicy spełnia określony warunek w funkcji zwrotnej.

Zwraca wartość logiczną `true` lub `false`.

Przykład (użycie na tablicy):

```
const numbers = [1, 2, 3, 4, 5];
```

```
const hasEvenNumber = numbers.some(num => num % 2 === 0);
```

```
console.log(hasEvenNumber); // true
```

### Symbol

Symbol jest jednym z typów danych w języku

JavaScript, wprowadzonym w

ECMAScript 2015 (ES6). Jest to wartość unikalna i

niezmienialna, która może być używana

jako klucz identyfikujący właściwości obiektów.

Symbole są przydatne w kontekście

tworzenia i zarządzania własnymi właściwościami

obiektów, które są ukryte i niekolidują

z innymi właściwościami.

Oto kilka cech i zastosowań symboli:

Unikalność: Każdy symbol jest unikalny i różni

się od innych symboli. Nawet jeśli dwa

symbole mają taką samą nazwę, są one od siebie

różne.

Prywatność: Symbole są przydatne do tworzenia

ukrytych lub prywatnych właściwości

obiektów. Można je wykorzystać do dodania

własności do obiektu, które nie będą

konfliktować z innymi właściwościami i nie będą

przypadkowo nadpisywane.

Użycie jako klucze: Symbole mogą być używane jako

klucze w obiektach.

Dzięki temu można tworzyć niestandardowe

właściwości, które nie zostaną

przypadkowo nadpisane lub skonfliktowane z

innymi właściwościami.

Oto przykładowe użycie symboli:

```
// Tworzenie symbolu
```

```
const symbol = Symbol();
```

```
// Używanie symbolu jako klucza
```

```
const obj = {
```

```
  [symbol]: 'Wartość symbolu'
```



### Symbol (cont)

```
> };
// Dostęp do wartości za pomocą symbolu
console.log(obj[symbol]); // Wyświetli: 'Wartość symbolu'
// Porównywanie symboli
const symbol1 = Symbol('Symbol');
const symbol2 = Symbol('Symbol');
console.log(symbol1 === symbol2); // false
console.log(symbol1 == symbol2); // false
Symbole są przydatne w przypadkach, gdy potrzebujemy
dodatkowej warstwy
prywatności, unikalnych identyfikatorów lub niestandardowych
właściwości obiektów.
let obj = {
  a: 10
};
obj[symbol1] = 1000;
console.log(obj[symbol1]); // 1000
for (const v in obj) console.log(v); // a
console.log( Object.getOwnPropertySymbols(obj) );
```

### Promise

Obiekt Promise w języku JavaScript jest używany do obsługi asynchronicznych operacji i zarządzania ich wynikami. Promise reprezentuje wartość, która może być dostępna teraz, w przyszłości lub nigdy. Jest to sposób eleganckiego rozwiązania problemów związanych z asynchronizacją, takich jak pobieranie danych z serwera, wykonywanie żądań sieciowych czy operacje na plikach.

Główne cechy Promise:

Stan: Promise może znajdować się w jednym z trzech stanów:

- Pending (oczekujący): Początkowy stan, w którym Promise oczekuje na wykonanie.
- Fulfilled (zrealizowany): Stan, w którym Promise został z powodzeniem zakończony i zwraca wartość.

### Promise (cont)

> - Rejected (odrzucony): Stan, w którym Promise zakończył się niepowodzeniem i zwraca błąd.

Metody: Promise udostępnia kilka przydatnych metod do obsługi asynchroniczności:

- then(): Wykonuje się, gdy Promise zostanie zrealizowany, i obsługuje wynik sukcesu.
- catch(): Wykonuje się, gdy Promise zostanie odrzucony, i obsługuje błąd.
- finally(): Wykonuje się bez względu na wynik Promise, zarówno po zrealizowaniu, jak i odrzuceniu.
- Promise.all(): Pozwala na równoczesne wykonywanie wielu Promise i czeka na ich zakończenie.
- Promise.race(): Czeki na zakończenie dowolnego z podanych Promise.

Przykład użycia Promise:

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Przykładowe dane';
      // Symulacja sukcesu
      resolve(data);
      // Symulacja błędu
      // reject(new Error("Wystąpił błąd"));
    }, 2000);
  });
};
fetchData()
  .then((data) => {
    console.log('Pomyślnie pobrane dane:', data);
  })
  .catch((error) => {
    console.error('Wystąpił błąd:', error);
  });
```

### Promise (cont)

```
> })
  .finally(() => {
    console.log('Operacja zakończona');
  });
```

W tym przykładzie tworzymy funkcję `fetchData`, która zwraca Promise. Wewnątrz

Promise symulujemy opóźnienie za pomocą funkcji `setTimeout`. W zależności od rezultatu symulujemy sukces lub błąd, używając funkcji `resolve` i `reject`.

Następnie używamy metody `then()` do obsługi wyniku sukcesu i metody `catch()` do obsługi błędu. Metoda `finally()` jest wywoływana bez względu na wynik Promise.

Dzięki obiektowi Promise możemy w łatwy sposób zarządzać asynchronicznymi operacjami, a także w ładny sposób obsługi.

### Destrukturyzacja

Destrukturyzacja (destrukcyjne przypisanie) to składnia w JavaScript, która umożliwia wygodne i eleganckie wyciąganie wartości z obiektów i tablic oraz przypisywanie ich do zmiennych. Dzięki destrukcji można łatwo dostępować do właściwości obiektów lub elementów tablicy bez konieczności odwoływania się do nich za pomocą tradycyjnych notacji.

Destrukturyzacja obiektów:

```
const person = { name: 'John', age: 30 };
// Wyciąganie wartości z obiektu do zmiennych
const { name, age } = person;
console.log(name); // 'John'
console.log(age); // 30
```

Destrukturyzacja tablic:

```
const numbers = [1, 2, 3, 4, 5];
// Wyciąganie wartości z tablicy do zmiennych
const [first, second, ...rest] = numbers;
console.log(first); // 1
console.log(second); // 2
```

### Destrukturyzacja (cont)

```
> console.log(rest); // [3, 4, 5]
```

Destrukturyzacja może być również stosowana w bardziej złożonych przypadkach,

takich jak destrukcyjna destrukcja zagnieżdżonych obiektów lub tablic, przypisywanie

domyślnych wartości, ignorowanie niektórych elementów itp.

Przykład destrukcyjnej destrukcji zagnieżdżonych obiektów:

```
const person = { name: 'John', age: 30, address: { city: 'New York', country: 'USA' } };
// Wyciąganie zagnieżdżonych wartości
```

```
const { name, address: { city } } = person;
console.log(name); // 'John'
console.log(city); // 'New York'
```

Przykład destrukcyjnej destrukcji z domyślnymi wartościami:

```
const person = { name: 'John' };
// Przypisanie domyślnych wartości, jeśli nie ma odpowiedniego
klucza w obiekcie
const { name, age = 30 } = person;
console.log(name); // 'John'
console.log(age); // 30
```

Destrukturyzacja jest bardzo przydatnym narzędziem w JavaScript, które pozwala

na bardziej ekspresywny i zwięzły kod, szczególnie przy manipulacji obiektami i tablicami.

### ECMAScript 6 (ES6)

- Zmienne `let` i `const`:

```
let x = 5;
const PI = 3.14;
```

- Bloki z zakresem (block scope):

```
if (true) {
  let x = 10;
  console.log(x); // 10
}
```

```
console.log(x); // Error: x is not defined
```



### ECMAScript 6 (ES6) (cont)

```
> - Szablony łańcuchowe (Template literals):
const name = 'John';
const message = Hello, ${name}!;
console.log(message); // Hello, John!

- Funkcje strzałkowe (Arrow functions):
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5

- Parametry domyślne:
function greet(name = 'Guest') {
  console.log(Hello, ${name}!);
}
greet(); // Hello, Guest!
greet('John'); // Hello, John!

- Domyślne wartości obiektów:
function getUserInfo({ name = 'Anonymous', age = 0 } = {}) {
  console.log(Name: ${name}, Age: ${age});
}
getUserInfo(); // Name: Anonymous, Age: 0
getUserInfo({ name: 'John', age: 25 }); // Name: John, Age: 25

- Destructuryzacja (Destructuring):
const person = {
  name: 'John',
  age: 30,
  country: 'USA'
};
const { name, age } = person;
console.log(name); // John
console.log(age); // 30
```

### ECMAScript 6 (ES6) (cont)

```
> - Moduły (Modules):
// file1.js
export const PI = 3.14;
export function multiply(a, b) {
  return a * b;
}
// file2.js
import { PI, multiply } from './file1.js';
console.log(PI); // 3.14
console.log(multiply(2, 3)); // 6

- Klasy (Classes):
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(Hello, ${this.name}!);
  }
}
const person = new Person('John');
person.sayHello(); // Hello, John!

- Rest parameters: umożliwia przekazywanie dowolnej liczby
argumentów do funkcji
w postaci tablicy. Pozwala to na bardziej elastyczne i dynamiczne
definiowanie funkcji.
function sum(...numbers) {
  return numbers.reduce((acc, num) => acc + num, 0);
}
console.log(sum(1, 2, 3)); // 6

Kiedy funkcja zostanie wywołana, wszystkie przekazane argumenty
zostaną zebrane
```



By **sigeurd**  
[cheatography.com/signeur/](https://cheatography.com/signeur/)

Not published yet.  
 Last updated 2nd July, 2023.  
 Page 42 of 49.

Sponsored by **Readable.com**  
 Measure your website readability!  
<https://readable.com>

### ECMAScript 6 (ES6) (cont)

> w tablicę i przypisane do parametru parametry. Ta tablica może mieć dowolną liczbę elementów, w zależności od liczby przekazanych argumentów.

Przykład użycia rest parameters:

```
function sumuj(...liczby) {
  let suma = 0;
  for (let liczba of liczby) {
    suma += liczba;
  }
  return suma;
}
console.log(sumuj(1, 2, 3, 4)); // Output: 10
console.log(sumuj(5, 10, 15)); // Output: 30
console.log(sumuj(2)); // Output: 2
console.log(sumuj()); // Output: 0
```

Rest parameters można również wykorzystać w połączeniu z innymi parametrami funkcji. Jednak rest parameters musi być ostatnim parametrem w definicji funkcji, ponieważ zbiera wszystkie pozostałe argumenty.

```
function foo(a, b, ...rest) {
  console.log('a:', a);
  console.log('b:', b);
  console.log('reszta:', rest);
}
foo(1, 2, 3, 4, 5);
// Output:
// a: 1
// b: 2
// reszta: [3, 4, 5]
```

- Spread operator: Operator rozproszenia (spread operator) w JavaScript jest oznaczony

### ECMAScript 6 (ES6) (cont)

> trzema kropkami (...) i służy do rozwinięcia (rozproszenia) elementów z kolekcji (np. tablicy) lub obiektu w miejscach, gdzie oczekiwane są oddzielne elementy. Operator rozproszenia umożliwia wygodne kopiowanie, łączenie i rozszerzanie danych.

Oto kilka przykładów zastosowania operatora rozproszenia:

Rozproszenie tablicy:

```
const numbers = [1, 2, 3];
const copiedNumbers = [...numbers];
console.log(copiedNumbers); // [1, 2, 3]
```

Łączenie tablic:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArray = [...arr1, ...arr2];
console.log(combinedArray); // [1, 2, 3, 4, 5, 6]
```

Rozproszenie obiektu:

```
const person = { name: 'John', age: 30 };
const copiedPerson = { ...person };
console.log(copiedPerson); // { name: 'John', age: 30 }
```

Przekazanie argumentów do funkcji:

```
const numbers = [1, 2, 3, 4, 5];
const maxNumber = Math.max(...numbers);
console.log(maxNumber); // 5
```

Tworzenie kopii obiektu z modyfikacją:

```
const person = { name: 'John', age: 30 };
const modifiedPerson = { ...person, age: 40 };
console.log(modifiedPerson); // { name: 'John', age: 40 }
```

Operator rozproszenia jest potężnym narzędziem, które umożliwia łatwe manipulowanie danymi w JavaScript. Może być stosowany na różnych typach danych, takich jak tablice, obiekty, stringi itp.



### ECMAScript 6 (ES6) (cont)

> - Rest parameters i spread operator to dwie różne funkcjonalności w języku JavaScript,

ale mają podobną składnię i korzystają z tych samych symboli: trzech kropek (...).

Rest parameters (...) używane jest w definicji funkcji do zbierania dowolnej liczby

argumentów i pakowania ich w tablicę.

Spread operator (...) używany jest w miejscu wywołania funkcji, tablicy lub obiektu

do rozwinięcia jego elementów i przekazania ich jako pojedyncze argumenty lub

elementy do innej funkcji, tablicy lub obiektu.

### ECMAScript 2018 (ES9)

- Rest/Spread Properties: Rest/Spread Properties umożliwiają rozwinięcie i zbieranie

właściwości obiektów. Przykład:

```
const person = { name: 'John', age: 30, city: 'New York' };
const { name, ...rest } = person;
```

```
console.log(name); // "John"
```

```
console.log(rest); // { age: 30, city: 'New York' }
```

- Promise.finally(): Dodano metodę finally() do obiektu Promise, która pozwala zdefiniować

blok kodu, który zostanie wykonany niezależnie od tego, czy Promise zakończy się sukcesem,

czy odrzuceniem. Przykład:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error))
  .finally(() => console.log('Zakończono żądanie'));
```

- Asyncroniczne iterowanie po obiektach:

Wprowadzono możliwość asynchronicznego iterowania po obiektach przy użyciu pętli for-await-of. Przykład:

```
const obj = { a: 1, b: 2, c: 3 };
(async () => {
  for await (const value of Object.values(obj)) {
    console.log(value);
  }
})();
```

### ECMAScript 2018 (ES9) (cont)

```
> }
```

```
})();
```

- Obiekty RegExp z grupami named capture: Dodano możliwość nazwanych grup

przechwytywania w wyrażeniach regularnych. Przykład:

```
const regex = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;
```

```
const match = regex.exec('2021-09-30');
```

```
console.log(match.groups.year); // "2021"
```

```
console.log(match.groups.month); // "09"
```

```
console.log(match.groups.day); // "30"
```

- Async Iteration: Umożliwia iterację po obiektach asynchronicznych, takich jak obiekty

generujące asynchroniczne wartości. Wykorzystuje nowy interfejs

Symbol.asyncIterator i

for-await-of do obsługi iteracji. Przykład:

```
async function* getData() {
  yield await fetchData1();
  yield await fetchData2();
  yield await fetchData3();
}
(async () => {
  for await (const data of getData()) {
    console.log(data);
  }
})();
```

### ECMAScript 2020 (ES11)

Oto lista niektórych nowych funkcji i zasad wprowadzonych w ECMAScript 2020 (ES11):

- Optional Chaining (Optional Property Access):

Operator ?. pozwala na bezpieczne odwołanie się do właściwości obiektu, nawet

jeśli wcześniejsze właściwości są

undefined lub null. Przykład:

```
const user = {
```



### ECMAScript 2020 (ES11) (cont)

```
> name: 'John',
  address: {
    city: 'New York'
  }
};
console.log(user.address?.city); // 'New York'
console.log(user.address?.street); // undefined
```

- Nullish Coalescing Operator: Operator ?? pozwala na wybór wartości domyślnej tylko wtedy, gdy wartość jest null lub undefined, a nie w przypadku innych "fałszywych" wartości, takich jak false, 0 czy pusty łańcuch. Przykład:

```
const name = null;
const defaultName = 'John';
console.log(name ?? defaultName); // 'John'
const count = 0;
const defaultCount = 10;
console.log(count ?? defaultCount); // 0
```

- BigInt: Typ danych BigInt został dodany do języka JavaScript, umożliwiając obsługę liczb całkowitych o dowolnej precyzji. Przykład:

```
const bigNumber = BigInt(12345678901234567890123456789012-34567890);
console.log(bigNumber); // 123456789012345678901234567890123-4567890n
```

- Promise.allSettled(): Metoda Promise.allSettled() pozwala na oczekiwanie na zakończenie wykonania wszystkich obietnic bez względu na ich rezultat (sukces, odrzucenie lub wyjątek). Przykład:

```
const promises = [
  Promise.resolve('Success'),
  Promise.reject('Error'),
  Promise.resolve('Another success')
];
```

### ECMAScript 2020 (ES11) (cont)

```
> Promise.allSettled(promises)
  .then(results => {
    results.forEach(result => {
      console.log(result.status); // 'fulfilled' lub 'rejected'
      console.log(result.value); // wartość lub powód odrzucenia
    });
  });
```

- String.matchAll(): Metoda matchAll() służy do iteracji po wszystkich dopasowaniach wzorca w łańcuchu znaków z wykorzystaniem wyrażeń regularnych. Przykład:

```
javascript
Copy code
const text = 'Hello World!';
const regex = /[a-z]/g;
const matches = text.matchAll(regex);
for (const match of matches) {
  console.log(match); // kolejne dopasowania
}
```

- globalThis: Obiekt globalThis dostarcza odniesienie do globalnego obiektu niezależnie od środowiska (przeglądarka, Node.js itp.). Dzięki temu można jednolicie odwoływać się do globalnego obiektu bez względu na kontekst. Na przykład:

```
console.log(globalThis.setTimeout === setTimeout); // true
```

import() dynamiczne importowanie: import() to asynchroniczna metoda, która umożliwia dynamiczne importowanie modułów w czasie wykonywania. Umożliwia to dynamiczną załadowanie kodu modułu na żądanie. Przykład:

```
const module = await import('./module.js');
```

- String.prototype.replaceAll(): Metoda replaceAll() zastępuje wszystkie wystąpienia podanej frazy w łańcuchu inną wartością. Przykład:

```
const text = 'Hello World!';
```



### ECMAScript 2020 (ES11) (cont)

```
> const newText = text.replaceAll('l', 'X');
console.log(newText); // 'HeXXo WorXd!'
```

### ECMAScript 2022 (ES13)

- Deklaracja pól klas

Do tej pory jeśli chcieliśmy zdefiniować pole klasy, mogliśmy to zrobić tylko poprzez konstruktor. Czyli ...

```
class Cat {
  constructor() {
    this.name = 'Szarek';
    this.age = 1;
  }
}
```

w ES13 możemy to zrobić w bardziej intuicyjny sposób. Możemy więc napisać:

```
class Cat {
  name = 'Szarek';
  age = 1;
}
```

- Prywatne metody i pola klasy

W JavaScript przy posługiwaniu się klasami nie mieliśmy możliwości tworzenia prywatnych pól klas ani prywatnych metod. Można było stosować konwencję, która mówiła, że prywatne "rzeczy" wyróżniamy przedrostkiem `_`. Wyglądało to w ten sposób:

```
class Cat {
  _name = 'Szarek';
  _age = 1;

  _getName() {
    return this._name;
  }
}
```

### ECMAScript 2022 (ES13) (cont)

> Jednak w zachowaniu kodu nie miało to żadnego odzwierciedlenia.

Więc jeśli spróbowaliśmy

pobrać wartość bezpośrednio z pola klasy:

```
const cat = new Cat();
console.log(cat._name); // -> 'Szarek'
```

... dostaliśmy bez problemu wartość pola. Z pomocą śpieszy nam oczywiście

ECMAScript 2022. Wprowadza ona prywatne pola klas oraz prywatne metody!

Prywatne metody oraz prywatne pola klas definiujemy od tej wersji za pomocą przedrostka

# Czyli nasz kod wyglądałby w ten sposób:

```
class Cat {
  #name = 'Szarek';
  #age = 1;

  #getName() {
    return this.#name;
  }

  getAge() {
    return this.#age;
  }
}
```

I teraz jeśli próbujemy się dostać do prywatnego pola klasy:

```
console.log(cat.#name); // -> SyntaxError: Private name #name is not defined.
```

do metody prywatnej:

```
console.log(cat.getName()); // -> TypeError: cat.getName is not a function
```

no i do metody publicznej:

```
console.log(cat.getAge()); // -> 1
```

... tak jak byśmy tego oczekiwali, dostajemy wartość.

- Operator `await` w najwyższym scope'ie

Temat dotyczy kombinacji operatorów `async` - `await`. W skrócie -

operatora `await` nie

mogliśmy używać poza funkcją oznaczoną operatorem `async`. Więc taki zapis generował



By sigeurd

[cheatography.com/signeur/](https://cheatography.com/signeur/)

Not published yet.

Last updated 2nd July, 2023.

Page 46 of 49.

Sponsored by [Readable.com](https://readable.com)

Measure your website readability!

<https://readable.com>

### ECMAScript 2022 (ES13) (cont)

```
> błąd:
const waitPrmise = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('RESOLVE!');
    }, 5000);
  });
};
await waitPrmise(); // -> // SyntaxError: await is only valid in async
functions
Nowa wersja, znosi takie ograniczenie i przykład powyżej będzie w
niej działał poprawnie.
4. Statyczne pola klas oraz statyczne prywatne metody
Od wersji ES13 możemy definiować statyczne prywatne pola klas
oraz statyczne
prywatne metody klas. Statyczne metody mają dostęp do innych
statycznych
(prywatnych i publicznych) metod poprzez this.
class Pet {
  static #count = 0;
  static getCount() {
    return this.#count;
  }
  constructor() {
    this.constructor.#incrementCount();
  }
  static #incrementCount() {
    this.#count++;
  }
}
const pet1 = new Pet();
const ppet2 = new Pet();
```

### ECMAScript 2022 (ES13) (cont)

```
> console.log(Pet.getCount()); // -> 2
```

- Statyczne bloki klas

ECMSScript 2022 pozwala nam definiować stateczne bloki kodu
wewnątrz klasy.

Deklarujemy je za pomocą: static {}. Statyczne bloki kodu są
wykonywane tylko raz
podczas inicjalizacji klasy.

Klasa może posiadać dowolną ilość statycznych bloków kodu. Bloki
te będą wykonywane
zgodnie z kolejnością ich deklaracji.

```
class Pet {
  static count = 0;
  static {
    this.count++;
  }
  static {
    this.count++;
  }
}
```

```
console.log(Pet.count); // -> 2
```

- Metoda at() dla tablic, stringów oraz obiektów TypedArray:

Jeśli chcemy pobrać element z tablicy o konkretnym indeksie to z
reguły używamy w tym celu nawiasów kwadratowych [], w ten
sposób:

```
const letters = ['a', 'b', 'c', 'd', 'e'];
console.log(letters[2]); // -> c
```

Pamiętamy, że indeksy zaczynają się od 0. Jeżeli chcemy pobrać
ostatni element tablicy,
to musimy zrobić takie fiku-miku:

```
console.log(letters[letters.length - 1]); // -> e
```

letters.lenght zwróci nam długość tablicy 5 i od tego odejmujemy 1,
żeby zgodził się indeks.

Dzięki nowej metodzie at() możemy wyciągać wartości z tablic w
bardziej naturalny sposób.



By **sigeud**  
[cheatography.com/sigeud/](https://cheatography.com/sigeud/)

Not published yet.  
 Last updated 2nd July, 2023.  
 Page 47 of 49.

Sponsored by **Readable.com**  
 Measure your website readability!  
<https://readable.com>



### ECMAScript 2022 (ES13) (cont)

```
> console.log(letters.at(0)); // -> a
console.log(letters.at(1)); // -> b
console.log(letters.at(-1)); // -> e
console.log(letters.at(-2)); // -> d
```

Jak widzisz pobieranie ostatniego elementu tablicy zostało uproszczone. Wystarczy, że do metody prześlemy ujemną liczbę, a pozycje będą brane od końca tablicy.

Metoda `at()` oprócz tablic została dodana także do string'ów oraz tablic typu `TypedArray`.

#### 7. Metoda `Object.hasOwn()`

Kto wie jak sprawdzić czy klasa posiada jakieś pole? Możemy to zrobić używając metody `hasOwnProperty` w ten sposób:

```
class Cat {
  name = 'Szarek';
}
const cat = new Cat();
cat.hasOwnProperty('name'); // -> true
cat.hasOwnProperty('age'); // -> false
```

Problem polega na tym, że tą metodę można w bardzo prosty sposób nadpisać i kompletnie zmienić jej logikę, więc nie powinniśmy takiego rozwiązania stosować w naszym kodzie.

Rozwiązaniem tego problemu jest skorzystanie z metody `call` na prototypie: `Object.prototype.hasOwnProperty.call(obj, propertyKey)` i tutaj jesteśmy już bardziej bezpieczni.

Jednak w najnowszej wersji JavaScript pojawiło się lepsze i bardziej eleganckie rozwiązanie. Dostaliśmy do dyspozycji metodę `Object.hasOwn()`, która wykonuje taką samą pracę jak `Object.prototype.hasOwnProperty.call(obj, propertyKey)`.

```
class Cat {
  name = 'Szarek';
}
```

### ECMAScript 2022 (ES13) (cont)

```
> const cat = new Cat();
Object.hasOwn(cat, 'name'); // -> true
Object.hasOwn(cat, 'age'); // -> false
```

- Metody `findLast` oraz `findLastIndex`

Teraz dostaliśmy do kolekcji dwie nowe metody tego typu, czyli `findLast` i `findLastIndex`.

Działają one tak samo, ale zwracają co innego:

```
const letters = [
  { value: 'a' },
  { value: 'b' },
  { value: 'c' },
  { value: 'b' },
  { value: 'd' },
];
letters.findLast((element) => element.value === 'b'); // -> { value: 'b' }
letters.findLastIndex((element) => element.value === 'b'); // -> 3
```

W odróżnieniu od `find` i `findIndex`, nowe metody szukają od końca tablicy.

Więc znajdą ostatni pasujący element.

Metoda `findLast` zwróci znaleziony element, natomiast metoda `findLastIndex` zwróci indeks znalezionej elementu.

